Some small and large language extensions for consideration.

This note will serve to record some extensions to SETL that might be useful. The first is merely an observation; the second would fit quite easily into the present framework; the others would require a more substantial change. Comments are solicited.

1. **Use of square brackets within expressions.**

   The notations \( f[a], f[a,b], f(a,[b]) \), etc. are provided, where for example \( f[a,b] \) is by definition the set
   \[
   \{ f(x,y), x \in a, y \in b \}
   \]
   and
   \[
   f(a,[b]) \text{ is the set } \{ f(a,y), y \in b \}.
   \]
   The same notations can be used with infix operators, so that for example
   \[
   [a] + 1
   \]
   for a set of integers is \( \{ x+1, x \in a \} \),
   while \([a] + [b] \) is
   \[
   \{ x+y, x \in a, y \in b \}.
   \]
   Note that these constructions can be compounded.

2. **Calculations within expressions.**

   In-line subroutines, somewhat like the direct application of PROC in BALM, might be useful, and could be provided in some such form as the following.

   Let **block** be a block of statements, containing certain statements of the form
return expn;

Then

(1) calc block;

or,

calc block end;

or

calc block end calc;

could be allowed as an expression. Variables within such an expression would have the name scopes determined by the surrounding context; the code in block would be executed up to the first statement of the form

(2) return expn;

encountered. The value of (1) would then be the value of the expn in (2); or, if the end of block were reached before any statement (2) were encountered, the value of (1) would be \_\_\_. Thus, for example, to use the sum of \(x, f(x), f(f(x)), \text{etc.},\) within an expression, this sequence being extended to the first zero value encountered, we could write

\[
a = \text{calc } s=0; y=x; (\text{while } y \neq 0 \text{ doing } y=f(y);) \ s=s+y;;
\]

return s;; + ...

3. Name-atoms and a type of pointer construction suggested by ALGOL 68.

Suppose that

a. Names are permitted as an atom type;

b. Two special blank atoms elementof and setof are introduced (for a purpose to be explained shortly);

c. The definition of the basic SETL operations are modified as follows.

c1. The value of the application operation

\[f(a_1, ..., n_n)\]
whose $f$ is a set name, is the $n+2$ tuple

(3) $\langle \text{elementof}, a_1, \ldots, a_n, \widetilde{f} \rangle$,

where $f$ is the name atom corresponding to $f$. (Like the BALM '=?'.

\begin{enumerate}
\item Similarly, the value of the application operation

(4) $f\{a_1, \ldots, a_n\}$,

where $f$ is a set name, is the $n+2$ tuple

$\langle \text{setof}, a_1, \ldots, a_n, \widetilde{f} \rangle$.

\item When an $n+2$ tuple of this form is used as the operand

of any built-in SETL operation (excepting, however, left-hand

operands of equality (i.e., '='), which we now regard as an

operation), it is 'evaluated', i.e., either

$$f(a_1, \ldots, a_n)$$

(or

$$f\{a_1, \ldots, a_n\}$$)

taken in the existing SETL sense, and this value used in place

of (3) (or (4)).

\item The same rule applies to the right-hand operand of

an equality sign. But when (3) is the left-hand operand of an

equality sign, we evaluate

$$\langle \text{elementof}, a_1, \ldots, a_n, \widetilde{f} \rangle = \text{expn}$$

performing the assignment

$$f(a_1, \ldots, a_n) = \text{expn},$$

in its present sense. Similarly, when

$$\langle \text{setof}, a_1, \ldots, a_n, \widetilde{f} \rangle = \text{expn}$$

is evaluated we perform

$$f\{a_1, \ldots, a_n\} = \text{expn};$$

in its present sense.
\end{enumerate}
These conventions allow such 'pointer-tuples' to be passed to subroutines and to be calculated by expressions and functions; giving powerful new possibilities (though complicating optimization). Thus, for example, we may write

```plaintext
if x>0 then f(x) else g(x,y) = expn;
```
as in ALGOL 68. The subroutine `in`, defined as always by

```plaintext
define a in b; b = b with a; return b; end in;
```
can then be invoked in the form

```plaintext
a in f(x);
```
Moreover, if we define a function

```plaintext
definef thing; external y; return if x>0 then f(x)
else g(x,y); end thing;
```
then we can write

```plaintext
thing = expn
```
with the expected result.