

SETL Newsletter Number 26
The currently specified form of SETL
from a more fundamental point of view

May 3, 1971
J. Schwartz

In a set of general reflections concerning the programming process on which I am currently working, observations are made which allow one to react to various currently specified SETL features on a somewhat less ad hoc basis than has until now been available. This note will begin to elaborate these reactions. The basic issue is as follows: a programming language must enable one to choose each element eventually to form part of a total program in a situation as little complicated by detailed compatibility restrictions as possible. That is, it should help break the total programming problem to be solved into manageable and maximally independent pieces. From this basic principle various corollary principles may be derived, which are listed below with their implications, pro and con, for SETL.

A. Principle of decision postponement (by homologousness).

Semantically analogous entity-classes which might be substitutable for each other should be treated in a syntactically identical way, so that the decision as to which type of entity is actually to be used can be postponed.

Good in SETL: The use of sets, postponing or avoiding detailed choice of data representations.

Bad in SETL: 1 - the distinction between subroutine and macro-block invocations (coming from the key word "do") which forces the decision to use either a subroutine or a macro block to be made prematurely.

2 - the distinction between various types of entities having sequential nature and from which "n-th" elements might be chosen, or over which iterations might take place. Example: the n-th element of a sequence s of characters is $s(n)$; the n-th element of a character string s is $n \text{ elt } s$. Similar discrepancies between sequences and tuples (discrepancies less easily fixed) have troubled many.

3 - the distinction between iteration over a set, and various other related iterations, as for example, over a tuple or over the elements of a tree or list.

The first is

$(\forall x \in s)$ block;

the second must be written as

(while s ne \curvearrowright doing s=tl s;) x=hd s; block;

Iteration over a list with head x_0 and chained by a mapping ℓ can be written

$x=x_0$; (while x ne \curvearrowright doing $x=\ell(x)$;) block;

Many related ills are traceable to the absence in SETL of user-specifiable data types, to which operators having fixed form, but meaning dependent on the type of data object, might be applied. If such "typed data structures" were available we might for example define an iterator $(\forall x \in s)$ for each type of data object s , by defining some manner of "sequencing" over nominal "elements". This idea is compatible with many of the basic SETL usages.

In addition: It is bad to complicate the initial elaboration of a section of code with a decision concerning the manner in which the code is to be employed.

Conclusion: The introductory boiler-plate attached to macros and subroutines should sometimes follow rather than precede their bodies. For example:

```
begin; (while  $\ell(a)$  ne  $\curvearrowright$ ) a= $\ell(a)$ ;;  $\ell(a)=elt$ ;  
is block join(elt);
```

is sometimes preferable as a convention to

```
block join(elt); (while  $\ell(a)$  ne  $\curvearrowright$ ) a= $\ell(a)$ ;;  
 $\ell(a)=elt$ ; end join;
```

Similarly, the is operator should operate left-to-right rather than right-to-left, e.g.

$$x = (a+b) \text{ is } x1 * d$$

is better than

$$x = (x1 \text{ is } (a+b)) * d$$

A systematic review of the conventions applying to present data types, with a view to reducing the number of syntactic forms employed and to rounding out the set of operations provided would complement an exploration of the more fundamental and interesting question of providing programmer-definable data types.

Good in SETL: the formal identity which obtains between functional application of one set to another and calls to programmed functions.

B. Principle of Grouping by Logical Relation.

Those items with whose details an item I is most closely concerned should find places near I, without distracting intermediate material entering as a nuisance.

Good in SETL: "doing" option of "while" statement; "quit"; "continue". It might also be worth expanding the "while" statement to include an option

$$(\text{while } C1 \text{ when } C2 \text{ doing block};)$$

equivalent to

$$(\text{while } C1 \text{ doing block};) \text{ if } \underline{n} \text{ } C2 \text{ then continue};$$

as well as a special case

$$(\text{while } C1 \text{ when } C2)$$

Bad in SETL: The if-statement, which intermixes the controlling conditions of a subcasing operation with the transformations to be performed in the various subcases; especially

when nested. A possible form, which has the interesting property of exploiting the two-dimensional nature of paper, is as follows. (We introduce an "iff" statement, and describe it in partly two-dimensional terms.)

The iff-statement consists of a header and a trailer. The statement header is introduced by the keyword iff, which is followed by a sequence of iff-elements, each of which is either

i - a test designator, consisting of a name followed by the sign "+";

ii - an action-transfer label, consisting of a name followed by the sign ",".

These elements occur in the sequence determined by the following rules:

a - The first element following the keyword iff is a test designator. To its lower left (and to the lower left of any test designator) follows its positive-case descendant; to its lower right (and to the lower right of any test designator) follows its negative-case descendant. Those descendants can be either test designators (which will have positive- and negative-case descendants of their own) or can be action-transfer labels. Such labels have no descendants.

b - The deepest-rightmost descendant in the collection described by (a) (which descendant is necessarily an action-transfer label) will be followed by a semicolon rather than a comma; this semicolon terminates the header of the iff-statement, and introduces its trailer.

Before going on to describe the (less unusual) structure of a trailer, we give the following example of a header:

```
iff                nodeterm +
                   arlexalts +          islockey +
                   findalt, maynext,    keypres +      arealts +
                                   aresecalts+maynext, findalt, maynext,
                                   findalt, maynext;
```

This header describes the same collection of tests and transfers as the following if-statement (we momentarily pretend that the test-names refer simply to boolean variables).

```
if  nodeterm  then
    if  arlexalts  then go to findalt; else go to maynext;;
else if  islockey  then
    if  keypres  then
        if  aresecalts  then go to findalt; else go to maynext;;
        else go to maynext;;
    else if  arealts  then go to findalt; else go to maynext;;
end if  nodeterm;
```

Note that the contextual information necessary to read the collection of tests appearing the above example (and also to write this collection without errors) is much more readily available (since available in a more local context) in the iff-layout than in the conventional "nested-if" layout, even when the latter makes careful use of indentation.

The "trailer" section of an iff-statement consists of

c - A collection of actions with tests, each introduced by a label identical with one of the test labels occurring in the header section. Several actions may precede one test. An action is simply a SETL statement; a test is a SETL expression having a Boolean value, and is introduced by an occurrence of the sign "=" and terminated by a semicolon.

d - A collection of actions, each introduced by a label identical with one of the action-transfer labels occurring in the header. (An action-transfer label occurring in the header need not label any such action. If it does not, it must occur elsewhere in the same SETL subroutine, in which case the corresponding header-entry is interpreted as a transfer to the label in question.) A labelled action of this kind will be executed when a sequence of executed tests brings control to the corresponding header label. After the action is executed (and assuming it contains no transfers) control will pass to the next statement following the

iff-statement.

e - An iff-statement is terminated either by

ei - A (repeated) semicolon;

or,

eii - "end iff;"

or

eiii - "end iff token;"

The following example, expanding on the example given above, shows a complete iff-statement (adapted from the initial portion of "postparse"), and the if-test in conventional form to which it corresponds.

```

iff
    nodeterm +
    arelexalts +
    findalt, maynext,
    keypres +
    aresecalts+maynext
    findalt, maynext;
    islockey +
    arealts +
    findalt, maynext;
nodeterm:= nodtype(node) eq ⌒ ;
arelexalts:= (setalts is lexalts(synt)) ne ⌒ ;
islockey:= (keyloc is desc(node,keyloc)) ne ⌒ ;
arealts:= (setalts is altset(synt,nodtype(node))) ne ⌒ ;
keypres:= (keydesc is desc(node,keyloc)) ne ⌒ ;
aresecalts: key= if (x is nodtype(keydesc)) ne ⌒ then x
    else hd tl keydesc; = (setalts is secaltset(synt,key))ne ⌒ ;
end iff;

```

Now the conventional form:

```

if nodtype(node) eq ⌒ then
    if (setalts is lexalts(synt)) then go to findalt;
    else go to maynext;;

```

```
else if (keyloc is desc(node, keyloc)) ne ↪ then
    if (keydesc is desc(node, keyloc)) ne ↪ then
        key = if (x is nodtype(keydesc)) ne ↪ then x
                else hd tl keydesc;
        if (setalts is secaltset(synt, key)) ne ↪ then
            go to findalt; else go to maynext;;
        else go to maynext;;
    else if setalts is altset(synt, nodetype(node)) ne ↪ then
        go to findalt; else go to maynext;;
end if nodtype;
```

Useful additional options:

i' - A test-designator may optionally be replaced by what it would label, enclosed however in parentheses. That is, it may be replaced by a parenthesized sequence of SETL statements, terminated by a Boolean-valued expression (introduced by "=").

ii' - An action label may optionally be replaced by what it would label, enclosed however in parentheses. That is, by a parenthesized sequence of SETL statements.

A serious problem exists in regard to data structure layout. Layouts are now defined implicitly by the code which addresses them, and pieces of this code can be scattered over many routines. The difficulties which arise are of course alleviated by the simplicity which SETL structures have when compared to structures in lower level languages. Nevertheless, in complex situations, these structures can grow elaborate, and a mechanism for centralizing the basic structural decision concerning them is desirable. See also the paragraph immediately below.

C. Principle of Structural Isolation.

The inner details of a program section should be isolated to a maximum degree from detailed conventions established in other

program sections; semantically unitary entities established elsewhere which are to be invoked or accessed should be represented by unitary names and not by complex sequences applied to externally defined conventions. The main problem of this kind in SETL comes from the present limitations on the left-hand side of expressions. Suppose, for example, that we wish to make use of a data structure f in which an item of given semantic meaning is stored in $f(A)$ if a flag is set, but in $f(B)$ otherwise. It is most natural to write

```
itm f=expn;
```

which since it is isolated from external details concerning storage conventions is distinctly preferable to the more expressive

```
if flag then f(A) else f(B) = expn;
```

but SETL presently forces us to write

```
if flag then f(A)=expn else f(B) = expn;
```

which is even more undesirably explicit.

An attempt at a systematic analysis of this point will follow in a subsequent newsletter.