SETL Newsletter #32 May 15, 1971

Hyper-SETL procedural languages J. T. Schwartz

Though the SETL statement of an algorithm is now often shorter than a natural language description of the same algorithm, the natural language descriptions are nevertheless generally clearer in a somewhat elusive but still real sense. This suggests that natural language embodies useful descriptive mechanisms which SETL still has not captured, and which ought therefore to be sought after. The present short note contains a few preliminary observations in this direction.

1. A very important feature of natural language discourse arises from the fact that such discourse is highly error-tolerant. That is, numerous small deviations from standard grammar, of the kind that lead to such irritating situations in programming, are automatically corrected in normal discourse. In programming situations, one is normally reluctant to allow an automatic scheme for the correction of syntactic errors to be followed by execution. The essential reason for this seems to me to lie in the fact that once execution begins all feeling for the reasonableness of computation is lost, the computer in no real way monitoring the overall progress of its actions. In particular, even if an error might have been corrected in one of several ways, one will be chosen, and it will then not be possible to detect the fact that the computation which results is unreasonable, and that an alternate correction, leading to a different calculation, ought to be tried. These considerations emphasize the importance of various potential features of programming languages:

a. If the programmer's assumptions concerning his program could be made more readily available, then not only would additional static error checks be possible, but one might become considerably more willing to go ahead with program execution after error correction. Besides the 'assume' type of statement discussed in an earlier (mimoegraphed) set of notes on debugging, statements indicating the expected length of loops, the expected pattern of control transitions in a program, etc. might all be useful.

b. Error-correction mechanisms ought to interact much more intelligently with static global program analysis procedures (of the kind involved in optimization) than is now the case. For example, spelling-error-correction procedures could focus on variables live on program entry (improperly initialized variables), which are particularly likely to be misspelled versions of other variables; especially if these other variables have explicit definitions whose results are never used. Likewise, undefined functions are suspect as misspelled data objects.

These remarks also serve to emphasize the great importance of diagnostic aids. Mechanical aids, such as selective text retrievals and partial program analyses, which aim at increasing a programmer's maximum toleration for local complexity, are also desirable. It would for example be quite useful to be able to request display of all uses of a given assignment.

c. Beyond the relatively straightforward issues raised above we encounter the whole area of logical consistency checks in a higher sense. It is probably not possible to penetrate far into these matters now, though of course they deserve determined investigation.

2. Another important fact concerning natural language discourse, and one that it may be possible to exploit in a formal-language setting, is the fact that natural language makes clever use of syntactic ambiguities which are resolved by fragments of semantic information available from preceding declarations: For example, in natural language one may say

α: 'Proceed in increasing order thru the elements a of a sequence s. If a exceeds the element b which succeeds it, then interchange a and b.'

The most desirable translation of this into a formal language would be something like

β: '<u>sequence</u> s; (∀ a ∈ s) if a <u>gt</u> nextafter(a) (callthis b), then <a,b> = <b,a>; ... '

where the first statement is a declaration. But instead we are compelled to write

γ: '(1 ≤ ∀n < #s) if s(n) <u>gt</u> s(n+1) then <s(n),s(n+1)> = <s(n+1),s(n)>; ... '

in which a distracting position counter, which natural language manages to suppress, has become explicit, and in which the next element after a is referenced using the explicit definition of sequence succession, rather than, as in natural language, in terms of the logical relationship it bears to a.

The difference we have observed comes from the fact that various bits of semantic information concerning the notion 'sequence', as for example the fact that elements of a sequence may be thought of as having both a value a(n) and a position n, are not available for exploitation when the code γ is written. This has the consequence that a considerable measure of local complexity absent in the hypothetical code β appears in γ.

Consider what is necessary to make a 'hyper-SETL' programming style like β possible. We must first of all have some way of handling the basic declaration '<u>sequence</u> s;', which, somewhat after the manner of a macro, must give us the information needed to make all those deductions and transformations which are then necessary. These are roughly as follows:

i. Since <u>a</u> appears in the context a ∈ s, this name is being used for a 'sequence element' (this involves an 'implicit declaration').

ii. Iteration over a sequence is known to involve its elements in order, and really the indices of these elements. Thus (∀x ∈ a) is seen to be a shorthand for (1 ≤ ∀n ≤ #s), where 'n' is a position pointer, attached implicitly to 'a'; certain subsequent uses of 'a' will really be references to n.

iii. nextafter(a) is probably an elliptical reference to the sequence element a(n+1) (or to its position); this inference could only fail if there were something else about a (as perhaps its value, if this value were an integer) which could be incremented. Note then that in natural language a name is used ambiguously for a group of associated object-attributes, and the application of an operation to the name resolved by considering which particular attribute can logically be an argument of the operation. Among other advantages, the use of names in this style has the advantage of making explicit certain helpful logical associations between items which programming languages tend to treat syntactically as unrelated. This use of names also serves to hide various operations in which a known value of one attribute is used to select the corresponding value of another attribute. For example, in β, there is nothing corresponding to the explicit indexing operation s(n). This small effect can of course become quite large when more complex data structures than sequences are being addressed.

iv. Since there would be no point to applying the 'comparison' operator gt if the positions n and n+1 were the objects of reference, the first uses of a and b in β must refer to values and not locations, i.e., to seq(n) and seq(n+1) respectively. Similarly, since an assignment operation must be 'indexed', the form

<s(n),s(n+1)> =<s(n+1),s(n)>

implied by the

<a,b> = <b,a>

of β can be deduced.

We may in summary list certain of the principal notions that would have to enter into the design of a compiler capable of accepting inputs like β. Rather than treating tokens as undifferentiated names' after the fashion of current compilers, a reasoning compiler would have to associate specific attributes with tokens used to represent variables. Some of these attributes could be explicitly declared; others would have to be deduced from the contexts in which the tokens were used. The manner in which a text-fragment was to be expanded would depend not only on the keywords present in a text but also on the attributes of the tokens which it contained. (Note that the kind of 'attribute-dependent' macro-expansion style which this suggests is also not standard). In this way, by using a single name to represent various mutually associated attributes, we recreate within a programming language the vital natural-language notion of 'object'. This enables us to hide from view all the detailed code which, given one or more attributes of an object, accesses its other attributes.

We see from the above that ambiguity is exploited in various ways in natural language. Among other things, it allows a type of decision postponement. This suggests that the use of a parsing style well adapted to handle syntactic ambiguities might be appropriate to programming language also, and that the development of parsers having this characteristic might be a useful first step toward 'reasoning' parsers of the kind we have projected.