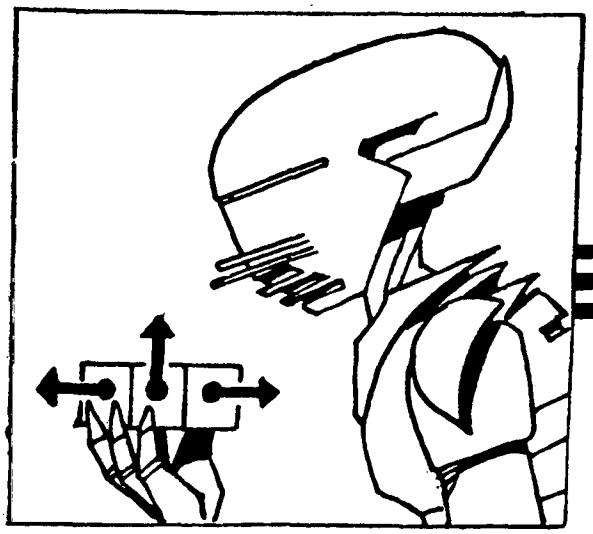

Query-io

We had a network server running in background exchanging ASCII strings with its remote clients. These strings included decimal numbers. When someone using the Lisp Listener on that machine set *print-base** *to 16, the background server started sending base 16 numbers to its clients which created errors.*

We have changed the server to always read and print in base 10. Why did the foreground process interfere with the background process and what can we do to prevent such interference?

Some Common Lisp implementations offer multiprocessing (sometimes called multiple stack groups) which is like multiprocessing on a conventional OS with one major difference. In Lisp, all processes are assumed to be working for the benefit of one user. A conventional OS assumes each process is working for the benefit of different, perhaps mutually hostile, users. This difference shows up in the attitudes towards sharing among processes.

Common Lisp encourages free and easy sharing. All Lisp objects defined by **defun**, **defmacro**, **defvar**, or **defparameter** are available for sharing by all processes. Therefore, it is relatively easy to establish something like ***print-base*** as a default for *all* system output functions. Your original problem was that your server's protocol was defined to use base 10, but your implementation of that protocol used the current default base which can change (as you found out).

A conventional OS takes pride in keeping its processes isolated. If you do want to share something among these processes, then usually you can share only data — not functions — and you typically must use special link editor directives or run time service calls to share that data.

There are really three questions here: "How do you share public modifications with the rest of the system?", "How do you keep private modifications from interfering with the rest of the system?", and "How do you prevent any modifications of others from interfering with you?".

First, all functions, macros, and global special variables you define will be available for sharing by everyone. If you define something with the same name and package as one that already exists in the system, then you have "shared" your redefinition with the whole world. You can avoid *accidental* sharing by creating your own package and then defining all your new functions, macros, and variables in that package.

If you need a private definition of a function or macro within the lexical scope of a certain block of source code, then use an **flet**, **labels**, or **macrolet** binding. Notice that this change affects only the source code enclosed by the binding. Other code called from within this source code will *not* see the change. There is no way in Common Lisp of privately modifying a function or macro definition over a dynamic scope. And, unfortunately, there is no way to protect yourself from the inadvertent modification by others of functions and macros you depend upon.

Things are different with variables. The form (**setf** *local-variable value*) where *local-variable* has not been declared special changes the value of *local-variable* defined in the enclosing lexical scope. And that's all it does. If there is no variable in the enclosing lexical scope named *local-variable*, it is an error. If there happens to be variables elsewhere in the code named *local-variable*, they are ignored.

However, the form (**setf** *special-variable value*) where *special-variable* has been declared special changes the value of the most local binding of *special-variable*. If there are no bindings of *special-variable*, then the global value of *special-variable* shared by all processes is changed. This characteristic suggests how to control who sees the modifications you make to variables and which modifications made by others that you see.

* If you want to share your modification of a global special variable with the whole system, then *set* that variable outside all bindings of that variable. For example:

```
(setf *print-base* 16.)
... code to print in hex ...
```

  The user sitting at the terminal may frequently set global variables because he presumably has a "global" view of what the system is doing. Conversely, a program running on that machine should never set a global variable because that program has little idea of what is going on elsewhere.

* If you want to privately modify a global special variable, then *bind* it. For example:

```
(let ((*print-base* 16.))
   ... code to print in hex ...)
```

  The user sitting at the terminal will seldom bother doing this, but this is the *normal* way global special variables should be changed within a program.

* If you want to protect your code from the ill-considered changes others might make to global variables while you are running, then bind the variable to itself. For example:

```
(let ((*print-base*  *print-base*))
   ... code the print in the default base ...)
```

  By binding the variable to itself, you have effectively taken a snapshot of its value at the time the **let** was entered. Alternately, you can bind it to the default value your code is expecting just in case someone had already modified its value by the time your code started.

In general, a **let** binding of a variable establishes a bulkhead: modifications made on either side of the binding cannot be seen on the other side.

Common Lisp says that *terminal-io* *is the stream connected to the console and that* *query-io* *is the stream used for asking the user questions. But sometimes, when my program outputs to* *terminal-io* *nothing appears on the screen and when it reads from* *query-io* *it doesn't see keyboard input. Why?*

In a simple uniprocessing Common Lisp implementation with no window system, ***terminal-io*** and ***query-io*** would probably work as you were expecting. However, on a machine with a window system, ***terminal-io*** outputs to a window which may or may not be visible on "the console" mentioned in the Common Lisp standard. Similarly, on a machine with multiple processes (or multiple stack groups), the keyboard is connected to ***query-io*** of a process which may or may not be your process waiting for a reply.

Until the Common Lisp standard expands to include window systems and multiple processes, then you must do two things:

* learn your system's user interface conventions for choosing which windows are visible on the console and for selecting which process receives keyboard input (e.g., clicking on a window with a mouse); and

* learn your system's programming conventions for coordinating the standard streams such as ***terminal-io*** and ***query-io*** as the user switches among windows and processes.

Although Common Lisp does not yet officially include a window system, it provided the seemingly redundant ***terminal-io*** stream to accommodate existing window systems. The other standard streams (***standard-output***, ***standard-input***, ***error-output***, ***trace-output***, ***query-io***, and ***debug-io***) may be directed anywhere you want them; but they are normally all directed to the same window.

Now we have a management problem of how to redirect all those individual streams as their window alternates between being buried and being visible. Perhaps the system could handle these six standard streams as special cases, but how could the window system manage other streams created by the program?

The solution is to use ***terminal-io*** as a common name for all streams (standard or otherwise) which are supposed to be directed to "the console". Now, when a window is buried or exposed, the window system informs the process of its new status by switching that process's copy of ***terminal-io*** appropriately. For this convention to work properly, it is important that all stream sharing the console be bound to *synonym streams* of ***terminal-io***. This convention will *not* work if the streams are merely bound to ***terminal-io*** itself. For example,

```
(let ((*standard-output* (make-synonym-stream '*terminal-io*))
      (*error-output*    *terminal-io*))
  ...
```

Inside this **let**, output to either ***standard-output*** or ***error-output*** will go to wherever ***terminal-io*** is directed.

Now consider what happens if ***terminal-io*** is set to a new window somewhere inside that **let**. Since ***standard-output*** is defined as a synonym stream of ***terminal-io***, output to ***standard-output*** is now directed to the new window also. No matter how ***terminal-io*** might be bound or set in the future, output to ***standard-output*** will always go to the same place as output to ***terminal-io***.

The ***error-output*** stream, however, is a different story. It was not bound to a synonym stream of ***terminal-io***. Instead, it was bound to whatever the value of ***terminal-io*** was at the time the **let** was entered. Later changes to the value of ***terminal-io*** will have no effect on ***error-output***. Whereas ***standard-output*** and ***error-output*** originally output to the same place, they may later output to different places if ***terminal-io*** changes.

The simplest way to make all these conventions and the default stream conventions mentioned in the Common Lisp standard work is to establish the following set of bindings upon entry to your process:

```
(let* ((*standard-output* (make-synonym-stream '*terminal-io*))
       (*standard-input*  *standard-output*)
       (*error-output*    *standard-output*)
       (*trace-output*    *standard-output*)
       (*query-io*        *standard-output*)
       (*debug-io*        *standard-output*))
   ...
```

Without these bindings, your process will probably run with whatever defaults happened to exist in the system when your process was created. If your process has its own window, then you should insert a binding of ***terminal-io*** to that window stream at the top of that **let*** so that all these standard streams default to that window.

Interestingly enough, you should still establish the bindings shown above even if you have a background process which has no window and is not supposed to output to the console. Why? If your process encounters an error, then system code may unexpectedly output to ***error-output*** or ***debug-io***. If your system is capable of handling unexpected output from a background process, then it once again is likely to assume that it can control all console streams by controlling output to ***terminal-io***. If you have bound the standard streams to a synonym stream of ***terminal-io***, then the unexpected error output from your background process should be handled according to your system's standard conventions. Without such bindings, non-standard things might happen, such as the dreaded "window lock".
