SETL Newsletter Number 21 April 15, 1971

AN OUTSIDE REVIEW

COMMENTS ON THE SETL DRAFT

The long preface is quite compelling, but its mandate is not fulfilled by the language described in the rest of the manuscript. As a contribution to the technical literature, I therefore recommend against publication of the manuscript. This is not to say that SETL is devoid of merit; on the contrary, there are many good ideas in it. SETL has a great power of expression, but it is still "just another programming language", with perhaps more rough edges than some.

In the comments which follow, I will discuss the SETL language, not the prose of the manuscript, since it is a well-written draft that I expect to go through some re-drafting before any possible publication.

Compared to the goals set forth in the preface, particularly

> As a final benefit, we expect the availability of the mathematicized programming language which will be described in the present work to broaden the frontier of contact between programming and mathematics. It should at any rate serve to emphasize to the mathematician that programming need not be a mass of petty detail only, that in fact it is concerned in a way only slightly unfamiliar with some of the issues which he is accustomed to confront...[page xi],

SETL has several major flaws which are discussed in detail in the paragraphs to follow:

1) It cannot be read by a mathematician.
2) There are no set operations analogous to APL array operations--everything must be done element-by-element.
3) The identity elements of mathematics are not preserved in all cases.
4) The treatment of N-tuples is anomalous.
5) There is an overall lack of conceptual consistency.

1) It cannot be read by mathematicians.

a) Character set. SETL appears to have a strong FORTRAN bias, with the use of eq, ne, and, etc. If this is to be a mathematician's language, then it should have mathematical symbols and use them in the expected manner (e.g. $=$, $\neq$, $\wedge$, $\vee$, $\supset$).

b) Operator precedence. While it is a desirable goal to make expressions as readable as possible, the devices adopted in SETL to avoid parentheses are no improvement. A mathematician will give up the precedence of multiplication over addition only with good cause, which is not forthcoming. The dollar sign notation is at best confusing: it is difficult to remember whether $+ raises or lowers the precedence. I would prefer Polish notation over the dollar signs. The same "non-intuitive device" argument applies to the notations 10b775 and [[label:]].

c) The multiple usage of a given symbol for different functions is confusing. A flagrant example is:

$< X - Y > = < A-B, C, D, E >$

$< X - Y > < A-B, C, D, E >$

$< X, Y > = < A-B, <D, E > >$

all of which apparently mean $\begin{matrix} X \leftarrow A-B \\ Y \leftarrow D, E \end{matrix}$

The pointed brackets are used to denote the different concepts of N-tuple, simultaneous assignment, and selection. The minus signs are used to denote the different concepts of integer subtraction and "zero" -- the place holder in selection.

2) As a set language, SETL has a strange lack of operations on sets. All operations on a set must be done explicitly element-by-element. In SETL, the union of two sets is not a primitive; instead the mathematician must construct an element-by-element definition of union, then use that as a subroutine (what is a subroutine?). With no set union, set intersection, set difference, general simultaneous operation on all the elements of a set, or general operation among the elements of a set, SETL algorithms must include a "mass of petty detail" specifying explicit iterations. Among other drawbacks, this forces the computer programmer's concept of strictly sequential operation onto a mathematical concept which is unrelated to time.

Compound operations are poorly conceived. Instead of explicitly specifying iteration (and for integer subscripts, order of iteration) over the elements of a set, it would be better for a compound operator to be conceived as implicitly applying to all the members of a set:

if S is the set {1,3,2} , then

+ : S is 6

* : {X+1, XεS} is 24

With the concept of simultaneous operation on all the elements of a set, the second example becomes

* : (S+1).

3) The identity elements of mathematics.

To make algorithms work for degenerate cases, it is crucial to define that operations on the null set produce the identity for that operation:

+ : nl is 0

* : nl is 1

(old notation: [+ : X ε nl] x is 0, not Ω)

This does not preclude defining operations on Ω to give Ω or an error indication.

Is there an identity element I for "tuple building" such that $< X,I > = X$ and $< I, X > = X$? If so, example 2a on page 103 could avoid making a special case of the first element of the sequence. As the example stands, MAKETUP(nl) returns Ω instead of nl.

4) The treatment of N-tuples

SETL appears to be much more oriented around N-tuples than sets, yet the N-tuples described have serious drawbacks. The first drawback is the extremely specialized role of the first component of an N-tuple. This first component is to be used as an associative search key in function applications (denoted by SET(key), SET{key}, or SET[key], depending upon considerations that are not at all intuitive to a mathematician). The language therefore makes it difficult to perform an associative search using any other component of an N-tuple as a key. As an example, consider a process that represents the arcs in a flow graph as a set of ordered pairs:

A = { < fromnode, tonode > } .

In SETL, the notation A {node} can be used to specify the set of successors of a node. There is no equivalent way to specify the predecessors of a node. The subscripting and searching operations of APL do not have this asymmetry.

Another drawback of the N-tuple as a basic data structure is the "mass of petty detail" involved in using small integers as the names of the various components. For example, if one constructs a compiler symbol table as a set of N-tuples, one for each identifier, it involves a lot of detail to remember that:

< * z 3 > SYMTAB('XYZ')

gives (say) the datatype of XYZ. It is much clearer to have names for the various components of an N-tuple and to refer to:

TYPE('XYZ') .

The entire concept of selection of N-tuple components is ruined by its dependence on positional notation.

< ---*-- > N-tuple

makes it difficult to decide that the second component of an N-tuple is logically defunct and should be removed from the data structure. With positional notation, a component cannot be removed without changing all reference to all components physically following it. A structure declaration like that of PL/I (without the datatype information) has the strong advantage that elements can be freely added, deleted or rearranged without re-writing any of the selection notations in a program.

5) Conceptual inconsistency

a) SETL has no index sets. In a set language, a very natural way of specifying the application of an operation over an interval of integers is to have a notation for the set of integers from I to J inclusive.

$\sum_{1<I.N} I$ becomes +: [1,n] where the square brackets denote an index set.

Part of the power of the ∀ x ε S ... notation is that the operations on the elements of the set may well proceed in parallel. This power is lost if operations over integer ranges are forced to be serial. If it is in fact necessary to specify serial iteration over a set of integers in a specific order, then perhaps the recursion theory operator μ could be used to specify serial usage of integers from smallest to largest.

b) ∃[x] notation.

The side-effect of assigning to X is unnatural. It would be better to define the expression

$$\exists x \in S \mid x > 3$$

to have as its value the first such x encountered (or Ω if there is no such x) instead of True or False. The notation then becomes a shorthand for

$$x \mathbf{3} \{z \in S \mid z > 3\}$$

i.e., x is any element of the set of all members of S which are greater than 3 . The above notation can be simplified further if the concept of operations on all the elements of a set is allowed:

$$x \mathbf{3} S > 3$$

where $S > 3$ specifies the set of all elements of S which are greater than 3 . Searches over multiple sets like $\exists x \in S, \exists y \in T, \exists z \in U \mid e(x,y,z)$ could have an N-tuple as their value. In the present notation it is impossible to use the value found by an existence search, $\exists[x] \in S \mid \ldots$, without assigning a name to it (x) , and using this name in a separate expression. There are other cases in SETL where the concept of assignment is forced, instead of allowing the mathematician's natural concept of embedding any value-expression in a larger expression. Some of these will be discussed below under Selection and Replacement.

c) Subroutine definition and External statement.

In SETL, it is possible to say:

```
y = COMPILE 'define subname ...   end subname;'
```

Is it allowable to say:

```
y = define sub ...   end sub;
```

i.e., to assign the name of a subroutine which is not compiled later, but is defined at the same time as the assignment?

How is the ambiquity of parameterless functions resolved: in Y = functionname , where functionname returns an integer value, does Y become a function atom or an integer atom? How is the other result specified?

How can variable names other than parameters be bound to a subroutine so that they are stacked upon recursion? This appears to require some sort of declaration.

The EXTERNAL statement can be used to refer to a variable in an inactive subroutine:

```
define sub1;
   define sub2(x); ... end sub2;
   define sub3(y); sub2 external x; ... end sub3;
   ...
   end sub1;
```

What is the meaning of references to x in sub3 then sub2 is inactive? If the call chain has been:

sub1 calls sub2(a), sub2 calls sub3, sub3 calls sub2(b), sub2 calls sub3

are references to x in the first and second invocations of sub3 references to a and b respectively?

d) The problem of address vs. value in lists.

The insafter example on page 104 has the problem that the list cannot have two items with identical values because the value of an atom is used as its address. Is this a shortcoming of all list manipulation in SETL?

The example also has the bad property that it is legal to add a duplicate item, A , to the list, but after that Next(A) is undefined and will cause an error return, making it difficult to access either A . If duplicates are to cause this problem, it would be better for them to cause an error return upon insertion.

e) It is legal to say

A = Ω ; B = C ;

But it is not legal to say:

⟨A,B⟩ = ⟨Ω,C⟩; (page 74)

yet for any other value of D ,

⟨A,B⟩ = ⟨D,C⟩; is equivalent to A = D; B = C;

f) Selection and Replacement operators.

The whole section starting on page 80 is difficult to read. It appears that too many different concepts were forced into the same general notation, with a resulting unwieldiness. Selection and replacement operators have the absurd property that positional and subscript (structural address) notations may be intermixed.

⟨a,b⟩ = ⟨c,d⟩	is straightforward (a = c; b = d)
⟨a z 2,b z 1⟩ = ⟨c,d⟩	is a little confusing (a = d; b = c)
⟨a z 2,b⟩ = ⟨c,d⟩	is absurd (a = d, b = d)

What is the meaning of

⟨a,a⟩ = ⟨b,c⟩ ?

The * notation should not be restricted to only one appearance in an expression. For example, to form a set of ordered pairs which are the first and third components of a set of 3-tuples, it is presently necessary to use the following contortions:

TWO = {⟨⟨x--⟩z, ⟨--x⟩z⟩, z ∈ THREE}

(also note the two unrelated meanings of the symbol "⟨").

This could better be written as:

TWO = {⟨*-*⟩z, z ∈ THREE}

Or better yet:

TWO = ⟨*-*⟩ THREE

Are ⟨---*⟩ and ⟨-,-,-,*⟩ equivalent? (page 25 and page 83)

Why is the replacement concept necessary at all? If assignment could be easily imbedded in expressions (like in APL), replacement would appear to be redundant. The is function on page 128 would not be needed.

In summary then, SETL is at worst just a collection of strange notations and devices, and at best it is "just another programming language". Compared with the elegance and clean design of APL, SETL fails to attract the mathematical mind. It does not use the mathematician's symbols, his notation, his precedences, or his identities. Its mass of petty detail is no smaller than that of other languages. The algorithms presented are little more than transliterations of what would be written in ALGOL or APL. Yet the idea of a set as a datatype (or data structure) and the partially-fulfilled idea of specifying operations on all the elements of a set are very powerful notions and are good candidates for incorporation in some existing programming languages.

Example on page 148 re-written to include concepts of set operations, cross product, etc.

```
1    DEFINEF doms(nodes,entry,cesor);
2       nntr = entry X nodes-entry;  todo = entry;
3       WHILE todo ≠ NULL BEGIN
4          node ∈ todo;  todo = todo-node;
5          ∀c ∈ cesor(node) BEGIN
6             new = nntr{node} - c - nntr{c};
7             IF new ≠ NULL THEN BEGIN
8                nntr = nntr ∪ c X new;
9                todo = todo ∪ c;
10               END;
11            END;
12         END;
13      RETURN nodes X nodes - nntr - ⟨entry,entry⟩;
14      END doms;
```

NOTES:

line 1: nodes is a set of nodes, entry is an atom, cesor is a set of ordered pairs, the first component is a node, the second component is a set of successor nodes.

line 2: nntr is a set of 2-tuples. The first and second components are each single nodes. A given pair ⟨A,B⟩ signifies that it is not necessary to go through node B to reach node A. In general, there will be many pairs in nntr with the same first component. Note that this structure is different from page 148.

line 2: X means cross product of the two sets. In this case, the first set has only one element. The cross product of two sets is a set of ordered pairs (2-tuples).

line 4: The minus sign denotes set difference. A-B means the set A with all elements in (A intersect B) removed.

line 6: This forms the set of newly-discovered nodes which are not needed to reach c : those not needed to reach node; minus c itself, minus any nodes previously discovered.

line 8: This line could just as well go before the IF. Cross products involving the null set correctly give the null set, which can then be properly used in the union.

line 13: A set of pairs is returned, each pair of the form ⟨node, back dominator⟩ . Again, this is a slightly different structure from the one on page 148. The expression reads: all pairs ⟨node,node⟩ minus the pairs ⟨node, notneededtoreach⟩ leaving the pairs ⟨node,neededtoreach⟩ , minus the special case that the entry is defined not to back dominate itself.

Example on page 148 re-written for compactness.

```
 1   DEFINEF doms(nodes,entry,cesor);
 2      nntr = entry X nodes-entry;  todo = entry;
 3      WHILE todo ≠ NULL BEGIN
 4         node todo;  todo = todo-node;  s = cesor(node);
 5         new = s X nntr{node} - ⟨s,s⟩ - nntr[s];
 6         nntr = nntr ∪ new;
 7         todo = todo ∪ ⟨*-⟩ new;
 8         END;
 9      RETURN  nodes X nodes - nntr - ⟨entry,entry⟩ ;
10      END doms;
```

NOTES:

line 4: s is the set of successor nodes for the present pair of interest.

line 5: In this example, new is a set of pairs. Each pair ⟨A,B⟩ specifies that B is a newly-discovered node that is not needed to reach A . The expression reads: for each successor of node , nntr{node} are not needed to reach that successor (s X nntr{node}) , except that the successor itself should not be included (⟨s,s⟩) , and any previously-discovered unneeded nodes should also not be included (nntr[s]) . The last term could just be nntr itself, since only pairs with a successor as the first component are in new to start with. Note that the notation ⟨s,s⟩ is assumed to mean

$$\{\langle r,r\rangle , r\in s\}$$

A better notation could be found.

line 7: Only those successors that have newly-discovered notneededtoreach nodes are added to todo .

lines 5, 6, and 7 perform in parallel (for all successors) the same operations as lines 5 - 11 in the previous example.