Robert Bonic
March 1, 1973

HOW TO PROGRAM IF YOU MUST (The SETL Style)

by

Robert Bonic

1. Introduction

These lectures are addressed to mathematics students, but a certain amount of immaturity with computer science will be essential for a complete understanding.

Computer programs are algorithms written in a very stylized format and meant to be understood by a machine. The style in which we will write algorithms below is called SETL (SET Language) and has been devised by Jack Schwartz of the Courant Institute of Mathematical Sciences (N.Y.U.). It is a style that is very comfortable to a mathematician since it uses standard mathematical notations and modes of expression. The few new notions that will be required will be explained as we go along.

For our purposes a machine may be thought of as a black box, inside of which is a math graduate student slave. The student has a finite amount of paper, does not work infinitely fast, and understands the algorithms. She may respond by typing out answers on a piece of paper or printing them out on a TV-like screen.

A dialect of SETL, called SETLB, is currently running on the AEC CDC-6600 at N.Y.U. Future implementations will be orders of magnitude more efficient than the present one.

2. The Data Bank

A possibly multivalued function

$$g: A \rightarrow B$$

may be defined by giving its graph explicitly as a set of ordered pairs. The following notations will be used.

$$g(x) = \begin{cases} y & \text{if } \exists \text{ unique } y \in B \text{ with } \langle x,y\rangle \in g \\ \omega & \text{omega (or undefined) otherwise,} \end{cases}$$

$$g\{x\} = \{y \in B \mid \langle x,y\rangle \in g\},$$

and for a set $C \subset A$

$$g[C] = \{y \in B \mid (\exists\, x \in C \mid \langle x,y\rangle \in g)\}$$

Suppose $g = \{\langle 2,8\rangle,\langle 3,6\rangle,\langle 4,7\rangle,\langle 2,6\rangle,\langle 8,1\rangle\}$. Then

```
g(3) eq 6 , g(4) eq 7,
g(2) eq ω , g(15) eq ω,

g{3} eq {6} , g{2} eq {8,6},
g{15} eq nl (null set),

g[{2,3,4}] eq {8,6,7} ,  g[{8,9,10}] eq {1},
g[{-2,1}] eq nl
```

A data base is a tuple

$$\langle A, f_1, \ldots, f_m, R_1, \ldots, R_n\rangle$$

where A is a finite set, each f_i is a function on A, and each R_j is a binary relation on A. Information that may be accessed includes all sets in the weak topology on A generated by the functions and relations defined on A. A typical such set may have the form

```
{xεA | f(x) gt 5 and (g{x}εK or Q(x) ne 6)}.
```

This information may be augmented by using the operators:

arb(S) and random(S)

which chose any old element from S, and a carefully chosen random element respectively.

Example

```
A = {1 ≤ n ≤ 1000};
male = 0; female = 1; married = 0;
single = 1; divorced = 2;
```

Take the functions

```
name = {<1,"Tom">,<2,"Eve">,...};
age = {<1,26>,<2,11>,...};
sex = {<1,0>,<2,1>,...};
income = {<1,17000>,<2,0>,...};
maritalstatus = {<1,1>,<2,1>,...};
address = {<1,"1234 Wood St., Chicago">,...};
yearsschool = {<1,14>,<2,5>,...};
```

and relations

```
friendsof = {<1,36>,<2,47>,...,
                 <942,419>,<47,2>,...};
debtsto = {<3,67>,<14,45>,...};
```

A great deal of information can be stored in such a data base, the 1970 census for example, and this information can be used in a variety of ways.

Information may be added to the data base by augmenting the base A and updating the functions and relations. For example, if b is given along with data about b, we merely need write

$$b \underline{\text{in}}\ A;\ f_1(b) = c;\ f_2(b) = g; \ldots$$
$$R_1(b) = \{q\};\ R_2(b) = \{r,s,t\}; \ldots \quad .$$

To delete a member from the data base we proceed as follows:

$$c\ \underline{\text{out}}\ A;\ f_1(c) = \omega;\ f_2(c) = \omega; \ldots$$
$$R_1(c) = \omega; \ldots \quad .$$

New functions may be defined using the operations of intersection, union, composition, transitive closure, etc. For example, if R denotes the relation "friendsof" then

$$Q = R \cup (R \circ R)$$

is a new relation. For each $x \in A$ $Q\{x\}$ consists of all elements of A that are either friends of x or friends of friends of x.

Of interest to a sports car salesman might be the set

```
   potentialcustomers = {x ε A |
age(x) gt 18 and age(x) lt 30 and income(x) ge
20000 and sex(x) eq male and
trafficpenaltypoints(x) le 4};
```

The salesman may get a list to give his secretary as follows:

```
(∀xε potentialcustomers)
  print name(x),phoneno(x); end ∀x;
```

or he may print out the following type of form letter:

```
(∀xε potentialcustomers)
print name(x);
print address(x); print;
print "Dear", name(x);
print "As a man with an income of", income(x),
"I am sure you will be interested in our new",
favorite_color(x), "dinkycar".
print "Some weeks ago I ran into your friend",
random friendsof{x}, "and he mentioned that you
were dissatisfied with your", presentcar(x);
  common = clubs{x} ∩ clubs{number("HalBean")};
if common ne nl then print "Perhaps I'll see
you at the", random(common); else print
"Please drop by my office";
print "Personally yours";
print "Hal Bean";
```

3. Simulation

There are many types of situations that can be modeled on a machine. When complete information is known, the model is decisive and empirical evidence can often be used in place of a more difficult analysis of the situation. This will be illustrated with the game of "craps."

A player rolls two dice to establish his point ε {2,3,...,12}. If his point is 2, 3, or 12 he loses. If it is 7 or 11 he wins. If neither of the above is the case, he continues rolling until he rolls a 7 in which case he loses, or his point again in which case he wins. A probablistic analysis shows that the odds are slightly in favor of the house. An empirical estimate of this is given on the right where the game is played one million times and the number of wins is recorded. The program illustrates the use of labels, while and go to statements, as well as the use of a procedure.

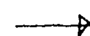
There are two types of procedures in SETL. An example of a functional type of procedure is

definef g(x); return x^2-3x + 4; ●

A procedure of subroutine type does not return a value, it does something and then merely returns. An example might be

define message(x); print x; return; ●

```
craps = {2,3,12}; wins = {7,11};
nowins = 0; nogames = 0;
   die = {1 ≤ ∀n ≤ 6};


definef roll; return random(die) + random(die);

playgame: if nogames gt 1000000 then
   go to finished;;
   no games = nogames + 1; point = roll;

   if point ε craps then go to playgame; else
   if point ε wins then nowins = nowins + 1;
   go to playgame;;

     newroll = roll;
   (while newroll ∉ {7,point})newroll = roll;;
     if newroll eq point then nowins =
     nowins + 1;; go to playgame;;

finished: print nowins,"games won out of",10000
```



The following conversations actually occurred.
M=mathematician   C=computer scientist
MC=mathematician and computer scientist

M to MC: What is a procedure?
MC to M: It's like a lemma!

C to MC: What is a lemma?
MC to C: It's like a procedure!

4. Comparison of Notations

It is quite easy to translate a mathematical algorithm into a SETL program. Translation of some typically mathematical phrases or constructions are shown below:

The programs below are not written from the point of view of efficiency which is a serious issue and will be considered later.

1. The Γ-function
   $\Gamma(1)=1$ and for $n>1$, $\Gamma(n)=n\Gamma(n-1)$

```
definef Γ(n); return if n eq 1 then 1 else
   n*Γ(n-1); ●
```

2. Let $\{S_1,S_2...,S_n\}$ denote the elements of the set of integers A arranged in increasing order.

```
definef incorder(A); S=nl; (∀xεA)
  k=1+#{yεA|yltx}; S(k)=x; end ∀x; ●
```

3. The elements of the above set arranged in decreasing order.

```
definef decorder(A); S=incorder(A);
  (1≤∀n≤#S)S(n)=S(n-#S+1); end ∀n; return S; ●
```

4. Is n a prime number?

```
definef prime(n); return if(1<∃k<n|n//k eq 0)
  then false else true; ●
```

5. The smallest prime greater than a given n

```
definef primeafter(n); k=n+1;
  (while not prime(k))k=k+1;end while;return k; ●
```

6. A list of the first 500 primes.

```
list(1)=1; (while #list lt 500)
  list(#list+1)=primeafter(list(#list));end while
```

7. The set of primes smaller than 500

```
primes={1<p<500|prime(k)};
```

8. The above set in increasing order

```
incorder(primes);
```

9. The greatest common denominator of two positive integers m and n.

```
definef gcd(m,n); return if m eq n
  then n else gcd(|m-n|,min{m,n}); ●
```

10. Single valued function

```
definef function(f); return type(f) eq set
  and (∀xεf|pair(x)) and (∀xεf,yεf|
  x(1) eq y(1) implies x(2) eq y(2));●
```

11. Domain of function

```
definef domain(f); return {x(1), xεf}; ●
```

12. Inverse of function

```
definef inverse(f); return {<x(2),x(1)>, xεf}; ●
```

13. Image of function

```
definef image(f); return domain(inverse(f)); ●
```

14. Injectivity

```
definef injective(f); return(∀xε domain(f),
  y ε domain(f)|x ne y implies f(x) ne f(y)); ●
```

15. Composition of functions(or relations)

```
definef R o S; return {<x(1),y(2)>, xεR, yεS |
  x(2) eq y(1)}; ●
```

16. The nth power of a function

```
definef power(R,n); return if n eq 1 then R
  else if n eq 2 then R o R else
  R o power(R,n-1); ●
```

17. The transitive closure R* of a relation R

```
definef star(R); T=R; n=2;
  (while power(R,n) ne nl) n=n+1;
  T=T ∪ power(R,n);; return T; ●
```

18. The set of all functions from A to B

```
definef fncts(A,B); return
  {f ε powset(A x B)|domain(f) eq A
   and function (A)}; ●
```

19. Permutations of a set A

```
definef permutations(A); return {f ε fncts(A,A)|
  injective(f)}; ●
```

20. Given an integer n>1, if it is even divide it by 2,and if it is odd multiply by 3 and add 1. It is not known if iteration of this always leads to 1. For n=7 we have 7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1

```
definef getstone(n); return if n eq 1 then true
  else if even(n) then getstone(n/2) else
  getstone(3*n+1); ●

definef even(n); return 2*(n/2) eq n; ●
```

21. The pair of prime twins greater than n.

```
definef nextprimepair(n); k=n+1;
  (while not (prime(k) and prime(k+2)))
  k=k+2; end while; print <k,k+2>; ●
```

22. The center of a group g whose multiplication table is m.

```
center = {yεg|(∀xεg|m(x,y) eq m(y,x))};
```

<u>Most programming languages</u>, e.g., FORTRAN, BASIC, APL, ALGOL, LISP, PL/1, COBOL, have evolved up from the machine and programming manuals often read like old fashioned text books on tensor analysis. SETL descends into computer science from mathematics, and is essentially a coordinate-free approach to programming. Problems of efficiency, which often amount to something like choosing a suitable coordinate system, can be handled automatically. The essential contribution of SETL is the introduction of abstract objects in place of artificial coordinate representations of them.

There seems to be a difference of opinion concerning the SETL style of programming as compared to the current one. The humorous excerpt below is taken from a 1971 review of a 300-page SETL manuscript written by an "expert" on programming languages.

"In summary then, SETL is at worst just a collection of strange notations and devices, and at best it is "just another programming language." Compared with the elegance and clean design of APL, SETL fails to attract the mathematical mind. It does not use the mathematician' symbols, his notation, his precedences, or his identities. Its mass of petty detail is no smaller than that of other languages. The algorithms presented are little more than transliterations of what would be written in ALGOL or APL. Yet the idea of a set as a datatype (or data structure) and the partially-fulfilled idea of specifying operations on all the elements of a set are very powerful notions and are good candidates for incorporation in some existing programming languages."

5. The Labyrinth

The following is from Algorithms and Automatic Computing Machines by B.A. Trakhtenbrot. The setting is a symmetric relation R over a set of nodes. Two nodes are given and the problem is to construct a path between them (if one exists).

In order to construct such an algorithm, we prescribe a special method of searching. At each step of the search we can separate the corridors into three classes: those through which Theseus has never passed (we shall call these *green* corridors), those through which he has passed once (*yellow*), and those through which he has passed twice (*red*). Furthermore, from any junction Theseus may move to an adjacent junction in one of two ways:

1. *Unwinding the thread.* Theseus moves along any green corridor to an adjacent junction, unwinding Ariadne's thread as he goes; this corridor is then considered yellow.
2. *Rewinding the thread.* Theseus returns along a yellow corridor to an adjacent junction, rewinding Ariadne's thread as he goes; this corridor is then considered red.

Note that Theseus is not allowed to go through a red corridor. We assume that Theseus makes some mark by which he can later distinguish a green corridor from a red one. He can distinguish the yellow corridors because they have Ariadne's thread stretched along them. The choice of each move depends upon the conditions which Theseus finds at the junction where he happens to be. These conditions will be one or more of the following:

1. *Minotaur.* The Minotaur is discovered at the given junction.
2. *Loop.* Ariadne's thread already passes through the given junction; in other words, there are at least two other yellow corridors leading from the junction.
3. *Green.* There is at least one green corridor leading from the junction.
4. *Ariadne.* Ariadne is at the given junction.
5. *Fifth case.* None of the above conditions prevails.

Our search method may now be described by means of the following table.

Condition	*Move*
1. Minotaur	Stop
2. Loop	Rewind the thread
3. Green	Unwind the thread
4. Ariadne	Stop
5. Fifth case	Rewind the thread

```
define labyrinth(ariadne,minotaur,R);

green=R; yellow=nl; red=nl; marked=nl;
  /*at each step after the first we have
    just traversed a path <a,b>.  We have
    the following cases*/

**case1 = b eq minotaur**
**case2 = b   marked**
**case3 = {x ε green | x(1) eq b} ne nl**
**case4 = b eq ariadne**
  /*the possible actions are*/
**stop = go to done**
**rewind = <a,b> in red; <b,a> in red;
  <a,b> out yellow; <b,a> out yellow;
  <a,b> = <b,a>; go to nextmove**
**unwind = x = arb {y ε green|y(1) eq b};
  b in marked; <a,b> = x;
  <a,b> in yellow; <b,a> in yellow;
  <a,b> out green; <b,a> out green;
  go to nextmove**
 x=arb{y ε R|y(1) eq ariadne};
 <a,b> = x; <a,b> in yellow; <b,a> in yellow;
 <a,b> out green; <b,a> out green;
nextmove: if case 1 then stop;;
          if case2 then rewind;;
          if case3 then unwind;;
          if case4 then stop;;
               rewind;
done: if b eq ariadne then print "no path exists"
  else print "The solution is", yellow; end if;
```

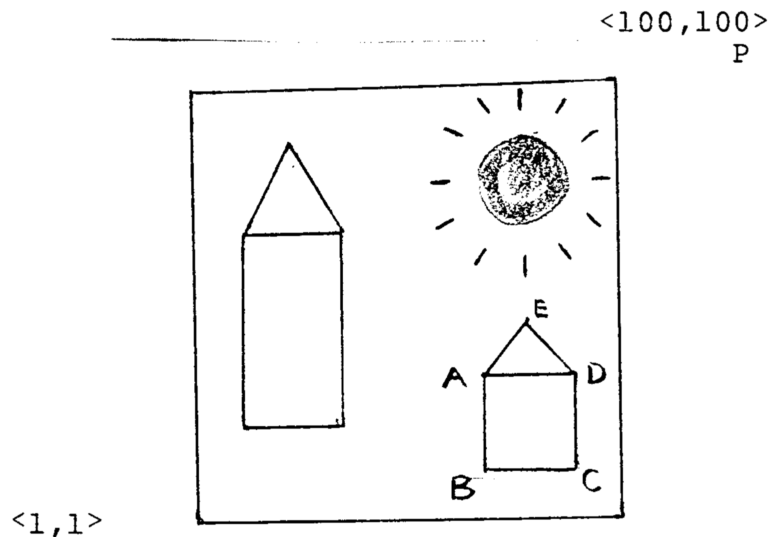
6. Finite Groups

We will write an algorithm to check the truth of the Feit-Thompson theorem for all groups of even order up to one million.

```
definef group(G); <e,g,m,i> = G; return
  type g eq set and type m eq set and
  type i eq set and
  e ε g and i[g] eq g and m[g,g] eq g and
  (∀xεg | m(x,e) eq x and m(e,x) eq x) and
  (∀xεg | m(x,i(x)) eq e and m(i(x),x) eq e) and
  (∀x,y,zεg| m(x,m(y,z)) eq m(m(x,y),z));●

definef subgroup(h,G); <e,g,m,i> = G;
  return h⊂g and group(<e,h,m,i>); ●

definef normalsubgroup(h,G); return subgroup(h,G)
  and (∀xεh, yεG(2)|m(i(y),m(x,y))εh); ●

definef generatedsubgroup(A,G);<e,g,m,i>=G;
  G=Awithe; n=0;(while n lt #B) n=#B;
  (∀x,yεB) m(x,i(y)) in B; end ∀x;
  end while; return B; ●

definef generatednormalsubgroup(A,G);
  <e,g,m,i>=G; return[⋂: B ⊂ g |
  A ⊂ B and normalsubgroup(B,G)]B; ●

definef commutator(G);<e,g,m,i>=G;
  A={m(x,m(y,m(i(x),i(y)))),xεg,yεg};
  return generatednormalsubgroup(A,G); ●
```

```
definef quotient(G,R); <e,g,m,i> = G;
  cosets = nl; (while g ne nl) x = arb g;
  coset = {yεg|m(x,i(y)|εh};
  coset in cosets; coset outof g; end while;
     M=nl; I=nl;
  (∀aεcosets, bεcosets) M(a,b)=m[a,b];
  I(a) = i[a]; end ∀a; return
  <h,cosets,M,I>; ●

definef solvable(G); (while G ne
  commutator(G)) G=quotient(G,
  commutator(G)); end while; return
  #G(2) eq 1; ●

A = {1≤n≤1000000}; g = permutations(A);
i = {<x,inverse(x)>, xεg};
m = {<x,y,x∘y>, xεg,yεg};
e = {<a,a>, aεA};
P = <e,g,m,i>;

if ∃ B ⊂ g | subgroup(B,P) and even (#B)
  and not solvable (<e,B,m,i>) then
  print "Feit-Thompson Theorem is wrong";
  else print "OK through 1000000";;
```

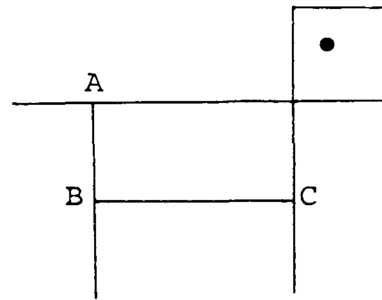
7. Drawing Pictures




```
A = <70,30>; B = <70,10>; C = <90,10>;
D = <90,30>; E = <80,40>;

house = brokenline(<A,B,C,D,E,A,D>);
sun = {<x,z>εP | (x-80)^2+(y-80)^2 le 100};
ray = {<x,y>εP | y eq 80 and 95 le x le 100};
  rays=ray;(1≤∀n≤11)rotated(ray,<80,80>,n*30)
          into rays; end ∀n;
tower = {<x,2y>,<x,y>ε translated(house,<-60,0>)};
  picture = house ∪ sun ∪ rays ∪ tower;
      draw(picture);
```

The code to implement the above is given on the right. A three dimensional version could be used to draw a picture in one perspective, rotate the axes, and then view it from a different perspective.

```
P = {<m,n>, 1≤m≤100, 1≤n≤100};
  star = "*"; blank = " ";
  blankline = 100*blank;
  lines = nl; (1≤∀n≤100)lines(n)=blankline;end ∀n

define draw(set); (∀<x,y> ε set)
  lines(x)(y) = star;;
  (100≥n≥1) print lines(n);; return; ●

define translated(set,point); <a,b> = point;
  return {<x+a,y+b>,<x,y> ε set}; ●

definef rotated(set,point,degrees); <a,b>=point;
  θ = degrees/6;return{<a+(x-a)cos(θ)+(y-b)sin(θ)
    b+(x-a)sin(θ)-(y-b)cos(θ)>,<x,y>εset}; ●

definef brokenline(t);return
  [∪: 1≤∀n≤#t-1] segment(t(n),t(n+1)); ●

definef segment(p,q); <a,b>=p; <c,d>=q;
  k = sqrt((a-c)^2+(b-d)^2); return
  {<a+(c-a)n/k,b+(d-b)n/k>, 1≤n≤k}; ●

definef cos(x); return 1-x^2/2 + x^4/24; ●

definef sin(x); return x-x^3/6 + x^5/120; ●
```

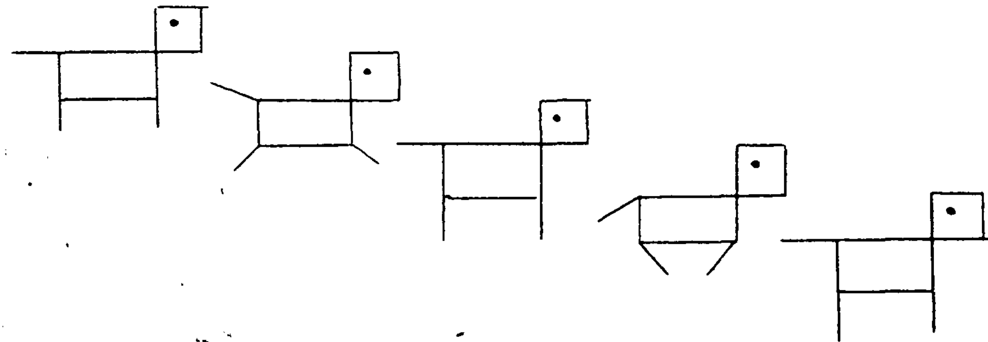


Assuming that the draw routine prints on 35mm film one can use the routines above to make movies. Suppose the sets

tail, body, fleg, bleg

are given along with the pivots A, B, C and dog = tail ∪ body ∪ fleg ∪ bleg. Then (1 ≤ ∀n ≤ 50) draw (dogs(n));; will produce the movie on the right if the dogs routine is defined as follows:

```
definef dogs(n); if odd(n) then return
  trans(dog,<n,o>);;
  θ = if divides(4,n) then -1 else 1;
  newdog = body ∪
  rot(fleg,C,θ*45)∪
  rot(bleg,B,-θ*45)∪
  rot(tail,A,θ*30);
return trans(newdog,<n,o>); ●
```



8. Forestry

A powerful aspect of SETL is that it allows procedure names to be elements of sets or components of tuples. This use will be illustrated below.

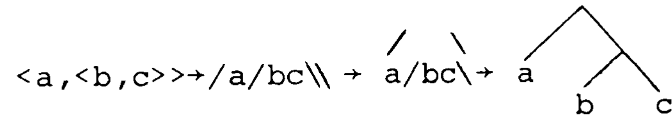
A binary tree is either an atom or a pair of trees. The atoms in the tree are called its leaves. Two trees will be called similar if they have the same structure when the leaves are ignored. A forest is defined as a set of trees. A forest may be enlarged by adding to it all trees of the form T(2) when T(1) and T already belong to the forest. Iterating this procedure will eventually lead to a fully grown forest. Suppose F is a forest. Then the program

```
(while #F lt # enlarged(F)) F=enlarged(F);;
```

expands F until it produces no new trees. The growth can be spedup considerably using features where a feature (of trees) is a function defined on all trees. Using the definition on the right of enlarging the forest, F, with respect to a set of features, feats, gives the more efficient growth

```
(while #F lt # enlargedwrt(feats, F))
   F = enlargedwrt(feats, F);;
```

How efficient the growth is dependent on how quickly the features can be evaluated. If f is a feature and the test f(S) eq f(T) can be computed much more quickly than the test S sim T then the feature may be useful.




```
definef tree(T); return if atom T then true else
  pair T and tree(T(1)) and tree(T(2)); ●

definef S sim T; return if atom S then atom T else
  S(1) sim T(1) and S(2) sim T(2); ●

definef forest(F); return (∀xεF|tree(x)); ●

definef enlarged(F); return F∪
  {T(2), TεF|(∃SεF|S sim T(1))}; ●

definef enlargedwrt(features,forest); new=nl;
  (∀Sεforest)
  (∀Tε{Rεforest|(∀fεfeatures|f(S)simf(T(1))})
     if SsimT(1) then T(2) in new; end ∀T;
  end ∀S; return forest + new; ●
```

Typical features might be

```
definef size(T); return if atom T then 1 else
  size(T(1))+size(T(2)); ●
definef lean(T); return if atom T then 0 else
  if size(T(1)) lt size (T(2)) then -1 else if
  size (T(1)) eq size (T(2)) then 0 else 1; ●
```

9. Real Reals

This section follows the spirit of Errett Bishop's *Foundations of Constructive Analysis* (pp. 18, 19, 26).

Let Z^+ devote the positive integers.

A procedure x defined $\forall n \varepsilon Z^+$ is said to be *real* if for every $k \varepsilon Z^+$ there is an $A(k) \varepsilon Z^+$ satisfying

$$|x(n)-x(m)| \leq 1/k \quad \forall\, m,n \geq A(k).$$

Two reals x and y are *equal* (x=y) if for every $k \varepsilon Z^+$ there is an $B(k) \varepsilon Z^+$ satisfying

$$|x(n)-y(n)| \leq 1/k \quad \forall\, n \geq B(k).$$

A real x is *positive* ($x > 0$) if for some $k \varepsilon Z^+$ there is a $C(k) \varepsilon Z^+$ such that

$$x(n) \geq 1/k \qquad \forall\, n \geq C(k)$$

A real x is *nonnegative* ($x \geq 0$) if for each $k \varepsilon Z^+$ there is a $D(k) \varepsilon Z^+$ such that

$$x(n) \geq -1/k \qquad \forall\, n \geq D(k)$$

It can occur that a given real number x is known to satisfy $x \geq 0$, but that neither $x > 0$ nor $x = 0$ is known to be true. More striking is the fact that there is no general method (nor will there ever be one) for deducing that $x > 0$ or $x = 0$ given that $x \geq 0$. Examples are given on the right.

```
definef a(n); return 0; ●
definef b(n); return 1/n; ●
definef c(n); return n/(n+1); ●

definef d(n); return
  if (1≤∃k≤2n | prime(k) and prime(2n-k))
  then 0 else 1; ●

definef e(n); return[+:1≤k≤n]d(k)/10^k; ●
```

The procedures a,b,c, and e are real numbers. On the other hand the procedure d is not known to be real. It generates a sequence of zeros and ones, but the Canchy criterion cannot be verified since it would involve finding a proof of Goldbach's conjecture.

The number e satisfies $e \geq 0$ since we have that $e(n) \geq 0$ for all n. To show that $e = 0$ would amount to showing that Goldbach's conjecture were true and a proof that $e > 0$ would provide a counter example to the conjecture. Hence, from $e \geq 0$ we cannot conclude either $e > 0$ or $e = 0$, as is done in the formal setting. In fact, many basic results in formal analysis are false in the constructive sense.

10. References and Remarks

1. Schwartz, J. *Abstract Algorithms and a Set-Theoretic Language for their Expression* N.Y.U. Notes (1972).

This is the basic reference work on SETL. It includes a discussion of programming principles, details concerning the design of the language, and a variety of algorithms written up in SETL.

2. SETL-Newsletters 1-100+. N.Y.U. Dittoed Notes.

A miscellaneous collection of papers mostly concerned with implementation problems, but some have an independent theoretical importance. Variously authored.

3. Mullish, H. *SETLB-Manual*, N.Y.U. Notes (1973).

A users' manual for SETLB with all the gory details explained and illustrated.

4. Cocke, J. and Schwartz, J. *Programming Language and Their Compilers*, N.Y.U. Notes (1972).

A comprehensive discussion of the problems involved in getting a machine to understand a language. Parsing algorithms and optimization techniques are discussed at great length.

5. Schwartz, J. *Sinister Calls*, SETL Newsletter #30. Krutar, R. *An Algebra of Assignment*, SETL Newsletter #50.

These two papers are concerned with a quite different approach to programming. It will be illustrated with a few examples.

Suppose x = <2,8,6,4>.

Then, setting y=x(3) gives y the value 6.

By contrast, setting x(3)=45 will change the value of x which then becomes <2,8,45,4>.

Rewriting x(3)=45 as x(3) *eq* 45 = *true*, opens the door to a new style of programming. For example, instead of writing an algorithm to solve the pair of equations

2x + 3y = 7 & 4x - 5y = 3,

one would merely write

2x + 3y *eq* 7 *and* 4x - 5y *eq* 3 = *true*;

The compiler would then fiddle away until it established the truth of this statement by finding x = 2 and y = 1.

The general principle here is quite simple and programs could be written in the following form:

Whatever you wish to be true = *true*.

A programming language developed along these lines would lead to a descriptive, rather than algorithmic, style of programming, and this could be very powerful. A descriptive sort of a sequence x(m) has the form

$$(1 \leq \forall k < \#x \mid x(k)\ \underline{\text{lt}}\ x(k+1)) = \underline{\text{true}}.$$

Eventually one might be able to write the ultimate program

Let there be light = *true*.

6. Bishop, E. *Mathematics as a Numerical Language*, *How to Compile Certain Formal Systems*, and *A General Language*. (Preprints-La Jolla)

In these three papers as well as his book (cited earlier) Bishop speaks about writing a compiler for mathematics. The input to the compiler would be a mathematical proof of the existence of some object, and the output would be a program that constructs the object. This would of course only be feasible in the constructive framework where "proof" refers to a constructively valid argument.

7. Westin, A. and Baker, M. et al., *Databanks in a Free Society*.

From the book jacket: "During the past two decades, as the large-scale record systems of most government agencies and private organizations have been computerized, the American public has become concerned that something fundamentally dangerous might be happening."

8. Bailey, A. *From Intellect to Intuition*, Lucis Publ. Co., 1932.

A philosophical work which is pertinent to the design of software for the increasingly popular notion of the teaching machine. Chapter 2, The Purpose of Education, leads off with the following quote from H.A. Overstreet:

"...education is undergoing important transformations. From a relatively external process of pouring in facts, it is increasingly becoming a process of evoking the deeper, generative possibilities that lie within the individual."