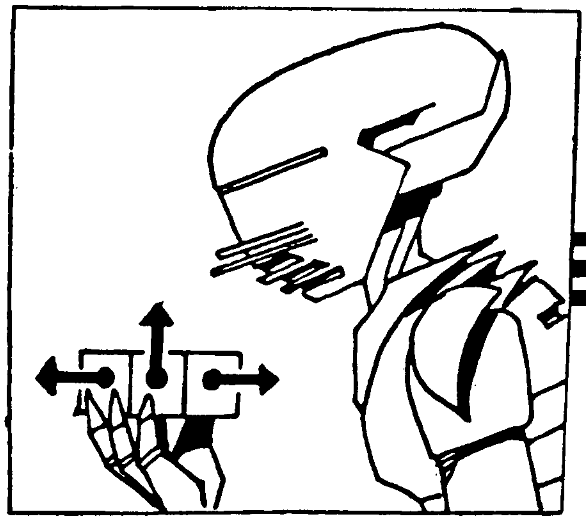

Query-io:

- Can you talk about key processing techniques?
A general problem in key processing involves the passing of keys from one function to another, with some defaulting or overriding of key values.
In the following expample, we want to create a vector with a fill pointer of 0. We are expected to support all the keys that make-array supports.

```
(defun make-empty-vector-with-fill-pointer (length &rest key-value-pairs)
   (apply #'make-array length :fill-pointer 0 key-value-pairs))
```

Since Common Lisp specifies that in case of key duplicates, the leftmost argument pair takes precedence, make-array will always get a value for the :fill-pointer key of 0.

Now, if we want to specify a default for :fill-pointer but we don't want to shadow the one given by the caller, we can do it this way.

```
(defun make-vector-with-fill-pointer (length &rest key-value-pairs
                                             &key (fill-pointer 0)
                                             &allow-other-keys)
    (apply #'make-array length :fill-pointer fill-pointer key-value-pairs))
```

fill-pointer will be bound to the value of the :fill-pointer key given by the caller, and will default to 0. Then the value, possibly defaulted, is explicitly passed to make-array. Notice the use of &allow-other-keys in the argument list, so that other keys, besides :fill-pointer, can be passed as well.

It is worth noting that the keyword parsing for make-array will have to be done at run time and therefore can be more costly than a call with required arguments only.

- In what situation should someone use an inline function instead of a macro?

Inline functions are designed to improve the performance of the calling function by integrating the code of the inline function into the calling function. This avoids the function call overhead and allows the compiler to make further global optimizations. The compiler guarantees that the resulting code is semantically equivalent to the one where the function is called.
On the other hand, a macro call expression is substituted with the expansion of the macro. It is intended to be used to create new syntactic constructs. Since it is an expression substitution, its behavior might depend upon where it is expanded.

Let's consider this definition of prog1:

```
(defmacro prog1 (form1 &body forms)
  `(progn
    ,form1
    ,@forms
    ,form1))
```

The form

```
(prog1 ,a
     (incf a))
```

will expand into

```
(progn
  a
  (incf a)
     a)
```

which looks correct. But the form

```
  (prog1 (incf a)
    (incf a n))
```

expands into

```
  (progn
    (incf a)
       (incf a n)
       (incf a))
```

This is wrong because the form (incf a) is evaluated twice. A macro writer must take care of macro argument evaluation problems by himself.

Here is a better definition.

```
(defmacro prog1 (form1 &rest forms)
  `(let ((exp ,form1)
    ,@forms
    exp)))
```

If we use it in this function definition,

```
(defun update  (value)
  (prog1 (get-value )
    (setf (get-value value))))
```

it is equivalent to

```
(defun update (value)
   (let ((exp (get-value)))
     (setf (get-value value))
        exp))
```

This does work.
But if we use it in this function:

```
(defun update-expression (exp value)
      (prog1 (compute exp)
         (setf (compute exp) value)))
```

it is equivalent to:

```
(defun update-expression (exp value)
      (let ((exp (compute exp)))
        (setf (compute exp) value)
      exp))
```

This is not what we want. The argument exp **has been shadowed by the** exp **used internally by the macro expansion. This is called "identifier capture". To avoid this, the macroexpansion must use a symbol that nobody else is using.** Gensym **is made for that.**
A correct definition of prog1 **is slightly more complicated:**

```
(defmacro prog1 (form1 &rest forms)
  (let ((s (gensym)))
    `(let ((,s ,form1)
       ,@forms
       ,s))))
```

In conclusion, since macros are more difficult to write than straight functions, when all you want is to get open coding instead of a function call, write it as a function proclaimed inline. Note that most implementations require that the proclaim form must appear before the definition of the function.
Here is an example of an inline function:

```
 (proclaim '(inline interlisp-assoc))
 (defun interlisp-assoc (item list)
  (and (consp list)
       (assoc item list :test #'equal)))
```