SETL Newsletter Number 30 May 7, 1971
Sinister calls J. T. Schwartz

This note will suggest a method of allowing rather general "left-hand-sides" in assignment statements. The forms allowed will include quite general indexing operations, conditional expressions, and calls to programmer defined functions. Example: using the scheme to be suggested, it will be possible first to program a function called last, which when called in the normal way returns the last element of an n-tuple; and then to use this function to the left of an equal sign, writing

```
last tupl = x
```

to change the final (and only the final) component of tupl. Then, for example, by executing

```
   x = < <1,2,3>,4,5>;
last hd x = 10;
   print x;
```

one will obtain

```
< <1,2,10>,4,5>,
```

etc.

The scheme to be proposed is generally applicable to "procedural" programming languages. It is based upon certain general algebraic observations concerning "retrieval and assignment" pairs of functions, which will be presented below. The scheme avoids the explicit transmission of pointers, and the complications which such transmission may lead to. The mechanism seems to me to constitute an appropriate systematic generalization to the left-hand sides of assignments of the standard subroutine-function linkage concepts applying to the right-hand side of an assignment; and to be as basic as (though not necessarily as generally useful as) these latter concepts. For this reason, I shall call the interprocedural linkages to be suggested "sinister calls", and call the conventional call conventions to be used on the right-hand side of an assignment "dexter calls".

To stress the general nature of the considerations involved, I shall begin with a few very general remarks. SETL belongs to the class of "procedural" languages, that is, to those languages which in one or another manner represent the algebra of transformations on a universe of stored data objects. The semantic operations most fundamental to such languages are:

i - access to a designated portion of a stored object.

ii - modification of a designated portion of a stored object.

iii - combination of transformations by successive application.

iv - choice of transformation to be applied depending on a predicate of particular stored data objects.

v - combination of two or more stored objects in some "algebraic" fashion, useful generally or for some specifically intended application area, the outcome of this combination process being some "output" object.

vi - repetitive application of a transformation until a certain condition is established.

With these primary operations various secondary operations may be listed; of these possibilities we shall mention only

vii - application of a given transformation to all the subobjects of a given object (iteration-over-parts).

All the considerations in the present note will center around operations i and ii. We call these operations retrieval and storage operations respectively.

What gives operations this character? First consider retrieval; and let op stand for an operator. If this is to be a retrieval operator, it must in the first place be free of "side effects". That is, if op a is called once, and then again, the same value should be returned both times.

Next, there should exist a storage subroutine corresponding to the retrieval operator. This would be some subroutine, call it

(1) opstore (a,x);

such that after executing (1) we can be sure that an immediately following call to op a will return the value x. Moreover, two successive calls to (1) should have the same effect as a single call; and, more precisely, whenever the value of op a is already x, opstore(a,x) should be an identity transformation. This last requirement is rather sharp, and pins opstore (a,x) closely to op a; moreover, it implies that the properly "to be a retrieval operator" is not possessed by all operators op. We shall in what follows speak as if a retrieval operator op determines its storage operator uniquely.

Note the following simple retrieval and storage operations in SETL: (some standing in the retrieval-storage relationship only in most, but not in degenerate, cases)

1. retrieval: hd a
   storage: a = if tl a eq Λ then x else <x, tl a>;
2. retrieval: tl a
   storage: a = if tl a eq Λ then a else <hd a, x>;
3. retrieval: a(i)
   storage: a = a lesf i with <i,x>;
4. retrieval: a{i}
   storage: a = a lesf i u {<i,y>, y ∈ x};

and among various other more complex examples which might be cited

5. retrieval: (for a sequence a)
   {<k-n+1, a(k)>, m≤k≤mm}
   (which might be written a[m:mm]

storage:

a = {pairs ∈ a | n (hd pair) ge m and (hd pair) le n}
u {<k+n-1, x(k)>, 1≤k≤#x}

(which, precisely because it is the storage operation associated with the above retrieval operation, might be written as

a [m:mm] = x;)

A formal notion of independent storage operators helps us pin down the intuitive idea that a storage operator should change no more than is required by its relation to a particular retrieval operator.

Let op and op' be two retrieval operators; anticipating the syntactic style to be developed we shall write calls on their associated storage routines as

(2) op a = x; and (3) op' a = x;

respectively. We call op and op' independent retrieval operators if in the succession of calls

(4) y = op a; op' a = x; z = op a;

the variables y and z necessarily receive the same value; provided that the same is true when op' and op exchange roles.

The following observation is now crucial: the composite operator op op' is a retrieval operator; its storage routine is defined by the code sequence

```
 i.  t = op' a; /* where 't' is a 'compiler temporary' */
ii.  op t = x;
iii. op' a = t;
```

the effect of this may appropriately be represented by writing

(5) $\underline{op}\ \underline{op}'\ a = t;$

Proof: (which please ponder)

If the retrieval($\underline{op}\ \underline{op}'\ a$) already gives x, then after (i) the retrieval ($\underline{op}\ t$) gives x also. Hence the operation (ii) may be omitted. But therefore the retrieval ($\underline{op}'\ a$) gives the value t; and hence (iii) may be omitted. Therefore the whole sequence (i,ii,iii) amounts merely to a store into a compiler-generated temporary; and is therefore equivalent to a no-operation. Q.E.D.

The construction (i,ii,iii) may evidently be iterated, with the following result. Let $\underline{op}$, $\underline{op}'$, ..., $\underline{op}^{(n)}$ be a sequence of retrieval operators. Then their product $\underline{op}$, $\underline{op}'$, ..., $\underline{op}^{(n)}$ is also a retrieval operator, and the assignment operation:

(6) $\underline{op}\ \underline{op}'\ \ldots\ \underline{op}^{(n)}\ a = x$

expands naturally as

(7)
$$\begin{aligned} t^{(n)} &= \underline{op}^{(n)}\ a; \\ t^{(n-1)} &= \underline{op}^{(n-1)} t^{(n)}; \\ &\ldots \\ t' &= \underline{op}'t''; \\ \underline{op}\ t' &= x; \\ \underline{op}'t'' &= t'; \\ &\ldots \\ op^{(n)} a &= t^{(n)}; \end{aligned}$$

Note that if $x = \underline{op}\ \ldots\ \underline{op}^{(n)}\ a$, then the middle and all following lines in (7) are equivalent to no-ops. We shall speak of the code sequence (7) implied by the statement (6) as unraveling (6). It is worth noting one additional feature of the expansion (7). Suppose that $\hat{\underline{op}}^{(n)}$ is a retrieval operator independent of

op, and that $\hat{op}$, ..., $\hat{op}^{(n-1)}$ is any sequence of retrieval operators. Then the products

(8) $\hat{op}\ \hat{op}'\ \ldots\ \hat{op}^{(n-1)}\hat{op}^{(n)}$ and $op\ op'\ \ldots\ op^{(n)}$

are also independent. Indeed, in this case, the only assignment to a in (7), that is, the last line in (7), has no effect on the value of $\hat{op}^{(n)}a$, and thus none on the value of

$$\hat{op}\ \ldots\ \hat{op}^{(n)}a.$$

More generally, if $\hat{op}^{(j)}$ is independent of $op^{(j)}$, then

$\hat{op}\ \ldots\ \hat{op}^{(j-1)}\hat{op}^{(j)}op^{(j+1)}\ \ldots\ op^{(n)}$ and

$\hat{op}\ \ldots\ \hat{op}^{(j-1)}op^{(j)}\ \ldots\ op^{(n)}$

are independent. This conclusion may be proved by "algebraic" reasoning from the previous special case, and may also be demonstrated directly.

Our conclusions up to this point may be summarized in the following

Statement A: The set of retrieval operators associated with the set of stored objects of a procedural programming language forms a semigroup, associated with the language in a natural way. The basic retrievals in SETL are the operations

$$f(a, \ldots, a_n),\ f\{a, \ldots, a_n\},\ \underline{hd}\ f,\ \underline{tl}\ f.$$

The operators presently written as <*zt n> and <* z n>, which retrieve either the n-th component of an n-tuple or the hd of this component ought also to be mentioned. (A storage and a retrieval sequence corresponding to this operator are given explicitly below.) In this note we shall use the variant notation tupl[n] for <*z n> tupl, and tupl[-n] for <* zt n> tupl. Note then that since the product of retrieval operators is a retrieval operator, it is reasonable to allow notations obtained

from the preceding notations by "compounding", as for example in

$$f(a,b)\{c,d\}[n](x)[-n];$$

Such forms may also be allowed freely on the left-hand side of an assignment, and will have a significance deducible from the discussion above.

Note that there exist the following algebraic relations between these retrieval operators:

$$f\{a,b\}\{c,d\} \equiv f\{a,b,c,d\};$$
$$f\{a,b\}(c) \equiv f(a,b,c);$$
$$f[-n][-m] \equiv f[-n-m+1];$$
$$f[-n][m] \equiv f[-n-m+1].$$

It is reasonable to allow

$$f[m_1, \ldots, m_n]$$

to represent the composite operator

$$f[m_1][m_2], \ldots, [m_n].$$

We may also consider retrieval operators $\underline{op}(p_1, \ldots, p_n)$ depending on several parameters, and the storage subroutines $opstore(p_1, \ldots, p_n, a, x)$ associated with them. It might be better to write the retrieval as $op(p_1, \ldots, p_n, a)$; showing the "retrieval target" a along with all the other parameters. Then we require that if the function $op(p_1, \ldots, p_n, a)$ has the value x, then the subroutine $opstore(p_1, \ldots, p_n, a, x)$ acts as an identity operator. In this case, a call to opstore may be written in the form

$$(9) \quad op(p_1, \ldots, p_n, a) = x;$$

though other syntactic forms (both for the left- and the right-hand sides) may be preferred in particular cases.

A SETL example of a usage like (9) is implicit in the fifth retrieval-storage pair discussed above. Another case of interest has to do with the components of n-tuples. Here we can define associated storage and retrieval operators:

retrieval: definef comp(m,tupl); return if m eq 1 then hd tupl else if n pair tupl then tupl else comp(m-1,tupl); end comp;
storage: define compstore(m,tupl,x); if m eq 1 then hd tupl=x else if n pair tupl then tupl=x else compstore(m-1,tupl,x); end compstore;

The linkage conventions which are to be discussed below will make the explicit appearance of the second of the above routines unnecessary, deducing its features automatically from the corresponding features of the function comp.

As with simple retrieval operators, so in the case of retrieval operators with parameters the composition of two retrieval operators is a retrieval operator. The natural interpretation of

(10) op(p,op'(q,a)) = x

is

(11) i'. t=op'(q,a);
ii'. op(p,t)=x;
iii'. op'(q,a) =t;

cf. (i,ii,iii) and (7). Note in particular that if

(12) x = op(p,op'(q,a)),

then after (11.i') is executed op(p,t) has the value x and op'(q,a) has the value t; hence (11.ii') and (11.iii') may be omitted. This is to say that the whole sequence (11) reduces to a no-operation.

Note also that the unraveling (11) of (10) treats all the arguments of op and op' on an equal footing, making it unnecessary to distinguish between a single "principal argument" and a remaining set of "parameters". The following observation emphasizes this point: the routine

```
(13)  definef  select(f,g,j); return if j gt 0 then f(j) else
                  g(-j); end select;
```

is a retrieval function. The associated storage function may be written simply as

```
(14)  define  selectstore(f,g,j,x); if j gt 0 then f(j)=x else
                  g(-j) = x; end selectstore;
```

Having noticed this fact, we may for example assign meaning in a standard way to the statement

```
(15)  select(f,select(g,h,i),j) = x.
```

Note that the value of

```
select(f,select(g,h,i),j)
```

may be expressed as follows:

```
if j gt 0 then f(j) else if i gt 0 then g(i)(-j)
                  else h(-i)(-j).
```

The reader may verify that if (15) is unraveled in accordance with the convention that has been suggested it leads to a sequence of operations equivalent to the statement

```
if j gt 0 then f(j)=x; else if i gt 0 then g(i)(-j)=x; else
                  h(-i)(-j) = x;;.
```

Statements of the form (15) may in certain cases be ambiguous; a fuller discussion of this problem, together with a suggested method of removing the ambiguity, will be found below.

A number of other useful generalizations can be made. Suppose that op is a retrieval operator, that fg is a one-to-one mapping, and that gf is a one-to-one mapping inverse to fg. Then fg op is a retrieval operator also; the storage operation that corresponds to it is simply

(16) op a = gf x;

The use of 'recodings' in this way may also be related to our general sinister expansion procedures. Suppose that fg and gf are related in the manner just described. Then x = fg a may be regarded as a retrieval whose corresponding storage operator is a = gf x; . The formal connection between these two operators required by the general definition of the retrieval-storage relationship is certainly satisfied. Thus we may write a = gf x as

```
fg a = x;
```

This special 'storage operation' differs from others in that it changes a completely, not merely partially. That is, in any code sequence

```
   a = expn;
fg a = x;
```

the first line is superfluous, since a is dead on entry to the second line. Suppose that we are working with a compiler that can detect this fact. Then, in our normal expansion

```
   t = op a;
fg t = x;
op a = t;
```

of

```
fg op a = x;
```

the first line will be eliminated, leaving precisely (16). Thus 'recoding' operators may be treated, in an entirely satisfactory fashion, as retrieval operators of a special kind.

At the level of hardware, the fact that storage and retrieval may involve extensive recoding; the increasingly complex transformations, reshufflings, etc., that affect "machine level" storage operations do not affect the programmer's fixed picture of the basic logic of these operations. It can be highly advantageous, at the programming-language level also, to have a similar facility. We may also take the hint from the hardware analogy that when we say that

(17) $\underline{op}\ a = x$

is equivalent to a no-operation after

(18) $a = \underline{op}\ x$

has been performed, we may in fact be ignoring very extensive "behind the scenes" operations (like the "paging" operations which hardware may perform). A "no-operation" in the sense intended is merely an operation irrelevant to all those aspects of an algorithm with which one must be concerned.

It may also be remarked that the 'unraveling' process discussed above may be carried over to more general nests of sinister calls. Consider, for example, the storage function *select* described by (13) and (14) above. It is heuristically clear that one ought to be able to assign a reasonable meaning to

(19) select(select(f,g,i), select(ff,gg,i), j) = x; .

If the sinister call appearing on the left were instead dexter, it would retrieve

```
if j gt 0 then if i gt 0 then f(i) else g(-i)
          else if i gt 0 then ff(i) else gg(i) ,
```

making it plain what storage operation (19) ought to represent. The appropriate way to unravel (19) is as follows:

(20)
```
 i.    t = select(f,g,i);
ii.    tt = select(ff,gg,ii);
iii.   select(t,tt,j) = x;
iv.    select(f,g,i) = t;
 v.    select(ff,gg,i) = tt; .
```

Note then that the sequence (20) is appropriately related to the dexter call

(21) x = select(select(f,g,i), select(ff,gg,i),j);

Indeed, if (21) is executed immediately before (20), then after (20.i) and (20.ii) have been executed, we have

(22) select(t,tt,j) ≡ x ,

so that (20.iii) is equivalent to a no-op, and may be removed. But then (20.iv) is a no-op, since preceded by (20.i); and (20.v) a no-op, since preceded by (20.ii). The storage-retrieval relationship between (20) and (21) is therefore plain.

The formal argument just given plainly applies to arbitrary combinations of retrieval functions by nesting; this remark leads to the following substantial generalization of the fundamental statement A made above.

Statement B: The family of multi-parameter retrieval operators associated with the set of stored objects of a procedural programming language is closed under the operation of substitution.

Note also that all of the above applies to 'parameterless' retrieval functions; functions of this kind might for example be associated with retrievals from (and stores into) certain behind-the-scenes system attributes. Consider for example the storage-retrieval pair defined as follows:

```
definef place; return if a gt 0 then b else c; end place;
define  storeplace(x); if a gt 0 then b = x; else c=x;
        end storeplace;
```

The appropriate unraveling of

(23) `op place = x;`

is then

(24)
```
t = place;
op t = x;
place = t;
```

very much in the manner of the prototype sequence (U i-ii-iii).

Yet another property of our procedure for unraveling a nested sinister call is worth noting. If we consider the sinister call

(25) `select(f,g,select(ff,gg,i)) = x;`

and note from the definition (13) of *select* that this function does not modify its third argument, it is apparent that the most appropriate expansion of (25) is

(26)
```
t = select(ff,gg,i);
select(f,g,t) = x;    .
```

That is, one would want to regard the inner call to *select* as being implicitly dexter.

Our normal sinister call expansion, applied mechanically, would instead give

(27) i. t = select(ff,gg,i);

ii. select(f,g,t) = x;

iii. select(ff,gg,i) = t;

But (26) and (27) are equivalent! Indeed, since (27.ii) does not change t, it follows that (27.i) and (27.iii) remain mutually inverse retrieval and storage calls, so that (27.iii) is a no-operation. Aside therefore from implications concerning efficiency, the standard sinister expansion (27) is perfectly acceptable. Note also that an optimizer capable of detecting the fact that select does not vary its left-hand side could automatically exploit this fact to suppress (27.iii) as redundant.

The procedure for expanding sinister calls suggested by (6)-(7) and (19)-(20) is thus general and unambiguous.

In some cases, the storage operation corresponding to a retrieval may be deduced from the form of the latter. In other cases, this may be entirely impossible, since a storage operation may set in motion some special train of actions (such as a call to a behind-the-scenes 'allocate' function) impossible to anticipate merely from the form of the associated retrieval operation. In either case, however, it is apt to be convenient to keep the storage and the retrieval code together, as a good deal of this code, if not all of it, is likely to be common to both situations. The following syntactic proposal is intended to conform to these principles.

1. Explicit store and load blocks. A function which performs a retrieval operation may contain *storage blocks* and *load blocks*. The form of a load block is

(28) (load) block;

the load block may also be terminated in any of the styles

(29) (load) block end;
(load) block end load;

The form of a store block is

(30) (store name) block;

which may also be

(31) (store name) block end;

(store name) block end name;

Here *name* is any legitimate SETL name. Code not belonging either to a store or a load block is called normal code.

Whenever a function is called, a 1-bit flag signifying whether the call is a dexter or a sinister call will be transmitted. Normal code will be executed in any case. *Store blocks* will be *bypassed* if the call is dexter; *load blocks* bypassed if the call is sinister.

2. Return statements within load blocs, store blocks and normal code. The form of a return statement within a load block will be that normal for return statements within a function. The form of a return statement within a store block will simply be of the form appropriate for a subroutine return. Within the store block (30), *name* will designate the quantity to be stored, which is available as an implicit argument of the call, transmitted when a sinister call to a function is initiated, but appearing by syntactic convention in the store block header rather than among the arguments listed in the function header. Note also that the sequence

return a;

in a load block has as its most obvious correspondent in a store block the sequence

a = name; return;

Return statements in normal code will have the form appropriate for function returns, but will be expanded in a somewhat unconventional fashion. Each statement

(32) return expn;

occurring in a normal code section will be syntactically analyzed, and expanded into two blocks of code, the first being a load block;

the second being a store block. The load block corresponding to (32) has simply the form

(33) (load) return expn;;

The store block corresponding to (32) involves a more elaborate transformation of (32). We explain this in 'top down' fashion.

a. The store block corresponding to (32) has the form

(34) (store name) block; return;;

whose *block* is a code sequence determined by the syntactic form of *expn*, in a manner to be explained. We call *block* the *corresponding code* of *expn*.

b. If *expn* has the conditional form

(35) if $cond_1$ then $expn_1$ else if $cond_2$ then $expn_2$ ... else $expn_k$

Then the corresponding code of (35) is

(36) if $cond_1$ then act_1 else if $cond_2$ then act_2 ... else act_k ,

where act_j is the corresponding code of act_j.

c. If expn is a call on a programmer-defined prefix function, and thus has the form

(37) $f(expn_1,...,expn_k)$,

then the code corresponding to (37) is that obtained by expanding the sinister call

(38) $f(expn_1,...,expn_k)$ = name;

here *name* is the variable name appearing in the header of the store block in which (37) is imbedded. We suppose however that in this expansion every superfluous sinister call

(39) $expn_j = t_j$;

corresponding to an argument of f which is not modified by f is suppressed; cf. the preceding discussion centering upon formulae (25)-(27).

d. The preceding remark (c) may evidently be carried over to programmer-defined functions defined in infix or prefix form, and to programmer-defined functions of zero arguments as well. We extend it also to all those basic SETL operations which are retrieval-operators. These are

hd a, tℓ a, a[n] with integer n, a(x), a{x}, etc.

e. If the principal operator in an expression is a basic SETL operation which is not a retrieval, the expression is disqualified. This remark applies to such operations as

{a,b}, a+b, etc.

The code corresponding to a disqualified expression is

```
error;
```

where the function invoked is a system error routine.

To give a composite example, we note that the store block corresponding to the return statement

```
return if a gt 0 then progf(a)[progf(b)] else Ω;
```

is

```
(store name) if a gt then t=progf(a); tt=progf(b);
        t[tt]=name; progf(a)=t; else error;;
```

It is also worth noting the operation

(40) y = <a,b>

is a two-parameter retrieval. The associated storage sequence is

(41)
```
a = hd x;
b = tℓ x;   .
```

Indeed, if (40) has just been performed, (41) amounts to a no-operation; while if (41) has just been performed, x being any expression, the operation (40) will have y = x as its effect. Thus (41) is the natural way to interpret the assignment statement

(42) <a,b> = x; .

That is, the SETL multiple assignment statement fits naturally into the scheme described here. The meaning assigned to complex nested assignments like

(43) <<a,b>,c> = x;

may now be derived by our general expansion rules.

These observations make it entirely plain that we may freely allow 'multiple' sinister calls, corresponding to the SETL multiple assignment form

<a,b> = x; .

For example,

<select(f,g,i), select(ff,gg,i)> = x;

expands very simply as

select(f,g,i) = hd x; select(ff,gg,i) = tl x; .

Having now described in sufficient detail the syntactic and semantic mechanisms involved in sinister calls, it is appropriate to give various examples of their use. We begin with examples not involving the explicit use of store or load blocks. Our first example is really intended for sinister use only. Called as

(44) newtop stack = x

it places x on a stack. If used in a dexter call, it returns the value Ω. The definition is simply

```
definef newtop stack; return stack(#stack+1); end newtop;
```

The following slightly more elaborate example gives a routine which will place a specified value in the first integer for which a sequence seq, defined on a sparse subset of the integers, is undefined.

```
definef slot seq; return if 1 ≤ ∃[n] ≤ #seq | seq(n) eq Ω then
      seq(n) else seq(#seq+1); end slot;
```

Note then that if the values of seq are

1,2,3,Ω,Ω,4,...

then execution of

slot seq = 10

gives seq the values

1,2,3,10,Ω,4,...

Suppose that we take a binary tree to be represented by a triple <node,ℓ,r>, where ℓ is the left descendant function in the tree and r is the right descendant function in the tree. The following function will permit a new element x to be hung at the lower-leftmost element below node; to do this, we have only to write

leftchild tree = x;

The required definition is simply as follows:

```
definef leftchild tree; <node,ℓ,r> = tree;
(while ℓ(node) ne Ω) node = ℓ(node);
return tree[2](node); /* square-bracketed integers used for tuple
components */ end leftchild;
```

The following function, which allows the last component of an n-tuple to be set to a value x by executing

last tupl = x;

shows that recursive sequences of sinister calls can be useful.

```
definef last tupl; return if tupl eq Ω or atom tupl then
   tupl else last tℓ tupl; end last;
```

Next we consider a routine which accomplishes associative or hashed addressing, storing attributes of character strings. The individual attributes are numbered.

```
definef m attr name; /*fetches-stores attribute
          of a given name*/ /* as in hashed symbol table */
/* hash converts a string to a hashed integer, which serves
   as an index to the table 'firsloc', which contains (either 0 or)
   an index to the table 'stack', which has entries

          <chainpointer (or 0), firstchar,
                numofchars, attribute entry> .

here 'firstchar' serves to locate the first character of the token,
packed into a packed character array 'string'; 'numofchars' is
again the token length, 'attribute entry' contains the various
attributes of the name */

start = hash name; if (loc is firsloc(start)) ne 0
   then oldloc = loc; go to lookfor;
/*else*/ lookup(start) = loc is (#stack+1);
newentry: n1 = len string + 1; n2 = len name; string=string+name;
   newtop stack = <0,n1,n2,initial>;
/* where initial returns the initial setting of an attribute
   entry, corresponding to the 'all attributes undefined'
   condition */
ret:  return m att (stack(loc)[-4]);
/* note here that we are using a[-j] in place of the former
   notation <* zt 4>a */
lookfor:  (while loc ne 0 doing oldloc=loc; loc=hd stack(loc);)
   if n2 first ((len string-n1) last string)
   /* bah */ eq token
   then go to ret; end if; end while;
   hd (stack(oldloc)) = #stack + 1; go to newentry;
end attr;
```

One may now write

```
m attr name = val;
```

with the expected results.

The details of the retrieval function j att k control the detailed manner in which particular attributes will be stored. This point will be discussed more expansively below. Here we may simply remark that if the attribute entry is simply a k-tuple of attribute values, we may simply write

definef j att k; return k[j]; end att;
definef initial; return <0,...,0>; end initial;

Now we give some examples involving the explicit use of store and load blocks. The first is a modified *newtop* function, intended for use with tuples. When called in the fashion (44), it adds x as the first component of a tuple. The necessary definition is simply

definef newtop tupl; (load) return Ω;; (store x) tupl=<x,tupl>;
return;; end newtop;

The reader will find it interesting to write a function *newlast* which when called in the form

newlast tupl = x;

adds x as the (next-to) last component of a null-terminated tuple.

The following more complex example will show the manner in which multiple sinister calls may be used. We consider a hypothetical (systems-) programming situation in which names will be encountered, and in which each name may have one of a substantial number of attributes. Certain attributes of a name may be undefined, however. We suppose those attributes of a name which are defined to be character strings, and shall

describe a treatment of this situation in which these character strings are maintained locally as long as the total number of characters which they involve does not exceed a certain limit When this total becomes excessive, however, the character strings are assigned an integer identifier with a fixed number of bits, and moved to a packed storage array. However, if the total space needed to store these integers itself becomes excessive, only some of them are maintained locally, while the others are moved to an overflow area in the packed storage array, where they are referenced by a pointer stored locally.

The last few sentences clearly describe a fairly elaborate storage mechanism. It is therefore worth emphasing that this whole mechanism can be hidden from a programmer using it, who can merely write

j attr name = x;

to assign the j-th attribute of name, and use the expression

j attr name

to fetch the value of this attribute.

The behind-the-scenes code thus invoked is as follows. The attr routine is precisely the hash table routine shown above. The inner retrieval routine att is different, however.

We shall suppose that a maximum of nats distinct attributes are to be accommodated, and that the 'attribute entry' appearing in the routine attr is a logical quadruple, consisting of the following items:

a. lstring, containing either locally stored character strings, or integers (represented by character strings) referencing character strings in a packed array (perhaps indirectly);

b. lflag, signalling which of these two possibilities actually holds;

c. deflag, a group of flags, nats in number, indicating which of the total set of possible attributes are actually defined. The j-th of these flags is represented as

j deflag entry

d. manyints, a flag which is set if so many integers have been stored that the use of an integer overflow area has become necessary.

Various significant conventions concerning the use of certain other pointers will emerge as we work our way into the code below. In all of this it is worth noting that the techniques adopted are, generally speaking, those appropriate to a situation in which, even though a fairly large number of attributes can be defined for any item, most items will in fact have only a few attributes actually defined.

```
definef j att entry, jkeep=j;
/* first examine for various degenerate cases */
(store newstring) if newstring eq nulc then
   if n j deflag entry then return;;
else /* if nonnull newstring */ new = if(j deflag entry eq f)
   then t else f; /* post new entry if appropriate */
   end if newstring; end store;
(load) if n (j deflag entry) then return nulc;; new=f; end load;
/* now count the number of preceding entries that are defined,
   in order to get the appropriate 'internal address' of the
   information sought */
ndefs = [+ : 1 ≤ jj < j | jj deflag entry]1 + 1;
/* and look up the information in a 'submemory' of appropriate
   type, depending on the flag settings */
return if &flag entry then ndefs locstring lstring entry
   else ndefs intof lstring entry;
end att;
/* next we give the routines which define the various types
   of 'submemory' used in the above */
/* first an auxiliary routine for retrieving the j-th word in a
   blank-delimited string memory */
definef j word string; /* locate start of j-th word */
bc = 1; (1 ≤ ∀k < j)(while string(bc) ne ' ') bc=bc+1;; bc=bc+1;
end ∀k; ec=bc; (while string(ec) ne ' ') ec=ec+1;; ec=ec-1;
(load) return string(bc,ec);
```

```
(store newstring) lencng = len newstring - (ec-bc+1);
if newstring ne nulc then
   string=string(1:(bc-1)) + newstring +
    string(ec+1: len string -(0 max,lencng))+(0 max -lencng)*' ';
else /* null word case */ string = string(1 (bc-1))
   + string(ec+2, len string) + (lencng+1)*' '; end if newstring;
   end store;

end word;
/* now the string store-load function, with the appropriate
overflow actions, and the special actions to be used  when
a new entry must be made */
definef j locstring string; att external lflag, deflag, entry,
   nats, new, jkeep;
/* retrieval is simple*/
(load) return j word string;
/* storage is more complex, as it involves insertion and
possible restructuring of the data form of an entry */
(store newstring) if n new go to notnew;;
/* insertion case. first count all characters present */
nc = [+: 1 ≤ ¥jj ≤ nats | jj deflag entry and jj ne jkeep]
   (len (jj word string)+1);;
/* make insertion if new element + blank will fit */
jkeep deflag entry = t;
if (nc+1+len newstring)  ℓt maxchars /* 'maxchars' is the
maximum number of characters that can be accommodated locally*/
then j word string = newstring + ' ' + j word string; return;;
/* else must convert to packed integer format; it is convenient
 to think of this as a matter of putting all the items with which
 we are concerned into a 'memory' organized according to the
 intof principle */
do transfer; return;
```

```
block transfer; lflag entry = f; /* indicating transition
to new format */ newst = initialint /* starting an 'empty
memory' or 'workspace' for building up the required code format*/
jj=1; (while jj word string ne nulc doing jj=jj+1;)
jj intof newst = jj word string;; string = newst;
j intof string = newstring; end transfer;
[notnew:] /* not a new entry */ if len newstring &t len(j word
   string) then j word string = newstring;
if newstring eq nulc then jkeep deflag entry =f;; return;;
/* else will expand.count the characters already present */ else
nc = [+: 1 ≤ ∀jj ≤ nats | jj deflag entry] (len(jj word string)+1);
/* if the revised string will fit, merely make insertion */
if(nc+len newstring -len(j word string)) &t maxchars then
   j word string = newstring; return;;
/* otherwise convert */ do transfer; return; end if len;
end locstring;
/* now we give the routine which fetches and retrieves attributes
 represented indirectly using integer addresses (coded as
 character strings) which reference the packed character
 array called 'string' in the basic hashing routine 'attr' */
/* 'str' is our coded string of addresses */
definef j intof str; att external new,deflag, entry, jkeep;
   attr external string;
(load) <start,num> = j slot str;
   return edited string(start, start+num-1); end load;
definef edited token; nc=1; (while token(nc) ne ' ') nc=nc+1;
 (if nc eq len token the quit;; return token(1: nc-1); end edited;
/* the corresponding store procedure will have to check to see
 whether the attribute being stored will fit in the place avail-
 able for it. if not, a new place at the end of 'string'
 will be taken */
(store newstring) /* is this a new attribute */
```

```
if n new then go to notnew;; /* else */ jkeep deflag entry=t;
[newlike:] j slot str = <len string+1, len newstring>;
   string = string + newstring; return;
[notnew:] if newstring eq nulc then j slot str=<0,0>;
   jkeep deflag entry = f; return;;
/* else check to see if 'newstring' will fit */
 <start,nc> = j slot str;
if nc ge (ℓ is len newstring) then string(start: start+nc-1)
   = newstring + (nc-ℓ)*' '; return;;
/* else too big to fit. put at end of packed string */
 go to newlike; end store newstring; end intof;
/* next follows the 'slot' function which handles the coded
 addresses used above */
definef j slot str; /* once more,'str' is  our coded string
 of addresses, while 'string' is the packed character array
 introduced in 'attr' */
/* this is our approach:        each 'slot' will be of fixed
 length. it will be in the condition <0,0> if unused. when a
new element is to be stored, an entry will be pushed into an
unused location if possible, otherwise an overflow area will be
created. if an overflow area itself overflows, then a new
overflow area twice the size of the old will be created, and
the old entries placed at its bottom. note also that if the
manyints flag is set, the first slot in str contains a pointer
to the overflow area;
                but the access function 'field' hides the effects
of this */
att external manyints, entry, new;
if new then go to newslot;; /* else not new. scan to
appropriate location */
loc = 1; (while (loc field str) eq <0,0>) loc = loc+1;;
(1 ≤ ∀jj < j) loc=loc+1; (while(loc field str) eq <0,0>) loc=loc+1;;
end ∀jj; return loc field str;
```

```
[newslot:] /* here we determine whether the address desired
 is in the available range */
totslots = if manyint entry then locslots + numslots str-1
                            else locslots;
/* 'numslots' fetches the number of 'overflow slots' that
 have already been created */
if j gt totslots then go to expand;;
[putin:] if j gt 1 then go to jbig;;
/* else find first blank, and move everything up to it */
loc=1; (while (loc field str) ne <0,0>) loc=loc+1;;
moveup(1,loc); return 1 field str;
[expand:] if manyint entry go to ismany;;
/* here an overflow area is being created for the first time */
<start,nc> = <len string +1, len str>;
sizhead = len str -     sizslot;
/* 'sizslot' gives the number of characters in a single 'slot */
newst = str (sizhead +1 , len str) + sizhead ' ';
string = string + newst; str =     sizslot ' ' + str(1,sizhead);
1 field str = <start,nc>;
manyint entry = t;
/* thereby switching to new mode of reference */  go to putin;
[ismany:] /* here we must perform expansion when an
   overflow area already exists */
<start,nc> = <len string+1, 2*numslots str>;


/* temporarily drop flag to alter reference mode */
manyints etnry = f; <oldstart,oldnc> = 1 field str;
1 field str = <start,nc>; manyints entry = t;
string = string+string(oldstart, oldstart+oldnc-1)+ oldnc*' ';
go to putin;
```

```
[jbig:] /* scan to (j-1)st  character,  keeping account of
   last blank;
loc=1; lastbl = Ω; (while(loc field str) eq <0,0>) lastbl=loc;
loc=loc+1;; (1 ≤ ∀jj < j-1) loc=loc+1;
(while(loc field str) eq <0,0>) lastbl=loc;loc= loc+1;;
end ∀jj; /* if next character is blank, make insertion there */
if(loc+1) field str eq <0,0> then return (loc+1) field str;;
/* otherwise pack down to last blank, if any */
if lastbl ne Ω then movedown(lastbl,loc);
   return loc field str;;
/* if no preceding blank, find next blank and move up */
loc2 = loc; (while(loc2 field str) ne <0,0>) loc2=loc2+1;;
moveup(loc,loc2); return loc field str;
end slot;
/* now miscellaneous auxiliary routines */
define movup(jlow,jhi); slot external str;
(jhi > ∀n ≥ jlow)(n+1) field str = n field str;;
                return; end moveup;
define  movedown(jlow,jhi); slot external str;
(jlow < ∀n ≤ jhi)(n-1) field str = n field str;; return;
   end movedown;
define numslots str; /*fetches-stores number of slots from
   first slot */
att external manyints, entry;
/* temporarily drop flag to alter reference mode */
manyints entry = f;        <   -  ,lopart> = 1 field str;
manyints entry = t; return lopart/sizslot; end numslots;
```

```
/* next the basic slot-field storage-retrieval function */
definef j field str; att external manyints, entry;
   attr external string; slot external sizslot;
if n manyints entry then return convert str((j-1)*sizslot+1,
         j * sizslot);;
/* else might be local reference, shifted */
if(j+1) le((len str) / sizslot) then
   return convert str( j * sizslot + 1, (j+1) * sizslot);;
/* else is remote reference */
jover = j + 1 - ((len str) / sizslot);
<start,-> = convert str(1,sizslot) ;
return convert string((jover-1)*sizslot+1+start,jover*sizslot+start);
end field;
definef convert cslot; /* the conversion used just above,
   showing the manner in which a 1-1 conversion is represented
   as a store-load pair */
/* for simplicity, we suppose an octal representation,
   through of course a denser representation is possible */
(load) /* converts from octal characters to binary pair*/
slot external sizslot;
return <oct cslot(1,hpend), oct cslot(hpend,sizslot)> ;
/* here 'hpend' is the number of characters in the 'high portion'
   of a slot */ end load;
(store octpair) /* performs the reverse conversion */
cslot = (hpend poct hd octpair) + (sizslot-hpend)poct tl octapir);
   return;  end store;
/* the routine poct does octal conversion with padding
   to a specified length */
end convert;
definef j poct n; return(j-len oct n) '0' + oct n; end poct;
/* now the function which initializes a string of specified
   length to the initial 'all zeros' condition  expected by
   the routine locstring */
definef initialint; slot external sizslot, locslots;
return sizslot * locslots *'0'; end initialint;
```

Summary and anticipations:

Reviewing the above, we see that we may be reasonably well-satisfied with the manner in which our sinister call conventions have served to isolate from each other the decisions concerning storage technique which had to be taken to build up the rather complex storage mechanism described by the preceding algorithm. The simplifications thus attained open the way for the following observation, which may be a pointer to certain still more significant improvements in the way in which we are able to look at this very important memory management area. In building up the memory management structure described by the overall package of programs attr-att-locstring-intof-word-slot we are in effect connecting together a set of mechanisms, each of which calls upon certain of the others for services, and which all together define the more complex mechanism that is the total memory management system. It should be possible, by classifying the types of submechanisms that are used, and by establishing standardized linkages between them, to build up a 'memory management language' which enables memory management techniques to be specified in a very compressed and convenient manner. This is a point very much deserving of future study, and one that is closely connected with some of the speculations concerning data-description languages which are to be found in newsletter 31.

To be somewhat more specific, we may note that the hashing routine attr described above functions as an 'address-transformer'. That is, it is addressed by two parameters 'm' and 'name', of which the latter may be an arbitrarily long character string. It then addresses the 'second level' mechanism 'att' with two addresses 'm' and 'j', both integers. In order to perform this service, 'attr' itself requires a number of supporting storage functions:

i. a storage 'firsloc', in which a fixed number of integers may be stored.

ii. a storage 'string', in which strings of arbitrary length may be stored, and in which they may be addressed by a pair of integers.

iii. a storage 'stack', in which triples consisting of an address within 'att', an address within 'string' and a self-reference address within 'stack' itself may be stored.

The second level routine 'att' itself then functions as an address transformer, and, rather more specifically, as a type of 'compressor'. Programmed somewhat more rationally than we have done, it would be addressed by two integer parameters m and k, and would then call upon the third level routine 'locstring', passing parameters j and k, j calculated from m but generally varying in a smaller range. For this purpose, 'att' requires the supporting storage function 'deflag', which, for each k, stores a vector of a number of bits equal to the range of m, etc.

Sinister calls and programmer defined object types.

The sinister call mechanism seems to harmonize rather well with the syntactic mechanisms presently contemplated for use in connection with programmer-defined object types. A few points of specific influence seem to occur, however; these will be described in a subsequent newsletter, proposing syntactic forms to be used in connection with object types.