The built-in operations of most languages are simply-recursive and thus allow only simply recursive operations to be written using built-in functions alone. Extra semantic power can be gained by making available a general-recursive primitive operation. It is especially appropriate to allow a binary operator op which accepts general list structures as its first parameter, in such a way as to make any general recursive function $f(x)$ realisable as

$$\text{represents}_f \text{ op } x.$$  

Such an op will in effect be a general-purpose interpreter; its left-hand arguments representing programs in an internally manipulable form. A simple variant of this scheme is proposed in John Backus' two papers on 'reduction languages'; cf. Backus [1] and [2]. Note that a language with this one feature need in principle have no other features supporting control, recursion, or assignment; an observation which is however of more theoretical than practical interest. Such a primitive will of course make a variant of dynamic procedure formation available. The following SETL extension is freely adapted from Backus' proposal. We generalise the definition of the 'curly bracket' operation $f(x_1, \ldots, x_n)$; this generalised 'application' becomes the op spoken of above. The following definition, which assumes a system mapping $\text{defof}$ defined on blank atoms, is used.
\[
f[x_1, \ldots, x_n] = \langle x_1, \ldots, x_k \rangle \quad \text{if } f \text{ is the null vector}
\]

\[
= f(l) \{x_1, \ldots, x_n, f\} \quad \text{if } f \text{ is a vector;}
\]

\[
= f(x_1, \ldots, x_{k-1}, \langle x_k, \ldots, x_n \rangle) \quad \text{if } f \text{ is a function of } k \text{ variables with } k < n
\]

\[
= f(x_1, \ldots, x_n) \quad \text{if } f \text{ is a function of a variable with } k = n
\]

\[
= f(x_1, \ldots, x_n, \Omega, \ldots, \Omega) \quad \text{if } f \text{ is a function of } k \text{ variables with } k > n \text{ and } x_n \text{ not a tuple}
\]

\[
= f\{x_1, \ldots, x_{n-1}, x_n(1), \ldots, x_n(\# x_n)\}
\]

in all other cases if \( f \) is a function

\[
= f\{x_1, \ldots, x_n\} \quad \text{if } f \text{ is a set}
\]

\[
= \text{defof}(f) \{x_1, \ldots, x_n\} \quad \text{if } f \text{ is blank } \# \Omega
\]

\[
= \text{error in all other cases}
\]

It is clear that the generalised 'curly bracket application' described above can readily be programmed if the \( \# \) operation can recover the number of parameters of a function and if an apply function is available for attaching argument list to objects. For this reason, we expand the list of SETL primitives very slightly,
letting \#f denote the number of arguments of \( f \) when \( f \)
is a function, and introducing an operator \texttt{apply} such that

(1) \[
\text{apply} \ <x_1,\ldots,x_n> = x_1 \ <x_2,\ldots,x_n>
\]

Some examples: To construct an element \( f_1 \) such that

\[
<f_1,f_2,\ldots,f_n> \ <x_1,\ldots,x_k>
= f \ <<f_2,\ldots,f_n> \ <x_1,\ldots,x_k>>
\]

put \( f_1 = <d,f> \), where \( d \) is as below. Since

\[
<f_1,\ldots,f_n> \ <x_1,\ldots,x_k>
= <d,f> \ <x_1,\ldots,x_k, <f_1,\ldots,f_n>>
= d \ <x_1,\ldots,x_k, <f_1,\ldots,f_n>, <d,>>
\]

we may define \( d \) by

\[
\text{define} \ d(u);
\text{return} \ u(#u)(2) \ {\text{apply}(u(#u-1)(2:)) + u(1:#u-2))};
\text{end} \ d;
\]

and then we have

\[
<f_1,\ldots,f_n> \ <x_1,\ldots,x_k> = f \ <<f_2,\ldots,f_n> \ <x_1,\ldots,x_k>>.
\]

This gives us a very easy 'composition' function

\[
\text{define} \ \text{comp}(f_1,f_2); \ \text{return} \ <<d,f_1>,<d,f_2>>; \ \text{end} \ \text{comp};
\]

To attach \( x \) as \( i \)-th parameter of an \( n \)-parameter function,
getting an \( n-1 \) parameter function, use \( <\text{at},f,i,x> \) with

\[
\text{define} \ \text{at}(u);
\text{return} \ u(#u)(2) \ {u(1:u(#u)(3)-1) + u(#u)(4)) + u(u(#u)(3):)}; \ \text{end} \ \text{at};
\]

To attach \( g(x_1,\ldots,x_k) \) as \( i \)-th parameter of an \( n \) parameter
function, getting an \( n+k-1 \) parameter function \( fg \) defined
by \( fg \ <x_1,\ldots,x_{n+k-1}> = f \ <x_1,\ldots,x_{2-1},g(x_1,\ldots,x_{1+k-1}),x_{1+k},\ldots,> \)

we can proceed similarly using \( <\text{subst},f,g,i,n> \); where \( n \) may be

\( \Omega \) if \( f \) or \( g \) is a function. The program for \texttt{subst} is fairly obvious.

In situations where generalised application is to be used
heavily, a syntax allowing applications with two arguments
two be written in infix position is of course desirable.