SETL Newsletter Number 52
Comments on SETL

September 16, 1971
Jay Earley

SYMMETRIC USE OF RELATIONS.

In keeping with the idea of separating semantics from implementation and the concept of decision postponement, the SETL user should be encouraged to set up his SETL data structures in terms of relations so as to leave open the issue of exactly what access paths are to be used. This distinction between access paths and relations is discussed in detail in [1] and [2]. In addition I have implemented an access path language called VERS [3,4] which I have done some experimenting with. The basic idea of an access path setup is that the user must know beforehand how his data structures will be accessed (not necessarily details of implementation, though). In SETL, this would mean always using sets of n-tuples only as functions from the first m elements to the last n-m elements, and never using the set former. SETL, of course, allows the latter construct, but restricts relations to being used only as functions. For example:

Consider an Algol symbol table. Suppose we are implementing on interactive Algol, so we need the entire symbol table all the time and we're not sure exactly how we will need to access it and modify it. We need two structures: one to show the tree-structured relationship of the blocks and one to associate with each block the identifiers declared in it. So we would like to have a binary relation on blocks called SONS and a binary relation on blocks and declarations called IDENTS. That way, we can write

$$SONS\{A\}$$

to refer to all blocks included in A, and

$$IDENTS\{A\}$$

to refer to all declarations made in A. But suppose we want to find the first block which encloses A, or suppose we have a declaration D and we want to find the block in which it was made. These can be programmed in SETL using "$\exists$" and the set former, but one should be able to write them as easily as the above two.

Alternatively, the user might set up all access paths explicitly by having four sets instead of two; that is, having a FATHER set and a BLOCK.OF set, so that he then could write

FATHER(A)

to get the enclosing block and

BLOCK.OF(D)

to get the block in which D was declared. However, then he must add elements to two sets whenever a new block is added (or delete from two whenever a block is deleted). Clearly, we don't want the programmer to be forced to decide which access paths he will use (or use most often) before he needs to.

I propose the following construct: One may use any relation as a function from any of its domains onto any others. This is done by putting "$\ast$"'s in the domain position wanted in the result of the function. Thus, we would write

SONS($\ast$,A)

to get the block enclosing A, and

IDENTS($\ast$,D)

to get the block in which declaration D appears. These then are generalized to work with [], $\{\}$, or () in the obvious ways. If A is an n-ary relation, then

$A(B_1,\ldots,B_m)$

is an abbreviation for

$A(B_1,\ldots,B_m,\overbrace{\ast,\ldots,\ast}^{n-m})$

and the same for $\{\}$ and [].

In addition, one might want to do the following: Suppose he has a ternary one-to-one relation; he would like to supply a value for one of the domains and ask for the corresponding value in either of the other two (or both). Right now, he can ask for both by writing

$$R(*, b, *)$$

if he supplies the second, for instance. But he should do equally as well to ask for only the third by writing

$$R(-, b, *)$$

This can be generalized meaningfully to relations which are not one-to-one as follows:

$R\{-, b, *\}$ means $\{c \mid \exists a \ni \langle a,b,c\rangle \in R\}$

So, in short, "$*$"'s specify output domains, filled-in entries specify input domains, and "-"'s specify "don't care" domains. We might also want to define what it means if all domains are filled in. This should be a set membership test as follows:

$R(a,b,c)$ means $\langle a,b,c\rangle \in R$

This proposal or course makes it more difficult to get an efficient implementation, but it should be quite easy for an optimizer to notice which access paths of a relation are actually used and to set up these so that they can be accessed as efficiently as what is allowed in SETL now.

<u>References</u>:

(1) Codd, E. F. A Relational Model for Large Shared Data Banks.

(2) Earley, J. On the Semantics of Data Structures, Courant Institute Symposium on Data Bases, 1971.

(3) Earley, J. Towards an Understanding of Data Structures. <u>Comm. ACM</u>, Oct. 1971.

(4) Earley, J. and Caizergues, P. VERS Manual. Computer Science Dept. University of California, Berkeley, 1971.

TUPLES, SEQUENCES, AND STRINGS

SETL has the above three constructs, all of which have approximately the same properties and the same kinds of operations available on them. This leads to confusion, difficulty in deciding which construct to use, and it violates the principle of minimalization of concepts for programming languages. Furthermore, tuples are clearly intended to be used for two very different purposes: (1) as fixed collections of objects where each member of the collection is accessed by name or position, and (2) as vectors are used in conventional programming languages or in, perhaps, APL.

I suggest that there should be basically two kinds of constructs which should replace the above three. Let me call them tuples and sequences. These are much different from SETL tuples and sequences, however. A tuple is much more restricted than a SETL tuple and a sequence considerably more general than a SETL sequence or tuple. I present only general concepts, not details of syntax.

Tuples. A tuple is a fixed collection of objects which is referred to either by writing the tuple syntactically, <a,b,c> or by naming the domains. This latter way should be clarified: An n-tuple actually defines a type of blank atom and n unary functions defined on that type. That is, we can represent a production in BNF as follows:

PROD=<DEF ∈ STRING,RT.SIDE ∈ SEQ(STRING)>

This defines a blank atom of type PROD with two unary functions DEF and RT.SIDE. DEF maps PROD's into STRING's and RT.SIDE maps PROD's into sequences of STRING's. We can use these functions on the right or left side of assignments:

```
S=RT.SIDE(P)
DEF(P)="E"
```

In addition, we can create a PROD and initialize it:

```
          P=PROD<"E",["E","+","T"]>
```

(Here the brackets are used to represent a constant sequence.) We can also use tuples in tests and set formers as in SETL. Unlike in SETL, tuples cannot be referenced by ordinal number or head and tail, and they cannot be iterated on or concatenated. Instead, we have sequences which have these properties, and more.

Sequences. The essence of a sequence is the ordering relation which defines it. So a sequence will be a binary, one-to-one relation on two identical domains of blank atoms of the appropriate type. Each blank atom also has a function VALUE defined on it which maps the elements on the sequence onto their values. Then, in addition to NEXT, PREVIOUS, and VALUE defined on sequence elements, a sequence will have various primitives defined on it such as FIRST, LAST, n'th, iterate over, concatenate, INSERT and DELETE an item (or sequence).first or last or at any point in the sequence, various subsequence operations and others. Notice that we now get the following advantages:

1. Strings can now be just sequences of characters or bits.
2. A subsequence is just a subset of the original sequence.
3. No prior decision need be made about whether the user wants a sequence or a tuple or a string. It is always a sequence.

Because we have defined sequences using a NEXT relation instead of a map from the integers we have the ability to insert and delete items at arbitrary places in the sequence without destroying any global references we may have to elements of the sequence.

Generalized sequences and iteration. The previous sequences are all data structures which are explicitly built up by the program. But just as some sets are implicit, so should some sequences be. So [1,n] could be the sequence of integers between 1 and n; [S], where S is a set, would be a sequence of the elements of S in some unspecified order. Most interesting would be to define a sequence by a generator subroutine or coroutine which would produce a new element of the sequence each time it was called, and finally fail when the sequence had ended.

Now, if we have the above constructs, then only one iteration statement in the language is needed: an iterator over sequences. Everything else is a special case of that.

CONDITIONS ON SETS.

This proposal would allow the programmer to attach a condition to the declaration of a set type or the creation of a set. This would be a boolean expression (presumably involving the set S) which would specify a condition which S must always satisfy, such as "S is a one-to-one binary relation", or "S does not contain two members x and y such that F(x)=F(y)". The most obvious use of a feature such as this is in debugging. Here, it would have the effect that if the program incorrectly tries to add or delete an element in S which would have caused a violation of the condition, an error message would be given. However, this feature can be used in a more fundamental way than this. It can be used to actually direct the flow of control in a correct program. This might frequently be used to detect errors in the data, rather than in the program; but it is also useful for performing the same function as an if statement, but in a more convenient way.

In order to do this, one needs to have the concept of "failure return" in the language. This feature is in SNOBOL4 and VERS. The idea is that any primitive in the language (or programmer-defined routine) can fail because of the detection of an error or some similar failure-related condition. In VERS, the user can specify that a particular use of such a primitive may fail by putting a "\" after the call followed by a label:

$$P(a,b) \setminus L$$

then if this call on P fails, the program branches to L. SNOBOL uses somewhat different conventions, but with similar conseouences.

Using this feature and the conditions on sets, we specify that any primitive which would have caused a set to violate its condition, does a failure return instead. This gives us the control we want. For example:

Let's construct a symbol table for VERS. VERS has a one-level modified block structure in which an identifier may be declared either in a routine (routines are not nested) or in a "data block". A data block has with it a list of the routines in which its declarations are valid. Furthermore, no identifier may be declared to be in a routine in two different ways, even if one declaration is in the routine and another in a data block. We will enforce this restriction using a condition on the symbol table. We need four types: DATA BLOCK, ROUTINE, DECL, and IDENT. The relation ROUTS defines which routines a data block will affect.

$$\text{ROUTS} = \{<\text{DATA BLOCK, ROUTINE}>\}$$

The SYM.TAB relation associates a DECL with any ROUTINE's in which it is valid:

$$\text{SYM.TAB} = \{<\text{DECL}, \text{ROUTINE}>\}$$

and the relation NAME gives the identifier which is declared:

$$\text{NAME} = \{<\text{DECL}, \text{IDENT}>\}$$

Now the condition on SYM.TAB is

$$\nexists R \in \text{ROUTINE} \mid \{D1, D2\} \ \underline{\text{le}} \ \text{SYM.TAB} \ \{*, R\} \ \underline{\text{and}}$$
$$\text{NAME}(D1) = \text{NAME}(D2)$$

(Note that I am using syntax from an earlier proposal, "Symmetric use of relations".) If declaration D is made in data block DB, then we execute

```
            SYM.TAB{D}=SYM.TAB{D} U ROUTS(DB)\
                  PRINT("CONFLICTING DECL")
```

This updates the symbol table if that is valid, and prints the appropriate message if it isn't.

Implementation. If this feature is used in its full power, implementation might be extremely difficult. We would probably like to implement this by detecting whenever a primitive is executed which can affect the value of the condition, retesting the condition after it is executed, and undoing it and performing a failure return if the condition is now false. However, there will be primitives which can affect the value of the condition without changing the contents of the set S. This would be done by changing other sets or relations which contain members of S. This can be detected, but it would be quite expensive and complicated. Perhaps a reasonable compromise might be to redefine the meaning of a condition so it produces a failure return only on primitives which actually add or delete items in the set.