SETL Newsletter # 194

Sandra Rapps
June 13, 1977

Nondeterminism, Backtracking and Pattern Matching in SETL

1. Introduction

Inherently non-deterministic algorithms are frequently encountered. For some of these algorithms, the desired output is any member of a particular class of objects with no distinction made as to the relative desirability of the members of the class. Such algorithms characteristically contain some phrase such as "arbitrarily select an object from a set s". In other cases there is a unique objective; however the precise steps which must be taken in order to reach the desired goal cannot conveniently be specified a priori. Algorithms which handle such situations provide several paths of exploration which are systematically explored until the desired outcome is obtained. Implementation of non-deterministic algorithms generally involves fairly exhaustive "trial-and-error" searching of some problem space.

The widely used *backtracking* method for handling such situations involves making "guesses" as to the first steps of a partial solution, and then attempting to extend the partial solution to a complete solution. As soon as it is determined that a particular exploration cannot lead to the desired goal, the last guess made is discarded (along with all side effects generated since that guess was made) and some other provisional choice is made. Ignoring time constraints, at some point either a solution to the problem is achieved, or all possibilities have been exhausted and the algorithm terminates with failure.

It is helpful to visualize the backtracking process in terms of a tree. Each node represents the state of the process at the time that some non-deterministic decision must be made. Every node has one descendant for each alternative which can be selected at that point. At each stage of exploration exactly one node of

the tree is considered to be "active", with the root node being active initially. Whenever a decision must be made some (previously unexplored) descendant of the current active node is selected and becomes active.

In a programmed representation of this process, a *data environment* is associated with each node of the exploration tree. This environment indicates the binding of values to all program variables at the time that the particular node was last active. Descendant nodes receive a copy of their parent's environment when they are initially activated. As program execution continues, all changes of variable values are made in the currently active environment only. When a node is re-activated (due to a failure of one of its descendants) its own data environment is once again made active, and the environment of the failed node, which can never be re-activated, is discarded. Note that any side effects which occurred while the process was executing in the environment of a particular node n are invisible to the ancestors of n and will only affect computation in n and its descendants.

At any point in the simple exploration process that we have described, only those nodes on the path leading to the active node may at some future time be re-activated, and only the data environments corresponding to these nodes must be retained. At the implementation level, these environments need not be stored in their entirety, typically they are stored incrementally, i.e., enough information is associated with each node so that its environment can be recreated from its parent's environment (as in the Bobrow and Wegbreit model) or from its descendant's environment (as in SETL Newsletter 166).

II. A Quick Survey of Backtracking in SETL

As indicated in NL 166, we can make backtracking available in SETL by introducing primitives ok and fail. ok is a pseudo-boolean function which always returns true; but whenever it is invoked a data environment is created and becomes a descendant of the current environment, with execution continuing in the new environment.

Execution of fail causes a return to the most recently executed instance of ok, reassigning a value of false to ok. The environment created when ok had the value true is discarded, and processing continues in the environment which was active immediately before the statement containing the ok was initially executed.

Any side-effects caused by execution of a sequence of instructions beginning with ok and ending in fail are obliterated. Thus, when the program terminates it appears as though only those statements which actually led to the solution had been executed.

Example (The 8-Queens Problem) The 8-queens problem involves placing 8 queens on an 8 x 8 chessboard so that no queen can be attacked by any other queen. The output of the following program is a vector *rows*, indicating that a queen is to be placed in column n of row *rows*(*n*).

```
if ok then
     rows:=  nult; avail:=  {n, 1≤n≤ 8};
    (1≤ ∀ n ≤ 8)
          usable:=  {x ∈(avail - {r(j), r ∈ rows}-
                         {r(j) - n, r ∈ rows}- {r(j)+ n, r ∈ rows})};
          (while usable ne nl)
            x from usable;
            if ok then rows(n):=  x; continue ∀n; end if;
          end while;
          fail;
     end ∀n;
     print rows;
else print 'no solutions found';
```

Note that the final else statement is executed only if the initial ok takes on the value false. In each environment, n takes on a different value and a new set *usable* is created. When fail is executed (because the set of possible values of rows(n) for the current value of n is empty), the previous environment is re-entered, n and usable resume their former values, and a new guess is made for rows(n).

The preceding description of backtracking is too severely restricted to be useful in the cases we would like to handle. At any point in a program it only allows execution to continue in the current environment, fail back to its parent environment or generate and enter a new environment. No facilities for interaction or communication between environments are provided. To remove these restrictions, we can extend the ok and fail primitives in various ways:

(1) Introduce a new data type *environment* and function currentenv whose value is the currently executing environment. The only operations we allow on objects of data type environment are assignment, testing for equality, or use in one of the special ways indicated below.

(2) It is often useful for the value of a small number of variables to be preserved after a fail occurs within some environment. It should be possible to preserve the value that a variable had in an environment which fails, allowing, for example, a strategy which is dependant on the conditions that caused a failure to occur.

For this to be possible, we can introduce an of operator which enables assignments to ancestor environment to be made. If the value of *env* is some environment then

var of env: = exp;

sets the value of *var* to *exp* in environment *env* and all its descendant environments. A value for *env* of currentenv or any environment previously destroyed can be treated as an error. For reasons of efficiency, the of operator should be restricted to use only in the target of an assignment and should not be used to retrieve a value from a previous environment (as in x:= y of env) since descendant environments can always obtain values directly from their parents.

Examples showing the way in which this operator can be used are included in the section on SNOBOL-type pattern matching, which follows below.

(3) fail, as described above, always causes a return to the most recently executed instance of an ok. It is often the case that at some point during execution it can be determined that an entire sequence of choices has been incorrectly made and all environments corresponding to those choices should fail. In particular, a large number of environments may be generated iteratively or recursively and it is not always convenient or possible to explicitly include the code for propagating failure up the chain of oks. For this reason it is useful to provide a second form of fail. fail(env), where env is an ancestor of the current environment, causes the immediate failure of environment env (and all its descendants). The value of the instance of ok which generated env becomes false and the parent of env becomes the current environment.

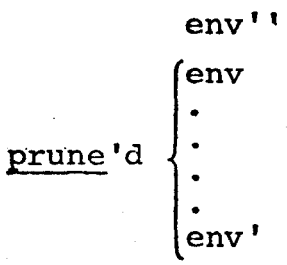
As an example of the use of the multiple fail, consider depth-first search. One commonly used heuristic in depth-first search algorithms is to provide a limit, *maxd*, on the length of any search path. Any attempt to extend the path length of a search beyond maxd causes immediate failure of that path. Such a search might be implemented as:

```
if ok then outside := currentenv;
    (1 ≤ ∀ d ≤ maxd)
        if ok then block1;else block2;end if; end ∀ d;
    fail(outside);
```

Assuming that block1 and block2 contain no occurrences of ok (they may, of course contain fails), at most maxd environments are retained at any one time, awaiting a fail to reactivate them, and therefore no path through the search tree on length > maxd is completely explored.

(4) The backtracking scheme thus far described does not allow nodes to be deleted from the tree unless a fail is executed. However, when a search involving backtracking terminates successfully, all ancestors of the environment in which the goal was reached are still in existence, awaiting a subsequent failure.

Should a second backtracking sequence be initiated within the same program it would be possible to inadvertently fail back into the already terminated search. To prevent such undesirable effects, we suggest a primitive prune(env). As its name suggests, prune removes a portion of the environment tree. The value of env must be an ancestor of the current environment, env'. Execution of prune(env) causes env and all its descendants to be removed from the tree, with env'' the immediate ancestor of env, becoming the current environment:



In addition all variables retain in env'' the values they had in env' immediately before prune was executed, making it appear as though all assignments made in env and its descendants were in fact made in env''. Execution continues with the statement immediately following the prune statement and the next execution of fail causes the failure of the ok which created env''.

The prune operator may also be used to cause a sequence of decisions to be treated as a single decision and force them to all fail together, if at all.

Execution of prune has the side effect of freeing storage by discarding all information associated with the deleted environments. Judicious use of prune can alleviate the stack overflow problem which might arise when a large number of environments must be saved.

III Nondeterministic Select Operators

A more general approach to providing non-determinism in SETL is the introduction of a new select operator, ∋*. This operator is defined to select an arbitrary element of a set in the same way that the current ∋ does, but would 'remember' which elements had been selected. If at some subsequent point it is

determined that the element selected does not lead to a desired solution, backtracking will occur and a different object will be selected. The semantics of the ∋* operator are as follows: When x:= ∋*S is executed, and element of S is selected and assigned to x, this element is somehow marked as 'already selected', and a new environment is generated and entered. A failure occurs when an attempt is made to select an element from a set which has no more unselected elements. When this happens, the most recent successful execution of ∋* is determined (that is the instance of ∋* which generated the current environment), the current environment is discarded, a new element is selected by ∋* and assigned, a new environment is opened, and processing continues in this new environment at the statement immediately after this ∋* operation. All computations performed between the time that ∋* selected an element and the time at which a failure occurred are effectively 'undone' by discarding the old environment.

The only exception to the description given above occurs when the last element of a set is selected. In this case no new environment need be generated, since a failure caused by the last element triggers an immediate failure back to some other instance of ∋*. Another important point to note is that if code such as:

```
x:= ∋* s;
s:= s + t;
y:= ∋* nl; /* this causes a failure back to
              the first statement */
```

is executed, x will only be selected from the original elements of s. This is because the selection of elements from s is done in the old environment, after the new environment, which contained the redefined value of s, has been discarded.

It is convenient to introduce for the ∋* nondeterministic primitive, a set of side-effect operations such as those described for ok and fail. But if we allow such operations we can write code like:

```
oldenv:= currentenv;
x:= ∋* s;
s of oldenv:= s + t;
y:= ∋* nl;
```

To define the semantics of such cases unabiguously, we must assume that ∃* operates on a copy of the original s and not on s itself, otherwise there would be difficulty determining which elements of s had already been tried, and which elements remained to be chosen. The same rule holds if we have expressions such as x:= ∃*{y ∈ s|p(y)}. The set on the right-hand side is computed once, in the original environment before the first element is selected, and any subsequent changes to s or to variables in p do not affect the set of values which may be assigned to x. In general, ∃* will always use a copy of a set, although some form of copy optimization could probably be successful in avoiding unnecessary copying.

Example: Using these nondeterministic selection operators a solution to the 8-queens problem can be written as:

```
rows:= nult;  avail:={n, 1≤n≤8};
(1≤ ∀ n≤ 8) rows(n):= ∃* {x ∈(avail-{r(j),r∈ rows}-{r(j)-n,r∈ rows}
                                        - {r(j) + n, r∈ rows})};
    end ∀n;
print rows;
```

For each n a set of possible values for rows(n) is computed and one of these values is initially selected for rows(n). When, for some n, this set is empty or all of its elements have been tried and discarded, the previous value of n is restored and a new element of its set of possibilities is selected.

In many non-deterministic algorithms, such as some forms of heuristic search, the order in which elements are chosen is very important. Since the representation of a set, and thus the selection algorithm, is implementation-defined, it is not possible to control the order in which elements are selected. For this reason we allow ∃* to operate on tuples:

$$x:= \exists^* \langle v_1, v_2, \ldots, v_n \rangle;$$

This operator will select first v_1, then v_2, etc.

In SETL the existential quantifier $\exists$ is also a selection operator, and the semantics proposed for $\ni^*$ could be extended naturally to $\exists^*$, e.g., we can agree that $\exists^* x \in s \mid p(x)$ would be equivalent to $x := \ni^* \{y \in s \mid p(y)\}$.

The ok and fail primitives can be defined in terms of these non-deterministic selection operations by ok = $\ni^*$ <true,false> and fail = $\ni^*$ nl. Note that ok will behave exactly as the ok primitive did since no new environment will be opened when false, the last element, is selected, and also that ok = $\ni^*$ {true, false} would work equally well, but in this case the value false might be assigned to ok first.

We also observe concerning the non-deterministic selection operators that if $x := \ni^* \{exp_1, exp_2, \ldots, exp_n\}$ is executed, all of the exp_i are evaluated at once, with any side effects caused by the computations occurring immediately. In most cases not all of these n values will be needed, so some unnecessary, and potentially harmful, computations will be performed.

One possible solution would be to include the $\ni^*$ primitive in a language which had some type of delayed-evaluation, or 'call-by-need' semantics. The evaluation of $\{exp_1, exp_2, \ldots, exp_n\}$ would then yield a set of pointers to code for the evaluation of the exp_i. These code blocks would be executed only for those elements which were actually chosen as values for x. These computations would be performed in the old environment and side effects would occur where appropriate. Also, elements could be selected from infinite sets since the sets would not actually be computed. Instead, some form of generating function could be used to produce one element at a time, as needed.

Since these semantics are not possible in SETL, $\ni^*$ and $\exists^*$ are only available in the more restricted form.

IV SNOBOL pattern Matching Expressed Using Backtracking

The ok-fail primitives and environment operations are sufficient to implement an imitation of SNOBOL in SETL. This section describes such an implementation. A collection of

pattern primitives which access a set of global variables is provided. These primitives are functions which attempt to match a portion of the subject string and either return the successfully matched substring, or else execute fail. The global variables needed are:

- globstring = the subject string
- cursor = pointer to the rightmost character of globstring which has already been matched
- globenv = the environment which was active at the time that pattern matching began

Other global variables can be used to imitate other SNOBOL features, such as ANCHOR.

The code for functions corresponding to SNOBOL pattern primitives appears in the next section. Note that most of the functions are straightforward. The value of the cursor usually increases when a successful match is made. The most interesting patterns are those with implicit alternatives-*arbno*, *fence*, *arb*, *succeed* and *bal*. Each of these functions executes an instance of ok which introduces a new environment and provides alternative action to be taken if a subsequent failure should occur. Consider, for example the definition of arb:

```
(0 ≤ ∀n ≤ # globstring-cursor) if ok then len(n); end if;end ∀;fail;
```

arb initially returns the null string (i.e., the string of length 0). Should fail be executed at some subsequent point in the computation, the value of ok is false, the iteration continues with the value of n increased by 1, a new environment is generated, and arb returns a string of length n. When the subject string is exhausted, arb itself fails. Note that fail forces a return to the previous environment, causing the cursor to automatically resume the value it had prior to the unsuccessful match.

More complex patterns may be created by the operations of concatenation and alternation. Concatenation of patterns is represented simply by string concatenation. For example, the SNOBOL pattern

```
BREAK(' ') LEN(1) ARB
```

becomes

```
break(' ')||len(1)||arb()
```

Alternation is considerably more complicated. Ideally, one would like to be able to represent

```
P1 | P2 | . . . | PN                                    (*)
```

by

$$\ni^* \{p_1, p_2, \ldots, p_n\} \qquad (**)$$

where p_i is the code representing a pattern. However, because the entire set in (**) must be created at one time, $p_1, \ldots, p_n$, each of which contains function calls, must be evaluated immediately, in the current environment. However, to get a SNOBOL-like effect, the pattern matching functions p_j must be applied at precisely the appropriate times, and, therefore, the $\ni^*$ operation cannot be used in its full generality. Of course, if the semantics of SETL allowed evaluation of elements to be delayed until they are actually used, then each of $p_1, \ldots, p_n$ would be evaluated as necessary in a new environment, and the desired effect would be achieved.

One method of circumventing this problem is to restrict $p_1, \ldots, p_n$ to be parameterless functions, and then executing

$$\ni^* \langle p_1, p_2, \ldots, p_n \rangle .$$

Another method, which does not require this restriction, involves the use of ok and fail and encodes (*) as

```
If ok then p1; else if ok then p2; ... else if ok then pn;
else fail;
```

the macros

```
eo(p) = if ok then p;
 o(p) = else if ok then p;
oe    = else fail;
```

can be used to make this code more compact:

```
eo(p1) o(p2) . . . o(pn) oe;
```

SNOBOL has two assignment operators which can be used within patterns. The conditional assignment operator (.) performs an assignment only after the match has been successfully completed. In SETL all assignments are made as soon as they are encountered, but are 'undone' if the match is unsuccessful and fail is executed.

The SNOBOL immediate assignment operator ($) performs an assignment whether or not the pattern containing this assignment is successful. Upon failure, the target variable of the assignment retains its new value. This effect can be imitated in SETL by use of the environment *globenv*, which is known to be the ancestor of all environments generated within the pattern matching segment of the program. SNOBOL assignments such as

VALUE $ X

are coded in SETL as x of globenv:= value;
Note that if the match fails completely and the ancestor environment of globenv is re-entered, this value of x is not retained. If this value must be saved the user could, of course, have executed

x of ances:= value;

where ances is the value of the ancestor of globenv.

The code needed to match a pattern p against a string s is:

```
if ok then globenv:= currentenv;
           cursor:= 0;                                              }
           globstring:= s;                                          }(*)
           prefix:= if anchor then nulc else arb(); ]               }
               some expression involving p
           prune(globenv);                                          }
           success:= true;                                          }(**)
     else  success:= false; end if;                                 }
```

The initial ok is used to generate a new environment which is considered the 'global environment' of the indicated pattern matching segment of the program. Failure of this environment indicates complete failure of the match, and thus success is set to false when ok = false. Matching in the 'unanchored mode' (if the match fails, restart the match with the cursor set to 1, then 2, etc., until the match fails with the cursor initially at #globstring) is achieved by executing arb(), which first

matches the null string, then the first character of globstring and continues matching larger and larger initial portions of globstring, which are then assigned to prefix. At this point pattern matching begins and any code which includes pattern functions may be executed. Upon successful completion of all pattern-representing code, the intermediate environments are prune'd and the success flag is set to true. Otherwise, some fail must have caused ok to take on the value false and success is set accordingly. For convenience, the macros startmatch(s) and endmatch may be used in place of code fragments (*) and (**), respectively.

Examples:

1. The SNOBOL statement

```
STRING PATTERN = REPLEXP  :S(LABEL1)F(LABEL2)
```

is coded as

```
startmatch(string);
junk:= pattern;
string:= prefix || replexp || rem();
endmatch;
if success then go to label1; else go to label2;
```

2. STRING PATTERN . X :F(LABEL)

becomes

```
startmatch(string); x:= pattern; endmatch; if not success
then go to label;
```

V A Library of Primitive Pattern Functions

(1) ANY

```
definef any(string);
    if cursor = #globstring then fail;
    if ∃ s(i) ∈ string | globstring(cursor + 1) = s
        then cursor:= cursor + 1; return s;
        else fail;
end any;
```

(2) NOTANY

```
definef notany(string);
if cursor = #globstring then fail;
if ∃ s(i) ∈ string | globstring(cursor + 1) = s
     then fail;
     else cursor:= cursor + 1; return s;
end notany;
```

(3) LEN

```
definef len(n);
     if n = 0 then return nulc;
     if #globstring - cursor < n then fail;
          else return globstring(cursor + 1:(cursor:=(cursor+ n)));
end len;
```

(4) POS

```
definef pos(n);
     if cursor = n then return nulc; else fail;
end pos;
```

(5) RPOS

```
definef rpos(n);
     if cursor = #globstring - n then return nulc; else fail;
end rpos;
```

(6) TAB

```
definef tab(n);
     if n = cursor then return nulc;
     if n < cursor then fail;
          else return globstring(cursor+ 1: cursor:= n);
end tab;
```

(7) RTAB

```
     definef rtab(n);
     if place:= (#globstring - n) = cursor then return nulc;
     if place < cursor then fail;
          else return globstring(cursor + 1: cursor:=place);
end rtab;
```

(8) REM

```
definef rem();
if cursor = #globstring then return nulc;
     else return globstring(cursor + 1: cursor:= #globstring);
end rem;
```

(9) SPAN

```
definef span(string);
     if cursor = #globstring or not ∃ s(i) ∈ string |
                                        s = globstring(cursor + 1)
     then fail; $ span must match at least one character
     (cursor + 2 ≤ ∀i ≤#globstring)
     if not ∃ s(j) ∈ string | s = globstring(i)
          then return globstring(cursor + 1: cursor:=(i-1));
     end ∀i;
     return globstring(cursor + 1: cursor:= #globstring);
end span;
```

(10) BREAK

```
definef break(string);
     if cursor = #globstring or ∃ s(i) ∈ string |
                                     s = globstring(cursor + 1)
          then fail; $ break must match at least one character
     cursor + 2 ≤ ∀i ≤ #globstring)
          if ∃ s(j) ∈ string | s = globstring(i)
          then return globstring(cursor + 1: cursor:=(i-1));
     end ∀i;
     return globstring(cursor + 1: #globstring);
end break;
```

(11) EXACTLY

Any arbitrary string may be used as a pattern component in SNOBOL. In SETL one must explicitly invoke the function exactly which either matches its argument, or fails.

```
definef exactly(string);
     if cursor + #string > #globstring or
          globstring(cursor + 1: cursor:=(cursor + #string)) ≠
                                                        string
     then fail;
     else return string;
end exactly;
```

(12) FAIL

The SETL fail may be used exactly as the SNOBOL FAIL primitive is used.

(13) FENCE

```
definef fence();
    if not ok then fail(globenv); else return nulc;
end fence;
```

Any fail which is executed in the environment created by the ok in *fence* causes failure of the global environment and therefore terminates the match unsuccessfully.

(14) ABORT

Execution of fail(globenv) causes immediate termination of the current instance of pattern matching.

(15) ARB

```
definef arb();
    (0 ≤ ∀n ≤#globstring - cursor)
        if ok then return len(n);
    end ∀n;
    fail;
end arb;
```

(16) SUCCEED

```
definef succeed();
    (while true)
        if ok then return nulc;
    end while;
end succeed;
```

(17) ARBNO

Although the argument of the SNOBOL function ARBNO may be any pattern, in SETL the argument is restricted to bring a parameterless pattern function which either returns a matched substring of globstring, or fails. Under this constraint, the function is:

```
definef arbno(pat);
    if ok then return nulc;
        else return pat() || arbno(pat);
end arbno;
```

Example:

```
SNOBOL-- ARBNO(ANY('ABC') SPAN('XYZ'). X
SETL-- definef p(); return any('abc')||span('xyz'); end p;
      x:= arbno(p);
```

Another way by which the effect of ARBNO can be imitated is to execute the following code:

```
z;= nulc;
(while not ok)      z:= z||pat; end while;
/* next statement */
```

pat may be any pattern expression. This code violates the convention that the value of a pattern must be the substring which it matches. However, the value of z at the next statement is always the matched substring.

(18) BAL

bal can be implemented by using an auxiliary primitive function, *gbal*, which matches the shortest non-null string balanced with respect to parentheses.

```
definef gbal();
    if globstring(cursor + 1) = ')' then fail;
    paren:= 0;
    (cursor + 1 ≤ ∀i ≤ #globstring)
        if globstring(i) = '(' then paren:= paren + 1;
            else if globstring(i) = ')'
                then paren:= paren - 1;
                  if paren = 0 then return
                    globstring   (cursor + 1: cursor:= i);
    end ∀i;
    fail;
end gbal;
```

BAL can now be defined as:

```
definef bal();
    return gbal || arbno(gbal);
end bal;
```