SETL Newsletter #34 May 16, 1971

Syntax revisions in preparation for implementation J. T. Schwartz

This note will suggest a number of syntactic improvements to SETL, hopefully in accordance with the general principles stated in newsletter 26. Some of the suggestions will involve a different reaction to the general issue of 'object types' than is involved in the earlier SETL specifications. Cf. also newsletter #31.

Comments on these suggestions are urgently solicited.

I. Atoms of type subroutine, function, label, blank, and boolean

No changes in the operations provided. Note however that the name-scoping rules for subroutines and functions may be revised in a later newsletter; especially in regard to the usage of the 'external' declaration. Some usages in dynamic compilation might also be changed.

II. Character strings

a. string + string, integer * string remain as is.
b. j elt string becomes string(j) .
c. len string becomes #string.
d. string(i,j) is that portion of a string between the i-th and j-th positions, or Ω if i or j is out of range; this is a retrieval operator, and may be used on the left.
e. dec, oct, nul, nulc remain as is.
f. string * integer becomes a hash of the string to give a bit-string of a length determined by the integer.
g. hol string regards a character string as a bit string in some dense internal format; holl is the number of bits needed to represent one character. Thus the length of hol 'abc' is 3*holl.
   hol bitstring pads a bitstring with zeroes to the nearest even multiple of holl, and then performs the reverse conversion.

III. Bit strings

Items II.a,b,c,d,f,g apply mutis mutandis.

n bin integer converts a positive integer to a bit string n bits long, padding with leading zeroes. bin bitstring provides the reverse conversion.Boolean operations for bit-strings stay as is, but with different notations. or and not remain the same; and may optionally be written as *; the exclusive or operation is written as a//b; and the operation a and n b written as a-b. The operation b or n a (implication) will be written a/b. We will write a ge b for bit-strings if a has a 1 wherever b has a 1; the operations a le b, etc. will be used similarly.

IV. Expanded object-type function

The tokens set, integer, boolean, bstring, cstring, label, blank, real, subroutine, function, pair represent particular integers mnemonically. They are used in connection with the function type to determine the type of an object. Thus the old usage

if pair x then... is replaced by if type x eq pair then ... . The particular object pair may also be designated as tupl. This prepars for the introduction of programmer-definable object types.

V. Atoms of type 'real'.

Real arithmetic will be provided, in a manner depending as usual on machine and implementation. The arithmetic operations +, -, *, /, and exp are provided for real numbers. The operation real log real, where the first constant is the logarithmic base is also provided, as are cos(x), sin(x), x min y, x max y, and abs y.

Real constants are written in the form n.m, n and m being integers, both required.

itop x is the least integer exceeding x;
ibot x is the greatest integer not exceeding y.

itop n is the real number most closely approximating the integer n from above, given the limited precision of real numbers.

bitr real converts a real number to a bit-string of appropriate length, this length being designated as bitrl.

bitr bitstring converts a bit-string of appropriate length to a real number.

## VI. Integer exponentiation

The exponential function m exp n is provided for integer m and positive integer n.

## VII. Tuples.

The treatment of tuples will be changed in a number of substantial ways. Instead of being regarded as sets, tuples will be regarded as a distinct data type (though a conversion function allowing them to be regarded as sets when this is advantageous will be available). In particular, we distinguish between a tuple consisting of one single component c and its single component. We can create this tuple by using the operator just, as in

x = just c;

The reverse operation is given by the operator ∋ , as in

c = ∋ x .

However <just a,b> will be the same as <a,b>, etc. Operations syntactically resembling those available for sequences (or character strings) will then be provided. Those will be as follows:

a. tupl(n) replaces the present <* z n>tupl
b. tupl(-n) replaces the present <*zt n>tupl .

Both of these are retrieval operators, and can be used on the left of an assignment statement to give a replacement effect.

c. <a,b> can be written as just a + just b. (See below, for use of '+' as tuple concatenator). Thus, for example, a sequence may be converted to a tuple by writing

tupl = [+: 1 ≤ n ≤ #seq] just seq(n);

d. tupl(m,n) is [+: m ≤ j ≤ n] just tupl(j);

This is a retrieval operator, and may be used on the left of an assignment statement.

e. We write #tupl for the quantity defined by

j = 0; (while tupl(j+1) ne Ω) j=j+1;; returnj;

f. concatenation of tuples is written as tupl + tup, and defined by the sequence

cat=tup; (#tupl ≥ ∀n ≥ 1) cat = <tupl(n),cat>;; return cat;

g. Tuples as iteration controllers. The iteration header

(1 ≤ ∀n ≤ #tupl)

can be written as

(∀n ∈ tupl).

The set-former and quantifier notations that derive from this are allowed also. Thus, for example, a tuple can be converted to a sequence by writing

seq = {<n,tupl(n)>, n ∈ tupl} .

h. Membership in tuples. We write x ∈ tupl for ∃n ∈ tupl | tupl(n) eq x .

i. Range of mapping f[tupl] will be the tuple [+: n ∈ tupl](just f(tupl(n)));

j. Conversion of tuples to sets, and vice-versa

This will be provided through the general 'regard as' mechanism, to be discussed in a subsequent newsletter on programmer-definable object types.

## VIII. Revised conventions for is

The internal assignment operator is will assign a value to its second rather than to its first argument. Thus we will write

```
if a+b is c gt d then ...
```

rather than

```
if (c is (a+b)) gt d then ...  .
```

The operator is will have low left precedence (equal to the precedence of '=') but high right precedence (equal to that of '('). Thus, for example, the code

```
if  a+b is c * d  is e gt then ...
```

is equivalent to

```
c = a+b; e=c*d; if e gt 0 then ... .
```

## IX. Built-in union, intersection, and other set-theoretic operators

The operators union, intersection, difference , and symmetric difference will be written as set + set, set*set, set-set, set // set. Set inclusion will be written as set ge set; the operators le, gt, lt will be used similarly.

## X. Random functions.

If n is a positive integer, then random n will be an integer chosen at random from the range $1 \leq j \leq n$. If x is a real number, then random x will be a real number chosen at random, and with a uniform distribution, from the interval [0,x] (if x is positive, otherwise [-x,0]). If s is a set, then random s will be an element of s chosen at random. Random elements may be chosen from tuples by writing tupl(random #tupl); and similar remarks apply to strings, etc.

XI. Nonmembership. The negative of the 'ϵ' relationship can be written as nϵ.

XII. Compound operators for while-iterations. The compound operator form

[op: ...] expr

is extended to apply to 'while' iterations. The syntactic conventions are as follows:

[op: while C doing block] expr

denotes the value which would result from the following iterative calculation:

```
v = Ω; times = 0; (while C doing block)
if times eq 0 then times=1; v = if times eq 0 then
        expr else v op expr; if v eq Ω then quit; end while;
```

The 'while-when' statement described below leads to a corresponding compound operator, of the form

[op: while C when CC doing block] expn;

XIII. While-when iterations. The statement form

```
(while C when CC doing block1) block2;
```

is introduced. This is equivalent to

```
(while C doing block1) if n CC then continue; block2;
```

The simpler form

```
(while C when CC) block;
```

is also allowed, being equivalent to

```
(while C) if n CC then continue; block; .
```

XIV. Then-if forms.

The statement

```
if cond1 then block1 else if cond2 then block2... ;
```

may be written

```
then block1 if cond1 else if cond2 then block2...;
```

or with a more elaborate transformation as

```
then block1 if cond1 but block2 if cond2...;
```

The simplest forms of this type of statement would be

```
then block1 if cond1;
```

and

```
then block1 if cond1 else block2; .
```

XV. At-blocks.

An at-block has the form

```
(at label) block;
```

and may also be terminated in any of the forms

```
(at label) block end;
(at label) block end at;
(at label) block end at label; .
```

Here, label must be a valid SETL name which occurs as the name of a label in the routine which contains the at-block. This label must be 'enabled' as the 'target' of an at-block by being enclosed in at least two sets of square brackets. If a given label is addressed by several at-statements, and thus is the target for placement of several blocks, these blocks may be inserted at the label by the SETL compiler in any order.

XVI. 'Local' subroutines.

If the declaration

```
subname local;
```

occurs within a subroutine or function S, where subname is some name known within S as a subroutine or function name, then all the variable names used within S are declared to have the significance which attaches to names identical in form occurring within subname. If this declaration occurs in the expanded form

```
subname local name1,name2,...,namek;
```

then all names used within S other than $name_1,\ldots,name_k$ are identified with names occurring in subname in the sense stated above. The names $name_1,\ldots,name_k$ however are local to S.

However, the designational effects of any external declaration appearing in S will override the effects of such a local declaration.

If S is directly imbedded within another subroutine, whose name is subname, then the declarations

local; or local $name_1$,...,$name_k$;

appearing in S has precisely the same force as

subname local; and subname local $name_1$,...,$name_k$;

respectively.

XVII. 'Inverted' form for subroutines and functions.

A subroutine or function may have a form like

[; subroutine body; define subname(a,b,...);] .

This form is precisely equivalent to

define subname(a,b,...); subroutine body; end subname;

The same remark applies to functions, subroutines, and also functions written in infix or prefix form, etc.

The slightly variant form

[; subroutine body; define subname(a,b,...);-]

is equivalent to

[; subroutine body; define subname(a,b,...);] subname(a,b,...);

i.e., to a subroutine definition followed immediately by an invocation of the subroutine with parameters identical to those which appear in its defining text. The same convention applies to functions, and may be used within expressions, thus making it possible to interpolate arbitrary code blocks into expressions. Suppose, for example, that in an expression we need to add the next-to-largest element of a sequence to some other quantity. This might be written

```
x = x + [; local mx; mx=[max: 1 ≤ n ≤ #seq]seq(n);
        return [max: 1 ≤ n ≤ #seq | seq(n) ne mx] seq(n);
        definef submax(seq);-].
```

The still more degenerate form

```
[; function body;]
```

is equivalent to

```
[; function body; definef fname] fname ,
```

where fname is some name generated by the SETL compiler. This 'unnamed function' form is intended for single use within expressions, and to make it possible to interpolate arbitrary code blocks into expressions. In this form, the declaration 'local' will be understood if no explicit local or external definition appears. The preceding example can be written using this option as

```
x = x+[; local mx; mx=[max: 1 ≤ n ≤ #seq]seq(n);
       return[max: 1 ≤ n ≤ #seq | seq(n) ne mx]seq(n);]
```

## XVIII. Modified macro-conventions

Macros will be provided in at least two forms. The simpler form of macro will be described immediately below. A more powerful but somewhat less easily used 'syntax macro' feature, adapted from the BALM 'means' mechanism, will be provided also, and will be described in a subsequent newsletter. This syntax macro feature may in turn be supplemented or replaced by a still more powerful language extendability feature, if improved insight into the design of such a feature is attained.

Here we shall describe only the 'simple' macro feature, which resembles the present SETL use of 'do'-blocks, but with certain improvements.

a. The keyword 'do' is abolished. macro-calls will be indicated merely by the occurrence within text of a macro name.

b. A macro may be defined in the form

block macname(arg$_1$,...,arg$_n$); body; end macname;

specified in the SETL notes. It may also be defined in the inverted form

[; body; block macname(a,b,...);] .

The variant form

[; body; block macname(a,b,...);-]

is equivalent to

[; body; block macname(a,b,...);]macname(a,b,...); .

XIX. Sinister calls. The sinister call mechanism discussed in newsletter 30 is adopted, with the syntactic conventions explained there. This replaces the rather elaborate 'extraction' and 'replacement' operator conventions described in section b, pages 80-91 of the SETL notes, which are dropped. The simpler assignment operator conventions described on pages 44-45 of the notes are retained, however.

XX. Iff-statements. An 'iff' statement, generalizing that sketched in newsletter 26, section B, will be provided. Detailed syntactic and semantic conventions remain to be worked out, and will be described in a subsequent newsletter.

XXI. Programmer-definable object types. These will be provided, syntactic and semantic conventions to be described in a subsequent newsletter.

XXII. Additional features contemplated for the first or for subsequent implementations.

A number of plausible extensions might be considered for future implementations.

A. Default parameters. (Probably not to be included in first implementation). General purpose systems routines often include a large number of parameters, many of which set rarely used option-flags or values. In this situation, it is valuable to allow calls to a subroutine or function to have fewer parameters than the number of parameters specified in the subroutine, and to take the missing parameters to be transmitted with the value Ω. The subroutine itself may then test for this parameter value and substitute a standard default value for it.

B. Mechanism Linkages. (Modification of external subroutine linkages). A subroutine S is linked to fixed external objects in two principal ways:

i. The subroutineswhich S calls are named explicitly within the body of S; the number of parameters which each of these subroutines expects also is indicated explicitly.

ii. A fixed distinction between internal and external objects is established within the subroutine; and explicit global names are given for the external objects.

These fixed conventions make it quite clumsy to establish certain types of inter-program linkages. For example, it may be appropriate in certain situations to interpolate some type of intermediate 'transformation' between S and a subroutine it calls, securing in this way some special effect. Or one may wish to connect a particular program (e.g. an edit program) to one of several sources of input, and to one of several sinks for output or intermediate memory management routines. In the present version of SETL, this can only be done either by transmitting to a given S as explicit functional parameters the identity of many of the subroutines which S will use, or by artifically exploiting the name-scoping rules, that is,

hiding a subroutine T within another subroutine which assumes its name and links in its place to something which calls T. This last procedure can be quite clumsy. A scheme that makes it possible to over-ride the normal subroutine-linkage mechanisms and to vary the pattern of inter-subroutine linkages flexibly could be useful. It can also be useful to centralize these global links in a single 'master switch' capable of supplying parameters and providing for special action interpolations as necessary. Ideas of this kind have been suggested by R. Krutar, and deserve additional consideration They may help point to ways in which SETL can be extended in the direction of "languages of mechanism". For a published discussion of some of these issues, cf. R. M. Balzer, "PORTS - A method for dynamic interprogram communication and job control," AFIPS Conf. Proc., v. 38, pp. 485-489, together with the references cited there.

C. Debugging features. When designed, these might have an impact on the language.

D. Extendability. This might also impact the language significantly.