SetL-59 REVISED

title: An Algebra of Assignment

author: Rudolph A. Krutar

country: United States of America

affiliation: Courant Institute of Mathematical Sciences
New York University

address: 251 Mercer Street
New York, New York 10012 U.S.A.

date: October 27, 1971 (submitted February 29, 1972)

area of paper: Programming Methodology

language: English

abstract: Assignment statements in procedural languages generally include the assignment of values to a limited class of expressions (such as subscripted arrays). It is the purpose of this paper to generalize the notion of assignment by proceeding along the lines of Schwartz' "Sinister Calls" [Schwartz 71]. The topics set forth below are motivation, technical details, and useful examples. The technical details include several abstract definitions. The useful examples include some surprises (like <u>let</u> $(1 \leq \forall j < n \mid A(j) \leq A(j+1))$ expanding into a bubble sort.

Sections

WHY?

HOW?

HUH?

Glossary

Approximate Syntax

References

WHY?

In most languages certain constructs select parts of a data structure but values cannot be assigned to these parts. For example, the APL assignment '(1 1 ⍉ M) ← E' should clearly mean 'assign the vector E to the diagonal of the matrix M'. Unfortunately APL permits only names and subscripted arrays to be assigned values. Any selection expression should be permitted because otherwise a design rule called 'programming generality' is violated: a construct should be permitted whenever it makes sense.

The designers of Algol 60 defined the for-statement in the following way:

> "Step-until-element. A for element of the form A step B until C , where A, B, and C, are arithmetic expressions, gives rise to an execution which may be described most concisely in terms of additional Algol statements as follows:

```
     V := A  ;
L1:  if (V-C) × sign(B) > 0 then go to Element exhausted;
     Statement S ;
     V := V+B ;
     go to L1 ;
```

where V is the controlled variable of the for clause and Element exhausted points to the evaluation according to the next element in the for list, or if the step-until-element is the last of the list, to the next statement in the program." [Naur et al. 60: 308]

They had no idea at all that this definition had any flaws (so says Perlis). (What does '<u>for</u> A[NEXT] ← 1 <u>step</u> INC <u>until</u> 100 <u>do</u>;' mean where NEXT and INC are parameterless integer procedures? How many times are these procedures to be called? The intuitively consistent definition would call the procedures once each time the loop is to be executed.) Unfortunately, implementers took the definition literally. A verbatim implementation of the definition would require that the label 'L1' not be used in certain places because the definition uses it. The macro expansion defined in this paper would treat such definitions in an intuitively consistent fashion.

HOW?

The technical details of various macro schemes are presented below. There will be many neologisms defined in this paper. As Dodgson once remarked, "... any writer of a book is fully authorized in attaching any meaning he likes to any word or phrase he intends to use". [Dodgson 97: 165] The meanings of these neologisms will appear below in logical order and in the glossary in alphabetic order.

<u>Blocks</u>, <u>statements</u>, and <u>expressions</u> are all syntactic classes. A <u>block</u> is a sequence of statements each followed by a semicolon. In some of its uses it may be followed by an ending comprised of zero or more appropriate visual cues. Since the null string is not a statement, two semicolons in succession mark the end of a block. <u>Statements</u> include assignment statements,

go to statements, macro definitions, conditional statements, and generally anything which is defined by a macro definition. Expressions are constants having a priori values, program variables which hold them, and anything which is defined by some definition of assignment. The meanings of VAR=CONSTANT and VAR=VARIABLE are given a priori.

The naive macro scheme is defined in terms of the notions of similarity and simultaneous substitution. A parameter is a name which if related to the name of a syntactic class is assumed to be a member of that class (BLOCK,STMT,...); otherwise, it is assumed to be an expression. A correspondence is a mapping from parameters to expressions. Such a mapping can be applied to a phrase by replacing each parameter in the phrase by its corresponding expression. This is called simultaneous substitution. Two phrases are similar if, for some correspondence, they are identical under simultaneous substitution, applying the correspondence to each of them. This meaning is perhaps too general -- successive restrictions which are acceptable are:

(1) no parameter occurs in both phrases,
(2) one of the phrases has no parameters,
(3) each parameter occurs exactly once in the other phrase.

The uses of similarity in the various macro schemes below all comply with these restrictions.

The processor for the naive scheme must recognize statements of the form 'macro BFORM; BLOCK ENDING', remember them and process other statements by finding definitions for them in which BFORM and 'STATEMENT;' are similar blocks. The correspondence which makes them similar is applied to BLOCK and each statement of the resulting block is generated. Statements for which the macro processor has no definition are not processed by it any further. Each generated statement is processed in turn. The following statement should make naive a synonym of macro:

```
        macro naive BFORM; BLOCK ENDING;;
              macro BFORM; BLOCK ENDING; end naive
```

It should be included in a test deck for any implementation.

A statement is expandable if it has an a priori meaning or if each statement generated by its macro expansion is expandable. Let X be a program variable. Then an expression E is trievable if the statement X=E is expandable. It is storable if E=X is expandable. A register is any expression which is both trievable and storable.

The intent of a program is a collection of intended assertions about the program and assumptions about the data. An assertion is valid if it can be derived from these assumptions. A program is valid if each intended assertion is valid. The data is valid if it complies with the assumptions. A statement in a program is valid if it is expandable and the program is valid. The assertions posed but not intended may be called the extent of the program. (E.g. "this program requires at most..."). An optimizer will modify a program in such a fashion as to influence its extent without damaging its validity. Such a modification has no net effect -- if the original program is valid only when the modified version is valid.

A statement in a valid program is superfluous if removing it has no net effect. Another statement is equivalent to it if replacing it with the other statement has no net effect. Two adjacent statements in a valid program commute if reversing their order has no net effect.

Every program language has a domain of values V, which are indestructible, a family of mappings from V^n into V, and a countable set of program variables, each of which may assume any value in the domain. Given program variables $X_1,\ldots,X_n$ and a mapping $f: V^n \to V$, then $f(X_1,\ldots,X_n)$ is a function which may or may not have a simple representation in the programming language.

A function $f(X_1,\ldots,X_n)$ is safe between points L1 and L2 in a program if the assertion is valid that $(\forall t)((f(X_1,\ldots,X_n)=t \text{ at L1}) \supset (f(X_1,\ldots,X_n)=t \text{ at L2}))$. It is safe over any phrase which has one entry point and one exit point if it is safe between those points. A constant is a function which is safe between any two points in the program. A function is a subfunction of another if it is safe whenever the other is safe.

A trievable expression E is conformable to a storable expression S whenever S=E is a valid statement. Let X and Y be program variables, and let Q be a register. X conforms to Q whenever X=Q is superfluous following a valid statement Q=X. Whenever Y=Q is superfluous following Q=X and, for some function $f(X_1,\ldots,X_n)$, the assertion that $Y = f(X_1,\ldots,X_n)$ is valid following (Q=X; Y=Q;), then Q is *retrievable*, and $f(X_1,\ldots,X_n)$ is its *transfer function*. (X may or may not be among $X_1,\ldots,X_n$.) A register Q is *restorable* whenver the statement Q=X is superfluous following a valid statement X=Q. Whenever Q is also retrievable, it is a *field*.

Consider the following trieval and storage operations:

```
macro X=value(Y);; X=Y; end =value;
macro value(Y)=X;;      end value=;
```

A trievable expression E is a *defined function* whenever *value*(E) is a field. In assertions, references to the transfer function $f(X_1,\ldots,X_n)$ of value(E) may be abbreviated as E or as $Xi_1 \ldots Xi_k \vdash E$ should $Xi_1,\ldots,Xi_k$ not be explicitly named in E. If *value*(E) is not a field, then E is said to have *side effects*.

Two functions are *isomorphic* if each is a subfunction of the other. Theorem: Isomorphism is an equivalence relation. The *data space* of a function is its equivalence class under isomorphism. Any property of functions which is preserved under isomorphism applies equally well to data spaces. Data spaces may be safe, constant, equal (isomorphic), *subspaces* (subfunctions), etc. If every common subspace of two data spaces is a subspace of some particular common subspace, then this particular subspace is their *intersection*, (which is uniquely determined). The *union* of two data spaces has a similar meaning. Theorem: If $\langle X,Y \rangle$ is the pairing function (CONS in LISP) and F,G are functions, then the union of their data spaces is the data space of $\langle F,G \rangle$. Theorem: The closure of the set of all defined functions of a valid program under union and intersection is a lattice over $\subseteq$, the subspace relation. The superior node of this lattice is the *data base* of the program and the inferior node is its *constant space*. Two data spaces are *independent* if their intersection is the constant space. Otherwise they *overlap*. We write $D_1|\ldots|D_n$ to mean that $D_1,\ldots,D_n$ are mutually independent data spaces.

A block is restricted to a data space if every independent data space is safe over the block. A data space is live at some point in a program if putting some block restricted to that data space at that point would have a net effect. It is dead otherwise. A data space is marred by a block if it is not safe over the block. If every subspace of a data space is either marred by a block or dead on entry to the block, then that data space is mashed by the block. Point L2 in a program cannot be reached from point L1 if (True at L1 $\supset$ False at L2) is valid.

Theorem: A data base is dead at some point in a program if it is mashed before any defined function is trieved for which the data base is a subspace of the given data space.

These observations may be amusing:

(1) A field F may be made safe over a block by using the macro

macro ⊢ save F across BLOCK;; ⊣ TEMP
TEMP=F; BLOCK F=TEMP; end save;

(2) Functions inherit properties of their data spaces; fields are functions;

(3) The constant space is forever dead;

(4) The program variables are mutually independent fields

(5) A field may be used as a temp in a block if it is dead before and dead after the block

(6) An optimizer or a garbage collector may deallocate a dead field unless it is a constant (never throw nil away!) Every time a garbage collector is invoked, it must be restricted to some data space which is dead at the point of invocation.

neomacro definitions

Naive macro definitions have a very simple interpretation but a very complicated unintuitive behavior. Not only are some expressions evaluated many times, but local names may interfere with the interpretation in some common cases. We will first modify the naive expansion scheme to include provision for local variables, then we will invent a new scheme of protected definitions which provides for global variables, frozen variables and shorter expansions as well. These definitions can be compiled like procedures or expanded in line. Later we will define two more schemes, the symmetric definition (which is just an abbreviation) and the expression definition (which connects function definition to statement definition). All these inventions will be defined in terms of naive macros.

In order to provide for the introduction of unique names to an expansion, we make a block similar to a form (FORM) also similar to $\vdash$ FORM $\dashv$ VARS where VARS is a list of names not occurring in FORM. We make these names correspond to variable names unique to that expansion: names guaranteed not to occur elsewhere in the program. Consider this definition, call, and possible expansion:

<u>macro</u> $\vdash$ do S; $\dashv$ L0; L0: S; <u>go to</u> L0;;
<u>do</u> <u>do</u> L0 $\leftarrow$ 1;
L0607: L0608: L0 $\leftarrow$ 1; <u>go to</u> L0608; <u>go to</u> L0607;

Indeed the names correspond in an intuitive way. This device is analogous to the <u>local</u> statement of IMP [Irons 70:33]. In fact, the <u>local</u> statement could be defined by:

<u>macro</u> $\vdash$ <u>local</u> VARS <u>in</u> BLOCK; $\dashv$ DoIt;
    <u>macro</u> $\vdash$ DoIt; $\dashv$ VARS;
        BLOCK; <u>end</u> DoIt;
    DoIt; <u>end</u> <u>local</u>;

An expression $f(e_1,\ldots,e_n)$ is similar to a form $f(x_1,\ldots,x_n)$ when each subexpression e_j corresponds to a variable x_j of the form. Local variables $y_1,\ldots,y_m$ and global variables $z_1,\ldots,z_k$ are augmented to the form by writing $z_1\ldots z_k \vdash f(x_1,\ldots,x_n) \dashv y_1\ldots y_m$ and making these automatic correspondences:

x_j : e_j	as it was	$j = 1,\ldots,n$
z_j : z_j	global parameters made explicit	$j=1,\ldots,k$
y_j : y_j^{exp}	local names become unique to an expansion.	

The naive expansion of the definition

<u>macro</u> $z_1\ldots z_k \vdash f(x_1,\ldots,x_n) \dashv y_1\ldots y_m$; body;

may take place as though all the parameters were stated explicitly.

The global parameters $z_1,\ldots,z_k$ are global in the dynamic sense: they are those variables in use at the call (as in APL). Any variables of the body which are not in the form are global in static sense: those in use at time of compilation of the definition (as in Algol 60). These globals will have somewhat more use in terms of protected definitions. A macro definition of the new kind is called a <u>protected</u> definition. It has two forms, implicit and explicit (the former being more common), both defined as follows:

<u>macro</u> <u>def</u> STMT; BLOCK <u>end</u> CUES;;
    <u>def</u> STMT; $Xi_1 \ldots Xi_k$ ⊢ BLOCK ⊣ $Xf_1 \ldots Xf_m$ <u>end</u> CUES;
    (where $Xi_1,\ldots,Xi_k$ are live before BLOCK and $Xf_1,\ldots,Xf_m$ are marred by BLOCK) <u>end</u> <u>def</u> STMT;
<u>macro</u> ⊢ <u>def</u> STMT; $Xi_1 \ldots Xi_k$ ⊢ BLOCK ⊣ $Xf_1 \ldots Xf_m$; ⊣ $T_1 \ldots T_n$;
    <u>macro</u> ⊢ STMT; ⊣ $T_1 \ldots T_n$;;
        (∀ Xi_j unless suppressed) Ti_j=Xi_j;
        call or expand TBLOCK;
        (∀ Xf_j unless suppressed) Xi_j=Ti_j; <u>end</u> STMT;
    (where BLOCK is similar to TBLOCK wherein the variables $X_1,\ldots,X_n$ of STMT correspond to the distinct variables $T_1,\ldots,T_n$. If both Ti_j=Xi_j and Xi_j=Ti_j are generated, then those assignments are suppressed which would cause subexpressions of Xi_j to be trieved twice or assigned twice) <u>end</u> <u>def</u> ⊢ ⊣ ;

Certain initializations and finalizations are suppressed whether or not they have a net effect on the program. These suppressions are necessary in order that no definition be expanded more often than is intuitively reasonable. Perhaps this definition can be worked out in a cleaner fashion so that no statements need be suppressed. At any rate, the action is: initialize some parameters, call the routine or expand the definition, and finalize some parameters. This works for definitions of = as well as for many other statement forms. I assume that the explicit definition initializes and finalizes in the orders given explicitly; but that the implicit definition carefully defaults these orders to preserve their order in STMT (e.g. <u>def</u> A(J)=X; X J A ⊢... ⊣ A;).

The following definitions may clarify the use of the protection scheme. Then we will prove some theorems about it.

```
def R0=<R1,R2>;     R1 R2 ⊢ call LISP.CONS; ⊣ R0;
def R1=hd R0;       R0    ⊢ call LISP.CAR;  ⊣ R1;
def R2=tl R0;       R0    ⊢ call LISP.CDR:  ⊣ R2;
```

which are necessarily primitive. And the storage definitions:

```
def <A,B>=C;     A = hd C;  B = tl C;;
def  hd C = A;   C = <A, tl C>;;
def  tl C = B;   C = <hd C, B>;;
```

are not. In the primitive case the variables are the names of fixed locations (like machine registers) and the system is expected to use them. Of course, such a variable can be made into a temporary by saving it in another temporary and restoring it later. Good optimization can make statements like <X,Y>=<Y,X> boil down to T=X; X=Y; Y=T; but the optimizer must know the trivial identities involving CONS, CDR, and CAR (following CONS, both CAR and CDR are superfluous). Assignment itself can be defined:

```
def  A ← B;   A = B;;
def  A → B;   B ← A;;
def  A ↔ B;   <A,B> → <B,A>;;
```

each of which does the intuitively correct action.

<u>Composability theorem</u>: A statement $f(e_1,\ldots,e_n)$ involving subexpressions $e_1,\ldots,e_n$ is expandable using the protected definition

$$\underline{\text{def}}\ f(x_1,\ldots,x_n);\ \text{INITIAL} \vdash g(x_1,\ldots,x_n) \dashv \text{FINAL};$$

provided that:

(1) $\text{INITIAL} \subseteq \{x_j \mid e_j \text{ is trievable}\}$

(2) $\text{FINAL} \subseteq \{x_j \mid e_j \text{ is storable}\}$

(3) $g(T_1,\ldots,T_n)$ is expandable with $T_1,\ldots,T_n$ being program variables.

[Proof: the generated naive macro definition is:

```
macro f(x1,...,xn) ⊣ T1...Tn
      (∀xj ∊ INITIAL unless suppressed) Tj=xj;
      g(T1,...,Tn)
      (∀xj ∊ FINAL unless suppressed) xj=Tj;  end f;
```

If T_j=e_j is generated in the second line, then x_j ∊ INITIAL, and e_j is trievable (by hypothesis 1), hence T_j=e_j is expandable. Likewise, if e_j=T_j is generated in the fourth line, then x_j∊FINAL, and e_j is storable (by hypothesis 2), hence e_j=T_j is expandable. The third line is expandable by hypothesis, hence f(e_1,...,e_n) is expandable according to the definition of the term.]

In point of fact, the definition of assignment is ambiguous. Consider the two definitions and assignment:

```
def  y = f(x1,...,xn);  call f;;
def  g(z1,...,zm) = w;  call g;;
g(d1,...,dm) = f(e1,...,em);;
```

which we assume to be expandable. Which definition is expanded first? Both orders are shown below:

```
/* g first */                        /* f first */

  (∀i|...) zi=di;
  /* w = f(e1,...,en) */
     (∀j|...) xj=ej;                      (∀j|...) xj = ej;
     call f;                              call f;
                                          /* g(d1,...,dm)=y */
                                             (∀i|...) zi=di;
     w = y;                                  w=y;
     (∀j|...) ej=xj;
  call g;                                    call g;
  (∀i|...) di = zi;                          (∀i|...) di=zi;
                                          (∀j|...) ej=xj;
```

assuming that y is not initialized by f and w is not finalized by g. The expansions are almost alike: only initializations of left-hand variables and finalizations of right-hand variables are out of place. Stated as a theorem, this observation becomes:

Theorem: If no protected definition used in the expansion of a statement initializes left-hand variables nor finalizes right-hand variables, then the order of expanding them is immaterial.

[Proof omitted].

But the hypotheses of the theorem are rather commonly violated (cf. the definition of hd C = A). Which order is to be preferred? Initializations are frequently commutable because they rarely involve side effects. The finalizations might be made in left to right order (so f would be expanded first). If the choice depends only upon the sequential order of the definitions, then further analysis becomes cumbersome: (what does <A,B>=(A←<3,4>) mean when ← is defined by

def X=(Y←Z); X=Z; Y=Z;;

Will A be 3 or <3,4>?)
If g is expanded first then initializations are made from left to right and finalizations from right to left, completing the evaluation of (X←<3,4>) before assigning X=3. This might prove to have more intuitive appeal. If within a definition, the finalizations are made from left to right, then <X,X,X>=<1,2,3> is equivalent to X=3. This achieves the intuitive rule of evaluating subexpressions completely before finalizing cognate expressions. Then J=J+(J←3)+J is equivalent to J=J+6. Furthermore, <sign(X), abs(X)>=<-1,12> is equivalent to X=-12 unless X=0. Many wonderful theorems about the preservation of properties when fields are independent lurk in dark corners waiting to be discovered. (1:30 A.M.)

In many cases the storage and trieval definitions of an expression are remarkably alike. Two abbreviations which exploit this symmetry are:

```
(1)     macro sym def FORM1=FORM2; STMT;;
              def FORM1=FORM2;  STMT;  end trieval;
              def FORM2=FORM1; rev STMT; end storage; end sym def;
```

where

```
        macro rev EXPR1=EXPR2;;  EXPR2=EXPR1;  end rev = ;
        macro rev if COND then STMT1 else STMT2;;
              if COND then rev STMT1 else rev STMT2; end rev if;
        macro rev (∀COND) STMT;; (∀COND) rev STMT;  end rev ∀;
              . . .
(2)     macro ⊢ defx FORM = EXPR; ⊣ VAR;
              sym def VAR=FORM;  VAR=EXPR;  end defx;
        macro ⊢ defv FORM = EXPR; ⊣ VAR;
              def VAR = FORM;  VAR = EXPR;  end (FORM)=;
```

Now, conditional expressions are defined by:

```
        macro VAR = (if COND then EXPR1 else EXPR2);;
              sym def if COND then VAR=EXPR1 else VAR=EXPR2; end(if);
```

And these examples bear some interest:

```
        sym def stk X = Y;  X = <Y,X>;
        sym def X nee Y = Z;  <X,Y> = <Z,X>;

        defx A max B = (if A ≤ B then B else A);
        defx parts = <RANK,RHO,DEL,ABASE,VBASE>;
        defx tasks = pq JOBFILE;
        defx A[J] = (if pair(J) then <A[hd J], A[tl J]> else A(J));
```

The sym def statement form is just an abbreviation, but the defx statement form is the key definition which permits expression macros. Of course, it depends heavily on the notion of assignment.

A register is a file if every value stored in it may be retrieved once. If all values stored in a file have been retrieved, the file is empty and it should not be trieved until more values are entered. For each file type, there should be a field which defines emptiness of the file. For stk we might write:

```
        defv  stk X be empty  =  ¬ pair(X);
        def stk X be empty = b;  if b then X=0
                                 else if ¬ pair(X) then error;;
```

Then stk X may be cleared by writing 'let stk X be empty'. If the file is not supposed to be empty at some point, then 'let ¬ (stk X be empty)' generates an error if it is. This is equivalent to 'if stk X be empty then error' which verifies that the file is not empty. Remark: the occurrence of 'stk X' in 'stk X be empty' generates neither storage nor trieval expansions for 'stk X'.

The census conditions on a file are easily described by defining a file operator (√) which counts all values entered and retrieved. The second definition is the well-formedness condition for a census (the first gives meaning to assertions):

```
def assert b;  if ¬ b then error;
defv C ∧.≥ 0   =  (∀y ∊ range(C) | y ≥ 0);
def X = F √ C;  X=F;  C(X)=C(X)+1;  assert C ∧.≥  0;
def F√C = X;  F=X;  C(X)=C(X)-1;  assert  C ∧.≥  0;
def b = F√C  be empty;  b=F be empty;  assert b=(∀y∊range(C)|y=0);;
def F√C be empty = b;  F be empty = b;
                         if b then C = nl
                              else assert (∃y ∊ range(C)|y>0);;
```

With these definitions, if FILE is a file and CENSUS is a temp then FILE √ CENSUS is a file which may be used in place of FILE and which incorporates the requirements of files. A quantity F is a file if and only if F√C may replace every occurrence of F with no net effect on the program, C being a temp.

A priority queue is a file which always yields its smallest entry. The following definitions make pq A a priority queue.

```
def pq A be empty = b;  if b then A=nl else if A=nl then error;;
defv  pq A be empty  =  (A=nl);;
def ⊢ pq A=X ⊣ b,j,k;  b=true;  j=#A+1;  A(j)=X;
     (while b ∧ (j>1) doing j=k) k = j÷2; let(A(k)≤A(j)) nee ¬ b;;
                                                          end def;
def ⊢ X=pq A ⊣b,j,k;A(#A) nee (A(1) nee X) = Ω; j=1; b=true;
     (while b ∧ j<#A÷2 doing j=k) k=2*j;
          if k<#A then if A(k+1) ≤ A(k) then k=k+1;
          let (A(j) ≤ A(k)) nee ¬ b;;  end def;
```

The nee operator was defined earlier. It permits a register to be saved before it is assigned (X nee OLDX=NEWX;)

A generalized deque can be built rather easily if the symmetric sum operator is defined on atoms (or any other associative and commutative operator for which $A \oplus A = \Omega$ and $A \oplus \Omega = A$; Ω is most convenient but any particular value can be substituted; if $A \neq B$ then the exact value $A \oplus B$ is not important.) Genuinely symmetric lists will be defined, and stack operations will be permitted on either end. LINK and VAL are two SETL functions.

```
def A to B;  LINK(A) = LINK(A) ⊕ B;  LINK(B) = LINK(B) ⊕ A;
          end merge/break;
defv   dq A be empty  =  (LINK(A)=Ω);;
def dq A be empty =b;  if b then A=newat
                       else if LINK(A)=Ω then error;;
def ⊢ X= dq A ⊣T;  X=VAL(A); A nee T=LINK(A); A to T; end pop;
def ⊢ dq A = X ⊣T; A nee T=newat; VAL(A)=X; A to T; end push;
```

Then a function walker can look like:

```
let (dq OUT be empty) ∧ (dq NEXT be empty); dq NEXT=TOP;
(while ¬ dq NEXT be empty)
     begin TEMP=dq NEXT;  dq OUT=TEMP;
     if atom(TEMP) then continue while;
     let GEN be empty;  LEFT = GEN;
     (∀<x,y> ε TEMP) dq GEN = y;
     GEN to NEXT; TEMP=dq LEFT; NEXT=LEFT; end while;
```

The deque in GEN was built up and flipped around with no effort.

HUH?

Several surprises have turned up. It was thought that only a few expression types could reasonably be defined as fields. This is not the case. Storage definitions have been found which make fields out of many constructs in SETL and APL. The first and foremost is a boolean operation, membership:

```
def x∈S=b;  if b then S = S with x
                 else S = S less x;
```

Assigning a truth value to a predicate in this case causes the predicate to assume that truth value. (A bit is any predicate which is a field to which true and false both conform.) We can write 'let ¬ (3∈A)' by declaring:

```
def let b;  b=true;;
def ¬ b=z;  b = ¬ z;;
```

In principle, A can be viewed as a bit vector and the statement becomes A_3 = false. In practice, however, such viewpoints are ignored. A statement like 'let PRED' means "make PRED become true -- I care not how". An alternative definition would have changed x instead:

```
def x∈S=b;  if b then x = min(S)
                 else x = max(S)+1;
```

which is certainly a field when S is a set of integers. The more general definition is usually to be preferred. Yet another definition makes x∈S a field (because x = ∋S is superfluous after S = ∋S).

```
def x∈S=b;  if b then x=∋S else x=newat
```

where ∋S is a random element of S and newat is always a value distinct from all values previously generated.

Whenever the value of a field must be changed, the storage operation may make random changes. The definition of flip aids in describing this phenomenon:

```
def b=flip; b=even(SEED); SEED=MODULUS|RC+RA*SEED;;
```

which is a random condition. Assume that T=flip is superfluous whenever T is dead (i.e. SEED is not a variable of contention in the intent of the program). The definition of either permits a random choice between two variables:

```
defx (either A or B) → (if flip then A else B);
```

Then the logical connectives become:

```
def a∧b=c;  if c then <a,b> = <true,true>
                 else if a∧b then (either a or b)=false;;
defx a∨b = (¬ a)∧(¬ b);
defx a⊃b = (¬ a) ∨ b;
defx a≠ b = (a∧¬b) ∨ (b∧¬a);                    (proof, anyone?)
```

The set theoretic operations can be defined as fields:

```
def A int B = Z; (∀xεAuBuZ) xεA∧xεB = xεZ;;
def A  u  B = Z; (∀xεAuBuZ) xεA∨xεB = xεZ;;
def A  -  B = Z; (∀xεAuBuZ) xεA⊃xεB = ¬ xεZ;;
def A  ⊆  B = Z; (∀xεA u B) xεA⊃xεB = Z;;
def A  ⊕  B = Z; (∀xεAuBuZ)(xεA) ≠ (xεB) = (xεZ); end sym diff;
```

And they might be used in:

```
let (x ε A int B) ∧ ¬ ((y ε A u C)∨b);          (deterministic)
let (x ε A int B)  ⊃  ((y ε A u C)∨b);          (random changes)
```

The storage definition of quantified expressions leads to some intriguing results. Let ∀COND be any phrase like ∀xεS or 1≤∀j<#A; and let ∃[COND] correspondingly be like ∃[x]εS or 1≤ ∃[j]<#A. The storage definition for universal quantification may then be:

```
macro (∀COND|PRED)=b;;
     if b then (while ∃[COND]| ¬PRED) let PRED
          else error;  end ∀;
macro (∃COND|PRED) = b;; ¬ (∀COND|¬PRED) = b;;
```

Naively found, a violation of PRED is corrected, then the search starts over. The transitive closure of a set S under a function can be defined:

```
macro ⊢ C= f closure S; ⊣ x; C=S; let (∀xεS|f(x)εS); end closure;
```

A sequence A can be sorted by simply demanding:

let (1 ≤ ∀j ≤ #S | S(j) ≤ S(j+1));

if the earlier definition of ≤ is accepted:

def X≤Y=b; if b≠X≤Y then <X,Y>=<Y,X>; end ≤ ;

Setting this "sorted" bit generates the bubble sort but a clever enough optimizer would convert that to a radix sort. The tree sort (or heap sort) can be given in a few lines:

(1<∀m≤n) let (1<∀j≤m)|A(j) ≤ A(j÷2);
(n≥∀m>1) let A(1) ≤ A(m) ∧ (1< ∀j≤ m÷2|(A(2*j)⌈A(m⌊2*j+1))≤A(j));

but many superfluous tests are made. Maximum and minimum are defined by:

defx X ⌈ Y = (if X ≤ Y then Y else X);
defx X ⌊ Y = (if X ≤ Y then X else Y);

Remark: X ⌈ Y = Z has transfer function Z ⌈(X ⌊ Y).

Various arithmetic expressions can be fields. They are tabulated:

definition	transfer function (on reals)
def sqrt(R)=Z; R=Z**2;;	abs(Z)
def sign(R)=Z; R=sign(Z)*abs(R);;	(if R=0 then 0 else sign (Z
def abs(R)=Z; R=sign(R)*abs(Z);;	(if R=0 then 0 else abs(Z))
def floor(R)=Z; R=floor(Z)+fract(R);;	floor(Z)
def fract(R)=Z; R=floor(R)+fract(Z);;	fract(Z)
def M mod N=Z; M=M-(M mod N)+(Z mod N);;	Z mod N
def M mod N=Z; M=M-(M mod N)+Z;;	Z mod N but M÷N ≬ M mod N
def M÷N=Z; M=N*Z+(M mod N);;	Z if Z ≥0, otherwise?
def even(M)=b; M mod 2=(if b then 0 else 1);;	b restricted to true,fals
def ⍒ V=Z; V[Z] = V[⍒V];; where ⍋ and ⍒ are the grade up and grade down operations of APL.	Z restricted to permutations
def V ⊥ W = N; W = V ⊤ N;; where ⊥ and ⊤ are encode and decode of APL	(×/V) ∣N
defv (i,j) = j+(i*(i+1)÷2);;	i,j,k restricted to nonnegative integers
def (i,j)=k; i=floor((sqrt(1+8*k)-1)÷2); j=k-i; end decoding;	

Dualities

Some operations which are complementary can be defined as fields. The similarity-substitution package is a good example:

```
def D = P1∿P2; if P1 and P2 are similar then D=their correspondence
                                        else D=false; end =∿;
def P1∿P2=D;   if D≠false then P1 = application of D to P2
                          else P2 = Ω;  end ∿=;
```

Given the meanings of similarity, correspondence, and substitution defined on page 3 , then P1∿P2 is a field. If P1=> P2 is a rule in some transformation (like the macro processor), and a third pattern F is similar to P1 then F∿P2 = F∿P1 will cause F to assume its transformed value .

Try, for example: F=(x*(y-q)+x*q)-x*z, P1=(a-b)+b, and P2=a. After F∿P2=F∿P1, then F= x*(y-z).

Other complementary operations which demand scrutiny are:

1. parse-print, really just another similarity-substitution scheme;
2. request-return, for various allocation schemes.
3. suspend-resume, the primitives of control,
4. swap in-swap out, (page in-page out), for use in operating systems
5. input-output , especially using coroutine control
6. ying-yang, consider all opposing actions.

Flaws with this approach (to give fair warning) include the limitations on macros (no decisions during expansion) and the copying of too many marred data spaces (explicitly if not implicitly). One may want to test whether a parameter is storable before initializing it and one should not have to state explicitly what happens to fields which are not to be changed (see hd, tl, floor, fract, sign, ... and try defx last x → (if pair(X) then tl X else X) or try defining ⍉ in APL so (1 1 ⍉ M)←E works).

Final disclaimer:

I make no claim that any of the SETL-like statements are legal SETL statements. I have assumed that the reader is familiar with SETL, APL, Algol, PL/1, LISP and Algebra.

GLOSSARY

assertion - a property of a program which is in question

assumption - properties which the data for a program is assumed to have

bit - any predicate which is a field. True and false conform to it.

block - a sequence of statements each followed by a semicolon

commute - two adjacent valid statements commute if reversing their order has no net effect

conformable - a trievable expression E is conformable to a storable expression S whenever S=E is a valid statement

conforms - a value which may be stored into a register and retrieved intact conforms to it

constant - an expression which represents a particular value

constant space - the data space of all constant functions; it is a subspace of any other data space

contains - every function contains its subfunctions

correspondence - a mapping from parameters to phrases

data base - that (smallest) data space of which each data space of a defined function is a subspace

data space - the equivalence class of a function under isomorphism

dead - a data space which is not live

defined function - a function for which value(function) is a field, where value( ) is defined by:

macro X=value(Y);; X=Y;; macro value(X)=Y;;;

equivalent - two statements are equivalent if replacing a valid occurrence of one by the other has no net effect

expandable - a statement is expandable if either it is primitive or every statement generated in processing it is expandable

expression - any phrase at a level lower than statements

extent of a program - assertions posed but not intended field - a retrievable register which is restorable

file - a register which can be assigned a series of conformable values and later spew them out (subject to a transfer function)

finalization - assigning parameters their computed values after a definition

function - a mapping in terms of program variables

generated statement - each statement produced by the expansion of a macro definition

independent - data spaces $D_1,\ldots,D_n$ are mutually independent $(D_1|\ldots|D_n)$ if D_i overlaps D_j implies $i=j$

initialization - evaluation of parameters on entry to a protected definition

intent of a program - some arbitrary collection of assertions about the program and assumptions about the data

isomorphic functions - functions which are subfunctions of each other

live - a data space is live if marring it would have a net effect

macro - any scheme which permits the definition of abbreviated statement forms; the naive (or holy) macro scheme in particular

marred - not safe

mashed by a block - a data space for which every subspace is either marred by the block or dead on entry to it.

naive macro definition - a macro scheme which depends only on similarity and substitution with very few bells and whistles

net effect - a property of modifications to a valid program. The modified version is valid if and only if the modification has no net effect.

overlap - two data spaces overlap if some common subspace is live

parameter - a quantified name in a naive macro definition which is used as a substitution point

pentachotomy law - For any two data spaces A and B, exactly one of the following relations properly holds:

1. $A=B$, some data space
2. $A\subseteq B$, A is a subspace of B, properly if $A \neq B$
3. $A\supseteq B$, A contains subspace B, properly if $A \neq B$
4. $A | B$, A and B are independent, properly if neither $A\subset B$ nor $B\subset A$.
5. $A \between B$, A and B overlap, properly if neither $A\subset B$ nor $B\subset A$.

<u>phrase</u> - any syntactically well formed sequence of names and symbols in a program

<u>program variables</u> - a countable set of names each of which has an associated value at any particular time. The primitive statements VAR=CONSTANT and VAR=VARIABLE are assumed to replace this value with another.

<u>predicate</u> - a boolean function

<u>protected</u> definition - a macro scheme in which the parameters are treated as program variables which may be initialized before entering the definition, and finalized afterward

<u>reached</u> - point L2 can be reached from point L1 if (<u>true</u> at L1 $\supset$ <u>false</u> at L2) is not valid.

<u>register</u> - an expression which is both trievable and storable. It may have strings attached.

<u>restorable</u> expression - a register Q for which Q=X is superfluous following a valid statement X=Q.

<u>restricted</u> - a block is restricted to a data space if every independent data space is safe over that block

<u>retrievable</u> expression - a register Q for which a superfluous trieval Y=Q may follow Q=X, in which case there must be some function such that (Y=function after Q=X; Y=Q;) is a valid assertion

<u>safe</u> - a function is safe between two points L1 and L2 whenever ($\forall$t)((function=t at L1) $\supset$ (function=t at L2)) is a valid assertion

<u>side effects</u> - if the trieval of an expression cannot be superfluous, then the expression has side effects. I.e. <u>value</u>(expression) is not a field

<u>similar</u> - two phrases are similar if some correspondence can be applied to both to make them equal

<u>simultaneous substitution</u> (<u>application</u> of a correspondence) - the scheme of replacing each occurrence of a parameter in a phrase with its corresponding phrase

<u>statement</u> - a primitive form with an a priori definition, or all but the final semicolon of a block which is similar to the first block form of some macro definition. Three examples:

(1) X=3

(2) <u>macro</u> rev ($\forall$COND) STMT;; ($\forall$COND) <u>rev</u> STMT; <u>end</u> <u>rev</u>

(3) <u>rev</u> ($\forall$x $\in$ <u>Dom</u> A W(x))|A(x) = B(x)

storable expression - an expression E for which the statement E=X is expandable (X a program variable).

subfield - the subfunction relation applied to fields. All fields are functions.

subspace - the subfunction relation extended to data spaces.

subfunction of a function - any function which is safe whenever the given function is safe.

superfluous - a valid statement is superfluous if removing it has no net effect. A statement is superfluous at a given point in a program if inserting it at that point has no net effect.

temp - a program variable (or field) which is dead before and dead after a given block

transfer function of a retrievable expression - that function determined by the meaning of 'retrievable'

trievable expression - an expression E for which the statement X=E is expandable

vacuous - a field is vacuous whenever a store into it is superfluous. E.g. value(field) is always vacuous.

valid assertion - an assertion which can be derived from assumptions in the intent of a program.

" data - data which complies with the intended assumptions

" program - a program for which all intended assumptions are valid

" statement - an occurrence of a statement in a valid program.

value - any member of the domain of indestructible manipulable objects of a program; the 'value' operator is defined by:

```
macro value(E) = X;; end no-op;
macro X = value(E); X=E;; end identity;
```

Approximate syntax (simple repetition denoted by (...)*)

```
BLOCK ::= ( STATEMENT;)*
STATEMENT ::= macro BFORM; BLOCK ENDING
      | EXPR=EXPR
BFORM ::= BLOCK                                  (defaulted)
      | (NAME)* ⊢ BFORM ⊣ (NAME)*
ENDING ::=                                       (visual cue is ;;)
      | end(SYMBOL)*
STATEMENT ::+ def SFORM; BFORM ENDING
      | sym def VAR=EXPR; EXPR=EXPR ENDING
      | defx EXPR → EXPR
SFORM ::= STATEMENT                              (defaulted)
      | (NAME)* ⊢ SFORM ⊣ (NAME)*
```

and so on. The alternatives for STATEMENT would best be generated directly from the macro definitions.

References

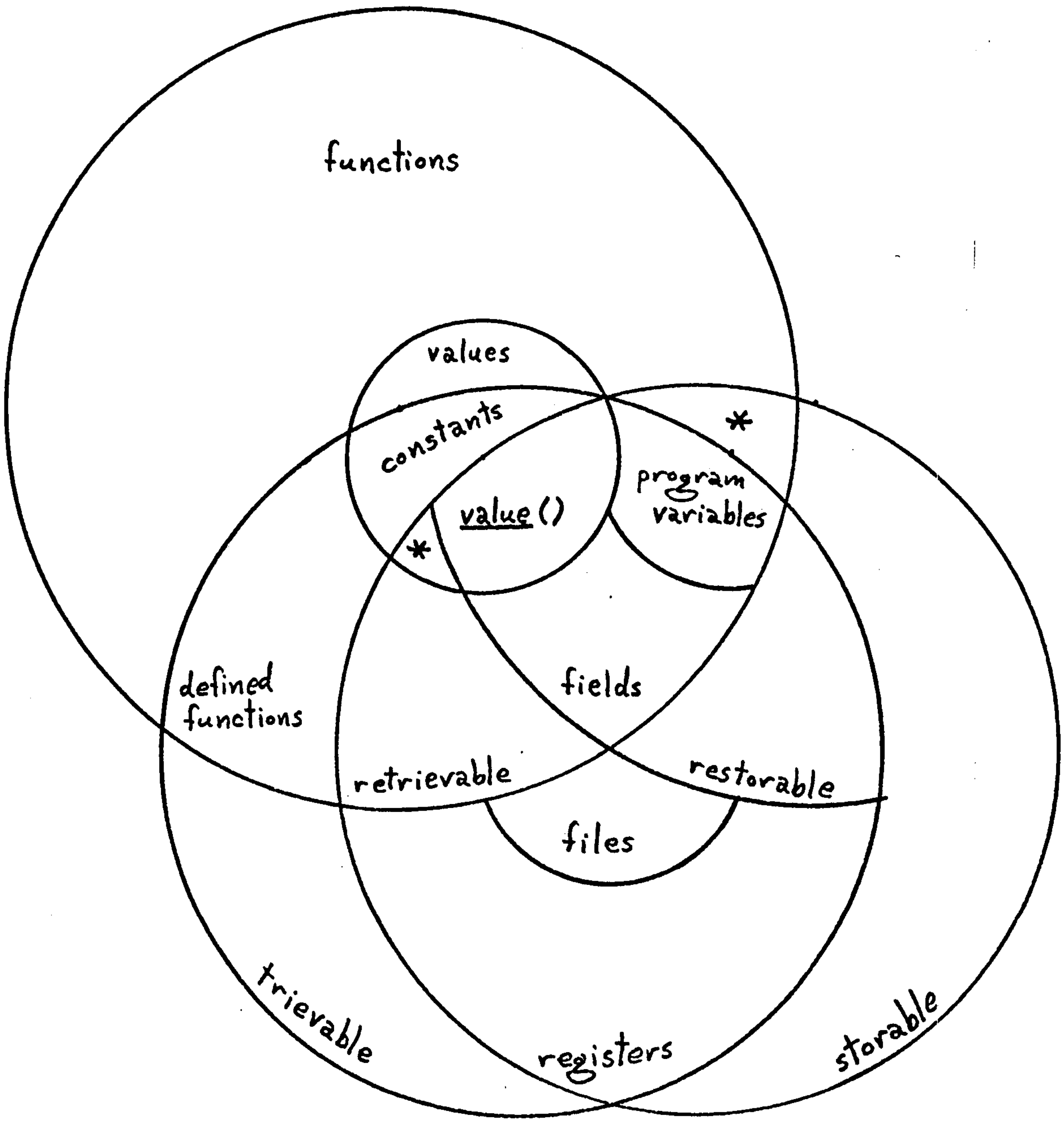
[Dodgson 97] Charles Lutwidge Dodgson, Symbolic Logic, Macmillan & Co., London, 1897; Republished: Berkeley Enterprises, New York, 1955.

[Naur, et al. 60] Peter Naur (editor) "Report on the Algorithmic Language ALGOL 60", Communications of the ACM, vol. 3, no. 5, May 1960.

[Schwartz 71] J. T. Schwartz, "Sinister Calls", SETL Newsletter Number 30, Courant Institute working document, May 1971.

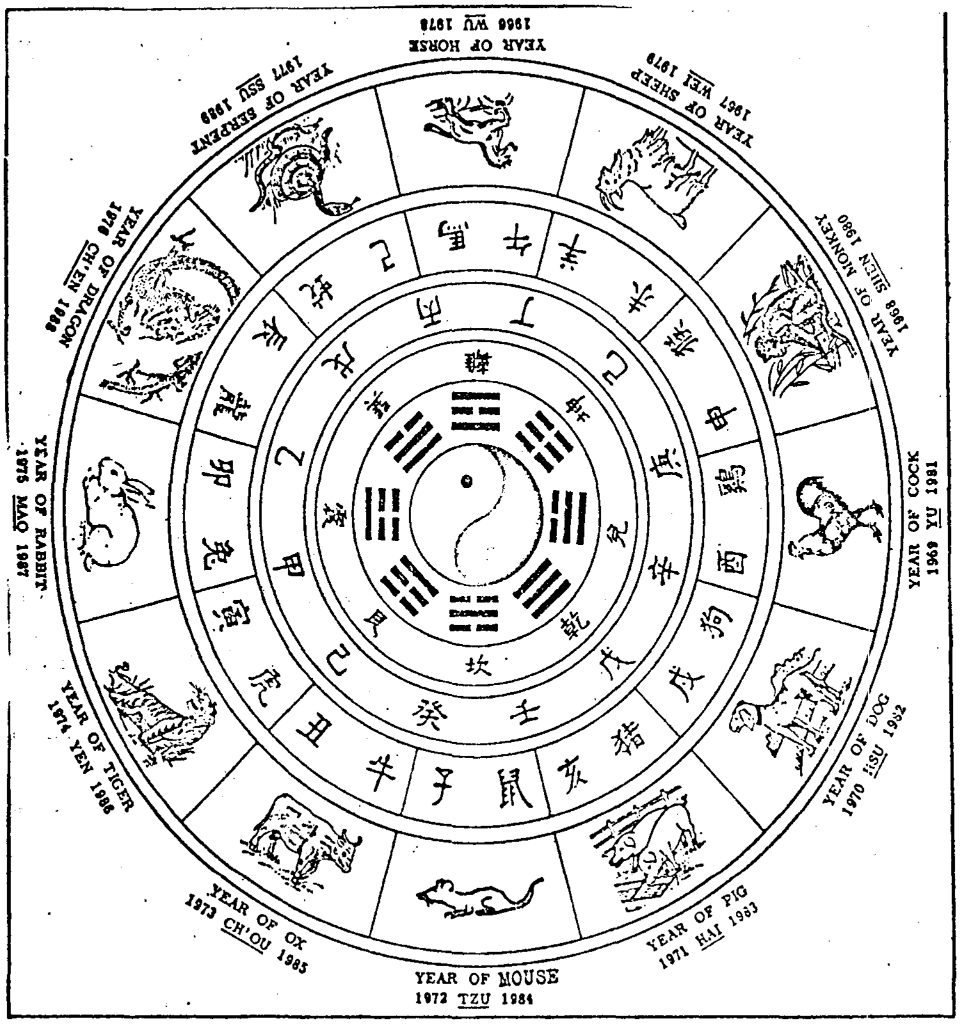
[Irons 70] Edgar T. Irons, "Experience with an Extensible Language," Communications of the ACM, vol. 13, no. 1, January 1970.

The relationships among some of the properties discussed are shown in the following Venn diagram:



*indicates subclasses for which my contrived examples appear to be contrived (3=X, meaning output X to device 3; and any field after its trieval definition has been deleted).

An amusing account of Venn's Method of Diagrams may be found in [Dodgson 97: 174-176] with some historical perspective.



Oriental animal cycle of years, adapted from I Ching, the Book of Changes. Yang and yin symbol at center represents duality in much of Chinese tradition and philosophy.