SETL Newsletter # 166

R. Dewar
J.T. Schwartz
April 8, 1976

An Easy Scheme for Incorporating Backtracking into the New SETL Implementation.

This newsletter will outline an easy scheme allowing backtracking to be incorporated in the presently planned new (optimized) SETL implementation. The scheme to be outlined can handle either simple ('PLANNER type') or full ('CONNIVER type') backtracking. To simplify our initial exposition, we will at first consider only simple backtracking.

Simple Backtracking: Let the two basic implementation level data areas be called HEAP and STACK. We introduce two macros HEEP and STAK which will always be used in writing stores into these arrays (whereas in loads we will always write HEAP and STACK directly.) In the non-backtracking case, these macros are simply

```
+ * HEEP = HEAP **; + * STAK = STACK **
```

In the simple backtracking case, the following macros are used instead:

```
+* HEEP(I) = HEAP(H(I))**
                    + * STAK(I) = STACK(H(I)) ** .
```

The routine H thereby introduced operates as follows:

Two auxiliary data structures CELLINX and RESTORE will be maintained by the routine H. CELLINX is a table of roughly byte-size entries containing two fields. The first of these, ENVNO, gives for each cell in the HEAP (equivalently, STACK), the index of the last prior data environment in which the value of the cell was changed.

The second, HASPOINT, indicates whether or not the cell has Dewar's standard 'specifier' format. RESTORE is an array, having entries somewhat in excess of WORDSIZE, divided into three fields:

OLDVAL: a WORDSIZE field, saving old-environment HEAP and STACK values;

OLDINX: a field approximately 1 byte in size, saving the old CELLINX value associated with the word held in the OLDVAL field;

PLACE: an INDEXSIZE field, giving the address in HEAP (or equivalently, STACK) from which the word held in the OLDVAL field comes.

The action of H(I) is as follows:

```
If   ENVNO  CELLINX (I) <CURRENTENVIRONMENTLEVEL THEN
           OLDVAL   RESTORE (RESTORETOP) = HEAP(I);
           PLACE    RESTORE (RESTORETOP) = I;
           OLDINX   RESTORE (RESTORETOP) = CELLINX(I);
     ENVNO  CELLINX(I) = CURRENTENVIRONMENTLEVEL;
          RESTORETOP = RESTORETOP + 1;
END IF;
RETURN I;
```

If necessary for efficiency, H can be made an in-line LITTLE primitive.

The CELLINX table should be maintained packed, e.g., on the 6600 we can pack 8 7-bit bytes to a word; thus only 12% of the space otherwise available for HEAP + STACK will be lost owing to the necessary to maintain the CELLINX table.

In this scheme, the ok and fail primitive of simple backtracking have the following representations:

i. The *ok* primitive

```
        OLDINX    RESTORE (RESTORETOP) = OLDRESTORETOP;
$   ALSO SAVE THE CURRENT INSTRUCTION LOCATION COUNTER
        OLDRESTORETOP = RESTORETOP + 1;
        RESTORETOP = OLDRESTORETOP;
        IF  CURRENTENVIRONMENTLEVEL > MAXLEVEL  THEN
                                ERROR;
        END IF;
        CURRENTENVIRONMENTLEVEL =
                    CURRENTENVIRONMENTLEVEL + 1;
        RETURN TRUE;
```

ii. The *fail* primitive

```
        DO   J = OLDRESTORETOP TO RESTORETOP - 1;
                 HEAP(PLACE RESTORE(J)) = OLDVAL RESTORE(J);
                 CELLINX (PLACE RESTORE(J)) = OLDINX RESTORE(J);
        END DO;
        RESTORETOP = OLDRESTORETOP - 1;
        OLDRESTORETOP = OLDINX RESTORE(RESTORETOP);
$   ALSO RESET THE INSTRUCTION LOCATION COUNTER TO ITS SAVED VALUE
        RETURN FALSE;
```

The garbage collector must make appropriate adjustments in the indices OLDINX RESTORE(J) and any pointers held in PLACE RESTORE(J) when it moves words in memory. Moreover, the occupied portion of RESTORE, i.e., the entries 1 thru RESTORETOP - 1 must be treated by the garbage collector just as if they were heap words, i.e., items referenced by pointers held in these entries cannot be treated as garbage, and tracing must proceed thru them.

(Note: excess tracing can be suppressed by keeping note of the lowest RESTORE entry changed after each garbage collection, which defines a part of RESTORE through which tracing need not proceed. The same remark applies to the stack.)

For the above conditions to be met, it must be known for every HEAP word and RESTORE word whether the word contains a pointer (in which case it will be in standard, typed descriptor format) or whether it is pointer free (e.g., an untyped real or an internal part of a bit or character string.) This information can be kept in the HASPOINT flag of CELLINX (for HEAP + STACK words) and in the OLDINX field of RESTORE (for OLDVAL RESTORE words). It is also necessary that the length of each block that can be located by a 'LINK' pointer should be available in the block itself. Since all link pointers reference set elements, we can ensure this by including a two bit field in each element descriptor: this field will characterize the descriptor as being either a 1-word block, a two-word block, or a long block whose full length is shown in a standard field within the immediately following word.

When a block of HEAP space is allocated, its length and the distribution of pointers within it can always be made known; thus CELLINX entries reflecting this fact can always be set. When STACK space is allocated (always in initializing the parameters and internal variables of a recursive call) the same is true, and appropriate CELLINX modifications can be made. Such modifications should always set ENVNO CELLINX to CURRENTENVIRONMENTLEVEL.

Extended Backtracking (i.e., support of environment manipulation). This can be provided by using a scheme closely related to the scheme just outlined. As in NL 153 we assume an environment tree ET. With each environment we shall associate a *restore vector* having the fields OLDVAL, OLDINX, PLACE already explained.

We distinguish two cases dynamically. If an environment e is momentarily not the ancestor of the particular environment that is executing, then its associated RESTORE vector records all the modifications that must be applied to e's immediate ancestor environment to produce e. If e is the ancestor of an executing environment, or is itself an executing environment, then its associated RESTORE vector records the modifications that must be applied to e in order to produce its immediate ancestor.

The environment-manipulation primitives that we propose to support are the following:

i. *env* copy *env'*. Here *env* and *env'* must be environments, neither of which is the currently active environment. Moreover, *env* must be an ancestor of *env'*. This introduces a new node into the environment tree, as an immediate descendant of *env*. The environment is a logical copy of *env'*, and is returned as the value of the copy operation.

ii. destroy *env*. This operation will check that *env* is neither the currently executing environment nor an ancestor of the currently active environment. It will then detach *env* and all its descendants from the environment tree, which will make several RESTORE vectors, and possibly also many other objects whose access chain leads through these restore vectors, garbage collectible.

iii. fix *env*. This operation, which has the flavor of 'success', checks that *env* is either the currently executing environment or some ancestor thereof, and destroy all environments which are not descendants of *env*; *env* becomes the root of the environment tree.

iv. *env* try *val*. This is a somewhat generalized version of the *cocall* function of NL 155. It transfers control to *env* , which, like every environment other than that which is currently executing, is waiting 'halfway through' an earlier try operation.

The quantity *val* is received by *env'* as the value of this earlier try operation. If *env'* is not a twig of the environment tree, then it is made a twig, specifically by destroying all its descendant environments.

The try operation can also be used monadically; the monadic operation try *val* creates a new environment *env'*, which becomes an immediate descendant (in the environment tree) of the currently executing environment; then *env'* try *val* is executed at once.

v. The nulladic special quantity self has the currently executing environment as its value. The monadic operator parent *env* has the parent environment of *env* as its value. The monadic operator descenvs *env* has the set of descendant environments of *env* as its value.

The generalized backtracking primitives just introduced will normally be used only within a few utility macros, functions, and subroutines, which will be used to realize backtracking control structures of somewhat higher level than the primitives themselves. These utility macros and functions will have access (probably exclusive access) to whatever auxiliary tables are needed to realize any desired control regime. We give a few significant examples to illustrate the style of programming that can be used:

A. The simple backtracking primitives can be defined as follows:

```
macro ok; try true endmacro;
macro fail; (parent self) try false endmacro;
```

B. Creating a sibling environment.

The monadic try primitive creates a descendant environment. For creation of sibling environments, we shall introduce a nulladic function *sibling* which when called will return a pair <true, *env'*>, where *env'* is a newly created sibling environment that can be considered to be suspended within the function *sibling*.

Any value *val* except its own parent can subsequently be passed to *env'* using the try primitive, at which time *env'* will receive <false, *val*> as the value of the function *sibling* within which it has been suspended. A typical use of this function might be in the combination

```
if hd(sibling( ) is sib) then go to label; else <junk,val>=sib;;
```

the code for *sibling* is as follows:

```
definef sibling;
x = self;  /* remember originally calling process */
 if try self is sib eq (parent self is par)
    $ this condition will be satisfied only in a strictly
      temporary auxiliary
    $ descendant environment created by the monadic try
      then
           junk = par try       (parent par) copy par ,
    $ the copy creates the sibling environment; the binary
    $ try return to the environment in which sibling
    $ was originally called
 end if; if x eq self then return <true,sib>; ; /* else */
 return  <false, sib>;  $ this return  is taken when the
                        $ sibling environment is entered
 end sibling;
```

C. An 'estimate' primitive. The estimate primitive replaces the simple backtracking fail with a more flexible estimate t, which estimates the amount of work needed to attain success (in some appropriate sense) by continuing calculation in the current environment. It is used in combination with a nulladic function ok. The definitions of these two functions are as follows:

```
define estimate t;
estimates (self) = t;
$ estimates is a map which records the estimated time to
$ completion of all existing environments
must = ∃est∈ estimates (env)|
                        est eq [max: est' = estimates(env')] est'
```

```
junk = env try false;
return;
end estimate;

definef ok;
if try true is val then  $ create copy of parent as sibling
     sib = (parent  self) copy (parent  self  is par);
     estimates (self) = estimates (par);
     estimates (sib) = estimates (par);
     estimates (par) = Ω; $ since par can no longer execute
end if;
return val;
end ok;
```

3. Additional details concerning implementation.

Implementation of most of the primitives described in the preceeding section is unproblematical. The self, parent, and descenvs primitives simply deliver information about the environment tree. The destroy primitive merely removes a portion of this tree. The fix primitive is almost as unproblematical, except that it may have the effect of giving the root of the environment tree a level different from 1. This is essentially harmless, but one wants to avoid an uncontrolled rise in environment level numbers. These numbers can be kept under control by lowering them systematically within CELLINX during garbage collection. Whenever the current environment level number would otherwise rise above the limit of acceptability, this same lowering operation can be applied.

To execute *env'* try *val*, which is potentially the most expensive of our primitives, we proceed as follows. Suppose that the current environment is *env*. Then we locate *env'* in the environment tree, giving an appropriate diagnostic if it cannot be located (e.g., if it has been destroyed).

Otherwise we determine the (highest level) common ancestor *enva* of *env* and *env'* by a chain-back process. Let the chain of ancestors leading back from *env* to *enva* be env = env_1,...,env_k=enva. Then the RESTORE information associated with each of these environments is used in turn, to rebuild the environment associated with *enva*. Following this, the chain of environments leading from *env'* to *enva* is traversed in the reverse direction to reconstruct the environment *env'*, and execution continues with *env'*. If *env'* is not an environment-tree twig, then all its descendant environments must first be erased.

The monadic try operation has a simpler and more efficient implementation. A new environment *env'*, with an initially empty RESTORE vector, is created and made an environment-tree descendant of the currently executing *env*; then execution continues with *env*.

Store and load operations are treated in the same way as the simple backtracking case, with levels contained in CELLINX being checked on each store operation. Information that needs to be added to RESTORE should always be appended to the RESTORE vector associated with the currently executing environment. The environment-switching costs associated with generalized backtracking are plainly higher than those associated with simple backtracking, but the intra-environment running costs are the same.

To implement the primitive *env* copy *env'* we can proceed as follows. Let the chain of environment tree nodes leading from *env* to *env'* be env=env_1,...,env_k = env'. First suppose that *env'* is not an ancestor of the currently active environment. Then we make a backward pass over env_k,...,env_2, building up a RESTORE vector v, for the new environment to be created, from the separate RESTORE vectors v_j of these environments.

This is done by taking all items from v_k, v_{k-1},...,v_2 in turn, and installing them into v. Each time an item x is about to be made part of v, the PLACE field of x will be extracted,yielding an integer I, and an auxiliary quantity AUX(I) calculated using a hash table built up temporarily during execution of copy operations. If J = AUX(I) is undefined, then x is appended to the end of the vector v, and AUX(I) is set equal to the component position of x within v. If J is defined, then we simply set the OLDINX field of v(J) equal to the OLDINX field of x. When all of v_k,...,v_2 have been processed, we create a new node in the environment tree; v becomes its RESTORE vector.

Next suppose that *env'* is an ancestor of the currently executing environment *envcur*. Then, by processing the RESTORE vectors v_2,...,v_k in forward sequence in the manner just indicated, but ignoring all fields other than PLACE, we can obtain a vector v which shows all the corrections that need to be applied to *env'* to yield the value *env*. By traversing the path from *envcur* to *env'*, we can reconstruct *env'*. Next, we make a pass over all the components x of v. Let J be the PLACE field of such an x; then we set OLDVAL and OLDINX of x to the current values of the J-th cell of memory and of CELLINX(J) respectively. This builds up the RESTORE vector associated with *env* copy *env'*. Once this is done, we traverse the path from *envcur* to *env'* in the reverse direction, to restore the current environment *envcur*, and create a new environment tree node as an immediate descendant of *env'*.

4. How to imitate some of the facilities of SNOBOL.

Griswold has remarked, and indeed it is not hard to see, that a language providing strings, a good set of string primitives, and backtracking can easily imitate the string matching facilities of SNOBOL. The following conventions accomplish exactly this imitation in a convenient way.

(a) A *pattern* P is a procedure, function or code sequence which accesses a public global variable (call it *globstring*) and either successfully performs some sort of (primitive or non-primitive) matching operation, or executes fail. If p does not fail, it modifies *globstring*, changing it from its original value to whatever part of *globstring* remains unmatched.

(b) Given a collection of patterns $p_1,\ldots,p_n$, we can combine them in various useful ways:

i. sequentially: p_1; p_2;

ii. as alternatives;

if ok then p_1; elseif ok then p_2; elseif ok then p_n; else fail;;

syntactically, it is better to introduce the macros

eo(p) = if ok then p; o(p) = elseif ok then p;

oe = else fail;

and write this as

eo (p_1) o(p_2)...o(p_n) oe;

another useful construction is:

iii. repetition:

arbno(p) = (while not ok) p;;

Patterns may have explicit parameters in addition to the implicit quantity *globstring* which they all access. This enables us to write patterns such as length(n), exactly('string'), approximately ('string') (which allows for some degree of misspelling), etc. Note that any executable statement can be inserted into a sequence of patterns, making it easy to determine if subparts of a string occur repeatedly, if a string subpart matches two or more separate patterns, etc. Moreover, by grouping the parts of a composite pattern together and making a procedure or function out of them, we can create new pattern objects, which can then be passed to other patterns as parameters.

For integrating all this into the SETL system, the following conventions might be appropriate: define a 'system' module called, e.g., *stringproc*, to which the public variable *globstring* belongs. Provide a library of string primitives, treating them as procedures which access this variable, and which only become available within another module m if *globstring* is included into m; in which case it may also be appropriate to make the macros o, and oe available in m.

Since in addition to the unmatched string portions which patterns will return (in *globstring*) we will often wish to make use of a matched string portion, it may be worth introducing the abbreviation $string_1$ - $string_2$ for string1 (1: # string1 - # string2).

5. Some remarks on optimization.

Any store to a variable that must necessarily have been written in the current environment can be performed in a specially efficient way, since its CELLINX field need not be checked. Since environments are always entered (and exited) at try operators, this remark will always apply to a store I =... if there is no path from a try to the store operation which does not pass thru another I =.... This remark shows that assignments I =... can sometimes be made more efficient within loops not containing any try operation by prefixing I = I to the loop entry.

In the simple backtracking case, the criterion for suppressing CELLINX checks that we have just stated can be relaxed significantly. Consider an assignment α of the form I =..., and let S' be the set of all ok operations with a true outcome that can preceed this assignment without some other assignment to the same variable I intervening. This set can be calculated by the standard dataflow technique used to calculate the 'reaches' function,except that forward tracing should begin with occurences of ok, and that in addition to the 'kills' which naturally occur at each occurence of an

assignment I =..., we insert a 'kill' immediately following each test "if ok then..." along the ok = false branch. Then if I is dead along the ok = false branch forward from each occurence of ok in the set S, the assignment x can be executed without checking CELLINX. The proof is as follows: any environment to which one can return after fail is executed will have been generated at some ok yielding the value true; after executing fail, this environment will be re-entered with an ok value of false. Thus it is clear from our assumption that since the moment at which we generated any stacked environment in which I will not be dead after return, some other assignment must have updated CELLINX.

This remark applies in a useful way to a wide variety of selection iterators of the form

```
if ∃I ∈ list | ok then a else b;;
```

whose expansion is

```
I = first(list);
(while I ne Ω)
     if ok then
          a; quit;
     else
          I = next(I);
     end if;
end while;
```

The optimization principle that we have encountered shows that CELLINX(I) need not be checked at the assignment to I within the loop.