SETL Newsletter # 186 April 21, 1977

Syntax and Semantics of a Restricted Backtrack Implementation

R. Dewar, J. Schwartz

If SETL is provided with some limited degree of backtrack capability, it will be possible for us to program pattern-matching operations of the SNOBOL type. This is a potentially significant semantic extension, especially if backtracking can be handled in an acceptably efficient manner. The present newsletter will try to define a set of backtrack primitives and an implementation for them which meets these criteria, and which in addition implies only minimal changes in the system developments currently under way.

The discussion of backtracking in NL 166 should be consulted, even though the present newsletter will modify earlier suggestions very considerably.

A. Semantic primitives. Nondeterminism is introduced via the selection primitive

(1) ∃s .

If s is a set, this selects an element from s, nondeterministically. If s is a tuple, bitstring, or character string, it selects a component nondeterministically, but in increasing order. If s is a positive integer, it selects an integer between 1 and s, nondeterministically, but in increasing order. (These last selections are nondeterministic in that they can be backtracked to and changed.)

If s is nl, nult, nulb, nulc, or a nonpositive integer, selection fails. This rule makes a separate fail operator superfluous; however, such an operator will be provided anyhow, for reasons of clarity and efficiency.

The backtrack implementation outlined in NL 166 assumes that all variables will be backtracked when failure occurs, and is devised to preserve efficiency even in the face of this assumption. In the present NL, we shall make the rather different assumption that only a relatively few variables will

be backtracked on failure; this should allow a reasonable level of efficiency to be secured by much less drastic modification of our present implementation.

The set of variables to be restored on backtrack will be defined statically. For this purpose, we introduce local and global backtrack declarations. Global backtrack declarations occur in the same place as var declarations in a module, and have the form

(2) backtrack $v_1,\ldots,v_n$;

where $v_1,\ldots,v_n$ is a list of variables. Local backtrack declarations have the same form, but occur within subprocedures. We also provide a local noback declaration, of the form

(3) noback $u_1,\ldots,u_m$;

The set of variables whose values are restored on failure back to a point P of nondeterministic selection are those which are in backtracking status at the (textual) location of P. If P occurs in the scope of (2), (3), and a local backtrack declaration listing $w_1,\ldots,w_k$, then the variables backtracked are:

$$\{v_1,\ldots,v_n\} + \{w_1,\ldots,w_k\} - \{u_1,\ldots,u_m\} .$$

The static character of (2) and (3) implies that all sets of variables which can simultaneously be in backtrack status at any moment during run time are known at compile time. The compiler can therefore associate each such set of variables with a pointer to the block of code which will backtrack the variables when this is necessary. (Additional details concerning this backtracking action are giver below.)

We allow the nondeterministic boolean valued expression ok as an abbreviation for ∃<true,false>. In addition, we introduce a closely similar but not identical nondeterministic boolean primitive okok, whose semantic specifics are as follows:

i. Each use of okok as an expression initially yields the value true, and also creates an identified backtracking environment which one can either fail out of in the normal manner or stabilize on success. To stabilize such an environment, we execute the statement

(4) accept;

This abolishes the topmost remaining backtrack point established by a previous execution of okok as an expression, together with all backtrack points stacked above it.

ii. When one fails back to an okok point backtracking of variables takes place in standard fashion, *the backtrack point established by the* okok *is deleted*, and the okok is made to return false. This rule establishes a certain symmetry between the treatment of failure and success returns to an okok point, but in the rare case in which there is a variable v in backtracking status at the moment of evaluation of the okok but not immediately before this moment, it has the consequence that changes to v made along the failed path from okok = true will not be propagated back to earlier environments, whereas changes to v made along the path from okok = false will, even if one backtracks to an earlier environment in consequence of a subsequent fail.

A typical pattern in which okok was used might be

(5)
```
if okok then
    code exploring a certain alternative;
    /* suppose that if we arrive here the alternative
       was correct */
    accept; go to L_success;
else
    go to L_failure;
end if;
```

In addition to accept, we introduce a statement

(6) reject;

which fails immediately all the way back to the last preceding okok point.

B. Implementation Approach.

We will describe a relatively high-efficiency, highly compiled implementation. Of course, a somewhat less elaborate, slightly more interpretive variant is easily possible.

2. Rather than simply containing return addresses and values stacked on calls (as at present), the stack will contain a sequence of areas with the following general layout.

low stack ...	return	backtrack	call	return	backtrack	call	hig
	$area_{n-2}$	$area_{n-1}$	$area_{n-1}$	$area_{n-1}$	$area_n$	$area_n$	... stac

Here, each backtrack $area_n$ will contain stored values of variables which might need to be backtracked to re-enter the n-th active environment, and call $area_n$ contains the values of variables stacked by execution of calls in this environment. The return areas contain information needed to undo returns made in later environments.

ii . To execute a non-deterministic selection or an ok, we proceed as follows. Let the set of variables momentarily being backtracked be $v_1,\ldots,v_m$, and suppose for example that backtracking is initiated by an evaluation of ∃s. We establish a new backtrack area on the top of the stack. In this area we save a copy of s, an iterator x referencing the first value of s, and also the values of $v_1,\ldots,v_m$; plus various utility pointers to code and to other system objects as described below. The share bits of $x,s,v_1,\ldots,v_m$ are set in the symbol table and on the stack. This initiates active environment n+1.

Subsequently, to re-enter environment n+1 at its logical 'backtrack point', we first use associated fragments of return areas to undo the effect of subroutine returns (with no matching calls past the backtrack point) that may have occurred

in the environment subsequent to this backtrack point (a process that will be described in more detail below). Then we restore the values of the variables $x, v_1, \ldots, v_m$ from backtracking area $n + 1$, and advance the iterator x. (However, if the backtrack point was established by an okok, it is abolished by popping the stack.)

iii. Call operations are handled in the normal way, by placing the values of all variables stacked by a called procedure (including its current parameters) on top of the stack, transmitting parameter values, and initializing symbol-table values of nonparameter variables to Ω by setting their is_undefined bits. This can be accomplished most efficiently by providing each procedure with pre-compiled machine-level 'stack' and 'unstack' entries, jumps to which cause in-line stacking and unstacking sequences to be executed.

Handling of return operations is more complicated. Some return operations, namely those which occur when call information is on top of the stack, are handled in the normal way (essentially, invoke the 'unstack' entry of the current routine). However, to handle a return which occurs when a backtrack area lies atop the stack we proceed differently. Instead of simply restoring the values of all stacked variables and popping the stack, we locate the last call area on the stack. This references the routine from which we are provisionally returning, and holds the pre-call values of all its variables; the symbol table holds the current values of these variables. We *interchange* values between the symbol table and the topmost fragment of call area; this converts this fragment of call area to a fragment of return area. All the return area fragments of the group $F_1, \ldots, F_k$ created by return operations executed when a given backtrack area B is on top of the stack are held on a list associated with B. Each new return performed when B is atop the stack adds a new F_{k+1} to this list.

The interchange operation which creates a new F_k can be handled most efficiently by providing each subroutine with an associate 'swap entry', which when invoked interchanges relevant variables between the symbol table and a designated stack area.

In general, it should be possible to handle all returns simply by jumping to a code address planted on top of the stack.

iv. Whenever we fail out of an environment, all the calls and returns which occurred subsequent to the opening of the environment, and all changes to variables in backtrack status, must be undone. We accomplish this as follows: force returns from all calls stacked above the backtrack block representing the environment (which is topmost); then, proceeding down the chain of return fragments associated with a backtrack block in the order $F_k,\ldots,F_1$, interchange all values held in the return fragments with the corresponding values held in the symbol table. This will restore portions of one or more return areas to the adjacent call areas.

Finally, restore all backtrack variable values by moving values from the backtrack area to the symbol table, and pop the backtrack area.

Note however that the values of variables which are neither in backtrack status, or stacked by a call or a return out of which we are backtracking, are not restored when we fail out of an environment.

v. Pointers to the topmost call currently on the stack, to the topmost backtrack block, and possibly also to the topmost backtrack block of okok type are always accessible.

Each backtrack block points to the last previous backtrack block. (A small optimization in the handling of backtracking is: attempt to advance before restoring variables. If advance fails, check last preceding block for identical restoration routine before actually doing restoration.)

vi. To handle the accept statement, we pop the stack from the topmost backtrack block down to the topmost backtrack block of okok type. To handle the reject statement, we force a chain of fails back to and just past the topmost backtrack block of okok type.

c. The SNOBOL primitives.

A forthcoming newsletter of S. Rapps represents these systematically in terms of general nondeterministic primitives. In general, our approach will be:

a. To reserve a variable name, e.g. *matchcursor*, at the system level. An implicit global declaration putting this variable in backtrack status is always assumed. By treating this variable in a special way we can gain some efficiency advantages. A global variable *matchstring*, not backtracked, is also assumed.

b. Provide a small library of string-oriented operators taken from SNOBOL. This will include variant of any, notany, len, span, break; also an exactly operator such that exactly 'XYZ' matches 'XYZ' and nothing else. These routines will all exemplify the way in which we shall treat patterns, namely as functions which access *matchstring* and *matchcursor*, and which either return a string, or fail. They will be optimized in the following ways:

i. The string catenate operator will check its arguments to see if it is handling two adjacent substrings of the same string. If so, it will preform a fast catenate, simply by producing a new string specifier. Similarly if one of its arguments is nulc.

ii. Calls

any string

will be handled roughly as

```
if string = savestring then return realany converted;
    else savestring=string; converted=convert string;
                        return realany converted;
```

Here, convert produces a bitstring representation of a set of characters (using machine level character codes), and realany

works from this. The operations notany, span, etc., will be handled similarly.

With these fundamental primitives working at high efficiency, we should be able to attain a reasonable fraction of the speed of SPITBOL.

We can also use a setup s; macro, equivalent to

matchstring = s; *matchcursor* 1;

D. A Few Syntactic Suggestions

To come close to the elegance of the SNOBOL alternative and ARBNO constructions, we need to think about syntax. A full expression-language approach may have advantages, but here we shall only explore certain more conservative possibilities:

i. Introduce a stacked, blank-named variable, local to the procedure or begin-end block (see below) in which it occurs. This makes it possible to evaluate an expression merely for its side effects, simply by writing an assignment, = *expn*, to the blank variable.

ii. Introduce a begin *code* end block. Syntactically, this is an expression; its value is the current value of the blank variable. We can exit from such a block either by a jump (which kills its value), or by a return, or by executing its end statement. Then the standard value language construction

statement; statement; statement; expression return L:

becomes simply

statement; statement; statement; = expression; return; L:

This allows us to write a close equivalent of the SNOBOL

(1) $(pat_1$ arbno(pat_2) $pat_3)$ $x

as

(2) x = pat_1 ! BEGIN aux=nulc; (while not ok) aux ! pat_2 ;;=aux; END ! pat_3 .

In this context we can also allow BEGIN to be abbreviated as [:, and END as]. Note then that ARBNO is simply a macro for

BEGIN aux=nulc; (while not ok) aux! pat;; =aux; END;

and that if no assignment is to be performed (2) can become

(2') $= pat_1; = arbno(pat_2); = pat_3$.

b. To handle alternatives, we need some kind of short-form 'case expression', for which we suggest

(3) $n\ th\ (expn_1, \ldots, expn_k)$.

If m lies in the range 1,...,k, the effect of this is to select $expn_m$ for evaluation and return its value; otherwise it fails. This allows the SNOBOL alternation

$$(pat_1 \mid \ldots \mid pat_k)$$

to become

$$\ni k\ th\ (pat_1,\ldots,pat_k) ,$$

and we might even allow the redundant k to be elided (here and in the corresponding case statement) to give

$$\ni th\ (pat_1,\ldots,pat_k) .$$