SETL Newsletter # 2, November 10, 1970

A critical comment by Pat Goldberg

The SETL language of J. Schwartz is an attempt to take the primitive operators postulated in the Zermelo-Frankel axioms of set theory and to implement them in a context useful for programming. That is, Professor Schwartz has described a programming language in which (finite) sets are a basic data type for which the primitive operators implied by the axioms are supplied. Atomic elements of type integer, characer, and bit, as well as the obvious operators on them, are also permitted. These basic items are fleshed out to a programming language by the inclusion of an assignment operator, basic control functions, and a procedure definition facility.

The intention is that SETL should serve as an executable specification language. That is, a programming process is envisioned in which a program is initially written and executed in SETL; finally, for optimization purposes, the program is hand translated into BSL. This hand translation not only requires deciding on BSL code for the operations involved, but also requires decisions as to how to represent the particular sets used as BSL data structures.

There are three, almost separate, aspects of this proposal that are worth commenting on: the viability of a two-stage programming process for systems programming; the usefulness of SETL, considered as a thing in itself, for specifying systems programs; and the acceptability of the SETL - BSL interface as currently specified. We shall address each of these items in turn.

It is rarely the case in the construction of a system that the system as finally implemented is completely specified before coding begins. This situation can be ascribed neither to laziness nor to weakness of will on the part of the designers; rather, it is largely due to the intrinsically evolutionary nature of the design process. Generally speaking, the initial design is modified in many ways before an acceptable system is constructed. These modifications are frequently the result of observations of the running of actual system modules or the operation of the system under accurate load conditions. These may result in changes either to the system data base or to the module organization, or both. The observations that lead to these changes generally cannot be made on a grossly unoptimized version of the system, since such a version does not necessarily reflect the timings and conflicts that will occur in the optimized version. Furthermore, the size and number of test cases that need to be run in order to establish the nature of the difficulty are usually large enough to overwhelm an interpretive system (I have personally had this difficulty using APL as a specification language).

Given that the measurements that lead to modification must be done in the lower level language and given that the translation from the specification language to the lower language is not automatic, the results are predictable; namely, the specification language is used only for the first iteration of the design, whereas modifications are made on the lower level program. The pressure of time and human nature being what they are, it slowly but surely becomes the case that the specification program does not specify the current design. At the end of the program, if the designers are conscientious, there may be a great push to recode the final design in specification language; in the meantime, much of the advantage has been lost.

For these reasons, it seems much more desirable to attempt to design a language which can serve both as a specification language and as the utlimate programming language, and to build one that incorporates as much possibility for flexibility as one can. From our point of view, this is the only possibility for specifying programs formally and for also guaranteeing that at all stages of evolution, the design specification is accurately reflected in the implementation.

We now leave the question of the viability of two-stage programming and turn to the question of the adequacy of SETL for the specification of systems programs. We shall not be concerned with the efficiency of the constructions, but rather with their adequacy and desirability.

One particularly useful feature of SETL is the ability to define unordered sets and to quantify over them. In systems code, it is not unusual to want to examine all elements of a set, although the order of examination is not important. In current languages this can be accomplished only by imposing an (arbitrary) indexing on the elements, either by putting them in an array or by connecting the elements together by pointers. It is interesting to note that part of Lowry's ESL proposal is aimed at ameliorating this situation.

There seem, however, to be some difficulties with SETL, at least as we understand the language. Since some of these problems are important while others are sytlistic, we have attempted to list them in decreasing order of importance.

1. In systems programming, it is frequently necessary to have a (non-atomic) element contained in more than one set. For example, a data control block may be contained not only in the set of all data control blocks, but also in the set representing some queue. It is important that the same element, not a copy of the element, occur in both sets, so that an updating of some field in the element will be reflected in both sets. But the basic SETL construction of adding an element X to a set B

    B = B. with .X

    seems to imply a copying -- or at least the effect of copying. In fact, one needs language to express both possibilities: both to add a copy of an element to a set and to add a reference to an element to a set.

2. Any reasonable specification language must make the data interfaces between modules absolutely clear. A module must specify not only the names of the formal parameters, but also at least the expected shape. In SETL, which is virtually without declarations, one can discover the kind of arguments required only by examining the flow of the program -- i.e., by understanding <u>how</u> the module accomplishes its task as well as what its task is. This means that syntax checking cannot determine whether or not two modules are compatible even in the most primitive sense. This difficulty is well illustrated in the example given of the Cocke-Younger parsing algorithm, where the first argument is required to be a rather complex structure. Without the accompanying English prose the structure of this argument would be extremelydifficult to determine.

3. Procedures seem to appear in SETL in only the most primitive of ways; in particular, procedures cannot be sent as arguments to other procedures, nor can procedures be members of sets. Furthermore, procedures cannot produce references (see above remarks), but only values. All of this makes it very difficult to hide the fundamental accessing methods that are being used under procedure references or to tag a set with a procedure for accessing its members. The presence of such features can add immeasurably to the ability of a program to maintain a certain flexibility concerning its data structures.

4. The rules for variable scoping in SETL are rather different from those used in current block structured languages and are no improvement. In particular, a non-local variable can be declared to be the same

one as the one of the same name, declared n blocks out. Such positional notation makes it difficult to add or or delete blocks from a program, since that may affect the count. Furthermore, even if such a facility is desirable, it seems more reasonable -- from a control point of view -- for the outermost (controlling) block to grant permission for this data sharing, rather than for an innermost procedure to be able to produce such side effects.

5. An explicit positional notation is used for accessing the elements of ordered sets: i.e., if W is a triple,

<*,-,*> W

is the ordered pair consisting of the first and last elements of W. This sort of notation really ties down the representation, in that any change in the ordering of the sets of W requires reprogramming the accesses. The APL notation is much better; for example

W[1,3]

is the same reference. This notation not only allows the computation of the indices, but permits the repetition and inversion of elements.

6. The representation of ordered triples is an ordered pair of ordered pairs is a silly bit of pedantry. It leads to such absurdities as the decision in the Cocke-Younger algorithm to use

<q,A,p> q > p

instead of the more natural

<p,A,q> p < q

simply because of the nature of the SETL accesses to be made. Clearly the access of p and the access of q should be equally trivial.

7. The language is too clever by half in the use of side effects. I cannot imagine intentionally displaying a specification language in which

<<X,*,Z > W,Z,X>

takes

<X,Y,Z>

as input for it and produces

<Y,Z,X>

as output. The utter lack of transparency in such an operation bodes ill for its correct interpretation by any reader.

But let us turn from these language details and look at the SETL-BSL interface. This interface seems totally inadequate. First of all, the form of a SETL call to a BSL module is different from a call on a SETL module. This means that as a module changes from a SETL module to a BSL module, all references to it must be updated wherever they occur. This is certain to be an unnecessary headache.

A more severe difficulty arises because no SETL data structure except an atom can be passed by reference. All other SETL structures must be copied into a contiguous section of storage, whose location is then passed to the BSL module. In an operating system context, where one wishes to develop rather intricately connected data structures (see above comments on references), it is difficult to see how to totally order the data base in a way that a BSL module can make sense out of. Furthermore, since this copying only reflects a static picture of a dynamically changing data base, this copying will have to be done repeatedly, each time the BSL module is called. The sheer inefficiency of this translation could overwhelm any efficiency gains gotten by recoding in BSL.

Finally, it seems unnatural that if one is planning to translate SETL modules into BSL that one would not have chosen the same scope rules for both. This disparity in rules will require extensive renaming of variables during the translation process.

It is probably pretty obvious by now that we do not view SETL, with its current bias as a promising development tool. Some of the ideas, however, in particular the introduction of unordered sets and the ability to quantify over them, could be an extremely useful addition to a programming language.