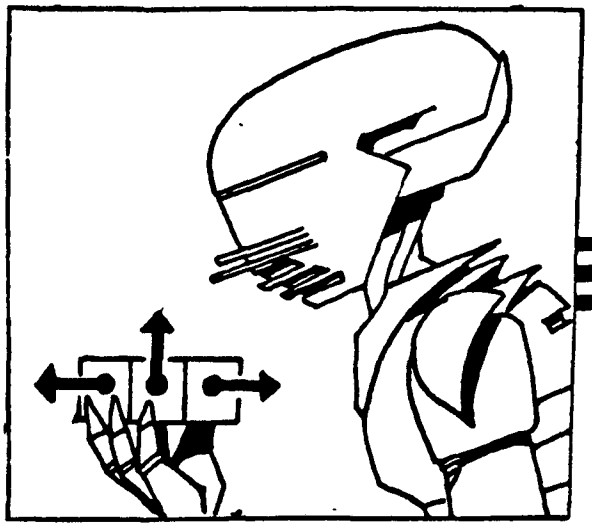

Query-io:

- Fill pointers confuse me, when are they used, when are they ignored?

A vector (One dimensional array) can be considered under two different perspectives:
From a low level perspective, it is a block of storage, whose dimension is set by `make-array` or `adjust-array`. All the elements of the array are accessible.
From a more abstract perspective, a vector represents a sequence of elements. When it is desirable for the sequence to have a varying length, the fill-pointer comes into play. It allows for sequences with flexible length to be represented as arrays, whose storage size is fixed. The fill-pointer determines the length of the sequence.

Common Lisp reflects these two views: Primitives meant to work on arrays ignore the fill-pointer, considering all the elements of the array, and primitives that are sequence oriented, will consider the fill-pointer to be the end of the sequence.

`Aref` and `(setf aref)` will access elements regardless of the fill-pointer, while `elt` will not.

`make-array` and `adjust-array`, will initialise or fill the array beyond its fill-pointer, while all the sequence functions like `fill`, `replace`, `find` will always stop at the fill-pointer.

String operations are considered to operate on specialized sequences, they all look at the fill-pointer.

`Equal` will consider fill-pointers for string and bit-vectors, and `equalp` will always consider fill-pointers.

Printing, when `*print-array*` is true, considers vectors as sequences, and will not print elements beyond the fill pointer.

Why do `read-char`, `read-line` *and* `peek-char` *take* `recursive-p` *as an argument?*

For `read`, `read-preserving-whitespace`, and `read-delimited-list`, the `recursive-p` argument being false indicates that a new top-level read context is established. It allows for correct reading of #n= and #n# syntax. It also allows for deciding if the whitespaces at the end of the expression read are to be preserved. None of that applies to `read-char`, `read-line` and `peek-char`. However, some systems are using the argument to report better the end of file error. If the end of file is encountered while the `recursive-p` argument is true, it means that it occurred in the middle of reading an object, and the error message can be more specific. Some systems are also using the `recursive-p` argument to set up or initialize some resource they need for interactive input, such as input editors. Calling `read-line` with `recursive-p` being true when the input editor is not setup might result in the absence of echo or incorrect key bindings.

Dear Patrick,
I was trying to define a structure like the following one:
`(defstruct pk1::foo pk1::a pk1::b)`
This didn't work until I switched my current package to from `pk2` *to* `pk1`, *why is that?*

You probably guessed, this is a package problem. The function that you would normally expect being automatically defined are foo-a foo-b foo-p and make-foo. They are being defined, but not in the package that you were expected. Common Lisp the Language says on page 308 that the name of automatically created functions are interned in the package current at the time of processing of defstruct. This time is different from the time of reading. The name foo is interned into pk1, because the name is interned at read time. make-foo is interned after the whole defstruct form has been read. It happens when the evaluator or the compiler expand the form. If pk2 is current at this time, make-foo will be interned in pk2. It is confusing enough that the CLOS design committee decided to do away with automatically generated names.