SETL Newsletter #133

J. Schwartz
June 20, 1974

A Higher-Level Control Diction.

1. Introduction; *Pursue* Blocks.

If dictional and semantic forms of higher level than those utilised in SETL were available to us, we would be able to regard SETL programs as 'hand compiled' versions of programs originally existing in a still higher level language. This would have several important advantages:

i. We would know, on firmer grounds than we now do, what sorts of constructions were likely to appear in typical SETL programs.

ii. We might become aware of higher-level constructions which SETL can only translate in a clumsy way, and this might suggest extensions to or modifications of SETL. In general, we could expect to design SETL with a surer hand if we were able to regard it as a cut-back version of a language of higher level then itself.

iii. A new level of optimised translation, with SETL as its target language, would emerge for study.

Till now only a few dictions of higher level than those provided by SETL have been suggested as SETL extensions. These are:

a. The suggestions for 'prescriptive' dictions which grow out of R. Krutar's generalisations of the SETL sinster call notion; cf. Newsletters 59 and 30.

b. Pattern matching dictions, generalised iterators, and other miscellaneous suggestions of Jay Earley; cf. Newsletters 52 and 56, 56A,56B,as well as the papers of Earley cited in NL.

c. Nondeterministic control dictions, as developed by Hewitt, Sussman, and others in the PLANNER, CONIVIVER, and QA4 languages; cf. AIA Newsletter 12.

The present newsletter will suggest a control diction of higher level than these presently available in SETL. The diction to be suggested is related to the dictions noted in pointed (a) above; like them, it has its origin in the observation that in many cases the intent of a code sequence is simply to force some set of conditions to hold simultaneously.

The new dictions we introduce center about the notion of a *pursue iteration*. Such an iterative block is opened by a header having the form

(1) (pursue *forall-iterator*) *block ender*.

Here, *forall-iterator* designates any SETL iterator of the ∀ type; *block* any block which could follow such an iterator; and *ender* a punctualing terminator which can either be ';', 'end ;', 'end pursue', etc. The semantic rules governing such an iterator within a SETL program P are as follows. Let the iterative block (1) be entered; let the variables bound in the forall iterator be $x_1,\ldots,x_n$. As long as there exist elements $V_1,\ldots,V_n$ in the range of the iterator such that the state of P's data is changed by substitution of $V_1,\ldots,V_n$ followed by execution of *block*, then *block* is executed. When no such $V_1,\ldots,V_n$ exist, the finish block (1) is exited.

We also permit degenerate constructions (1) in which the *forall-iterator* is null. The semantic rules which apply are much the same as those just explained, except that no bound variables $x_1\ldots x_n$ need to be replaced; in the degenerate case we execute the *block* of (1) as long as this changes the data state of P.

Here is a transitive closure routine written as a degenerate pursue block:

```
(2)                 (pursue) s = s + f[s];;
```

this may be compared to the standard SETL

```
(2')          (while s ne s + f[s]) s = s + f[s];;
```

note that (2) is noticeably less redundant than (2').
The following pursue block describes the bubble sort:

```
(3  (pursue 1 ≤ ∀ n < # f) if f(n) gt f(n+1) then <f(n),f(n+1)>
                                                = <f(n+1), f(n)>;;
```

This is very similar to the standard SETL

```
(3') (while 1 ≤ ∃n < # f | f(n) gt f(n+1)) <f(n),f(n+1)
                                             = <f(n+1), f(n)>;;
```

When several conditions are to be forced simultoneously the *pursue* construction can be distinctly more confortable than the *while* diction which comes closest to it in standard SETL. As an example, consider a graph *g* defined by a set *nds* of nodes and a map *naybs* which send
of all its neighbors. Let six sets all, al2, a21, bl, b2 be defined on each of the nodes of g, and suppose that we seek to find two functions f1 and f2 on g which for all n ε nds satisty both the equation

```
(4)  f1(n) = [+: n ε naybs(n)](all(n) * f1(n) + al2(n) * f2(n)
                                                       + b(n))
```

and the corresponding equation for f2.

The necessary program can be written in a very straightforward way as

```
(5)  (pursue ∀n ε nds)
         f1(n) = [+:m ε naybs(n)](a11(m)*f1(m)+a12(m)*f2(m)+b1(m));
         f2(n) = [+ m ε naybs(n)](a21(m)*f2(m)+a22(m)*f2(m)+b2(m));
     end;
```

2. A remark on the optimisation of Pursue Blocks.

We shall now describe a method which may in some cases allow pursue blocks to be optimised automatically; the same method is potentially applicable to other SETL iterative forms. Consider a pursue construction of the form

(6) (pursue $\forall x \varepsilon s$) *block*;

and let *active(s)* denote the set of all $x_o \varepsilon s$ which have the following property: if x is replaced by x_o and *block* is executed, then some part of the data environment of the SETL program containing (1) is changed. When x_o ε active(s) and *block* is executed, active(s) may of course grow; moreover, it may be possible by inspecting *block* to determine the set s' of all elements which could possibly be added to active(s) when *block* is executed. Suppose that this is possible, and more specifically suppose that one can generate a expression $\phi(x_o, a, x_1, \ldots, x_n)$, involving x_o, the current value *a* of the set *active*(s), and certain other variables $x_1, \ldots, x_n$ appearing in *block*, such that $\phi(x_o$, active(s), $x_1, \ldots, x_n)$ must certainly include s'. Then the pursue iteration (6) can be compiled as follows:

```
(7)  active = s;
    (while active ne nl doing active = φ(x,a,x1,...xn);) block;
```

When (6) is transformed into (7) by the process we envisage it may be found necessary to add to the set *active* all x for which a relationship $f_1(x_o) * f_2(x)$ ne nℓ holds; where f_1 and f_2 are maps which appear in *block*. To guarantee that these x can be found efficiently, an optimising compiler may chose to make use of the inverse map f_2^{-1}. If this is done, code updating the value of f_2^{-1} may have to be generated.

As an example of all this, consider the bubble-sort program (3). It is seen by inspection of the *block* B appearing within the pursue iterator (4) that when B is executed for a particular n only n-1 and n+1 can be made active. Thus a suitable optimiser might be able to compile (3) as

```
(8)        active = {n, 1≤n<#f};
          (while active ne nℓ)
                 n from active;
                 if f(n) gt f(n+1) then
                         <f(n), f(n+1)> = <f(n+1),f(n)>;
                         if n gt 1 then (n-1) in active;;
                         if n ℓt # f then (n+1) in active;;
                 end if f(n);
           end while;
```

Generally speaking, (8) is a better algorithm than (3); especially if, as we may assume without grave lack of realism, an optimising compiler handles the set *active* appearing in (8) either as a bit-vector supplemented by a list or simply as a bit-vector.

As a second example, consider the transitive closure routine (2). If this is rewritten slightly as

```
(9)              (pursue ∀x∈s) s = s + f{x};;
```

then the optimising procedure we have suggested might be able to realise it as

```
(10)      active = s;
          (while active ne nℓ)
                    x from active;
                    news = s + f{s};
                    active = active + (news - s);
                    s = news;
          end while;
```

In many cases, (10) will perform much more efficiently than either (2) or (9).