Efficiency in Nondeterministic Control
through
Non-Forgetful Backtracking

UUCS-77-114

by

Gary Lindstrom
Department of Computer Science
University of Utah
Salt Lake City, Utah 84103

October 15, 1977

This work has been supported in part by the National Science Foundation
under grant DCR73-03441 A01 to the University of Pittsburgh

Abstract

Nondeterministic (ND) control has long been used to express elegant solutions to complex search problems. Programs using ND control can be executed on conventional machines through a systematic examination of trial execution paths. Among the many approaches to the enumeration of these paths is *backtracking*, a depth-first search of the execution path tree. Despite its implementational advantages, backtracking in its purest form suffers from a "forgetfulness" of retracted execution subpaths. This can lead to exponential run-time on problems such as top-down parsing in which the same subproblem can reoccur in slightly different global contexts.

This paper presents an alternative form of ND control implementation incorporating "non-forgetfulness" into backtracking. Reoccurrences of previously searched subgoals are detected and their net computational effects recreated on demand. Since each distinct goal is pursued at most once, search problems such as general top-down parsing run in polynomial time. Moreover, in contrast to an exhaustive, bottom-up approach, goals are only pursued if appropriate in some global context.

A strategy for non-forgetful backtracking is outlined in terms of coroutines and ordinary backtracking. The description of an alternative implementation of this strategy using simply coroutines is referenced. Top-down parsing is used to illustrate the application of this technique in both linguistic appearance and execution effect. Finally, some directions for further research into generalizations of these results are suggested.

"History does not repeat itself except in the minds of
those who do not know history."
- - - Kahlil Gibran

1. MOTIVATION

Nondeterministic (ND) control ([Ch75], [Fl67], [Jh67]) is a natural control strategy for a wide range of search applications including parsing, graph traversal, game playing, and enumeration problems. Under ND control, execution branches are automatically selected according to their ultimate correctness (toward reaching a desired goal state) rather than by locally available selection criteria. Thus ND control semantics assume the services of an oracle who guides the execution through uncertain branches while avoiding blind alleys.

Such mystical control semantics can be simulated on conventional machines through any one of a variety of interpretation schemes, each of which systematically examines trial paths within the tree of all possible execution sequences. Such schemes may be completely correct (e.g. breadth-first execution tree searching), partially correct (e.g. depth-first execution tree searching), or heuristic (e.g. best-first execution tree searching with a bounded candidate path list).

The most popular approach to ND control simulation is backtracking ([Br76], [GB65], [Hn76], [Kn75], [GY76]), a partially correct method. Under backtracking, the oracle of ND control is simulated by tentative, reversible continuations of the program's current execution. On failure, the net effect of each retracted subexecution is one bit of information: "Not this branch".

Full reversal of these exploratory subexecutions precludes any persisting benefit from subcomputations that may be needed again later. Yet in many backtracking search applications, identical subgoals frequently reoccur within slightly different global contexts (e.g. in chess, searching for moves from a particular board configuration when it results from two or more distinct move histories). This "forgetfulness" has given backtracking a reputation for slowness which we will here attempt to rehabilitate.

2. FORGETFULNESS IN SEARCHING

Forgetfulness in searching algorithms (we use "searching" here in a generic sense) may be defined as follows:

> a searching algorithm is forgetful if the second, and subsequent, searches for goal G in controlling state S each require as much time to complete as did the first such search.

By controlling state we mean "that combination of global data that is instrumental in the success or failure of this goal". We denote the task of searching for goal G in controlling state S as (G,S).

To eliminate forgetfulness, the following extensions must be made to a searching algorithm:

i) a correct (and, ideally, minimal) specification of the controlling state for each task;

ii) a method of logging the result of each successful search undertaken on a task (G,S) when that task is first attempted;

iii) a capability for recognizing reoccurrences of (G,S) tasks as they arise;

iv) given such a repeated task (G,S), a means for recreating on demand the net state change that resulted from each success originally found on (G,S), and, finally,

v) a means of authenticating any regenerated success, by summarizing its computation, should that local success contribute to a global success that is to be formally exhibited.

Clearly, one method of achieving non-forgetfulness is through exhaustive, bottom-up searching with the aid of a global table recording all successes on each task ([Br76]). Such an approach suffers from the following drawbacks:

i) the overall control strategy is not data-driven in that goals are pursued independent of any global context guaranteeing their relevancy to the particular data at hand, and

ii) overt "data engineering" must be done in the maintenance of the global table, whereas under top-down searching such information is elegantly distributed throughout the local variables of the active search processes.

Our plan here is to present an adaptation of searching which incorporates both the global strategy of backtracking and the non-forgetfulness of bottom-up searching. Alternatively stated, we wish to "memoize" ([Mc68], [Mr70]) individual search functions operating within a backtracking regime. (Gaschnig [Gs77] and Friedman et. al. [FWW76] have also studied this problem.)

After our general approach is addressed in the next section, a particular linguistic setting will be introduced in section 4. Our method will then be illustrated in this setting using top-down parsing as a sample application (section 5). Finally, some directions for further research suggested by these results are offered in the concluding section.

3. NON-FORGETFULNESS IN BACKTRACKING

Our approach to non-forgetful backtracking is based on the following interpretations to the extensions cited above:

i) controlling state: we require the programmer to specify for each goal G the set S of global variables controlling that search.

ii) success logging: whenever a search is undertaken on a previously unsearched task (G,S) (an "original" search), the net result of each success on that task is recorded in a table local to the original searcher before that success is reported upward. The "net result" consists of the resulting values of a set A of global variables encapsulating the search effect. Like the set S, the set A is specified by the programmer.

iii) repeated task recognition: we assume the existence of an associative memory in which we store the names of original searchers for each (G,S) key as they are created in the course of the overall search logic.

iv) recreating successes: when a repeated task (G,S) arises, the original searcher for (G,S) is reactivated and put into regeneration mode.

The net result of each previously recorded success on (G,S) is recreated (via direct assignments to the variables in A), until the success set is exhausted. Failure is then reported, and the original searcher suspends awaiting a new regeneration request whenever (G,S) next arises.

V) success authentication: we make the simplifying assumption here that the A set for each task contains a global variable that accumulates salient information summarizing each local success. Thus no further action by the searchers is needed on the occasion of a global success, for that global variable already summarizes at that time the contribution of each participating searcher. (In a more general framework, one may wish special actions to be taken by each searcher contributing to a global success when that success is to be authenticated.)

Figure 1 summarizes this logic as applied to an individual search routine. This logic relies on a fundamental property of backtracking that is important to our scheme:

> any problem suitable for backtracking must assure the exhaustive completion of any search task (G,S) before any instance of (G,S) can arise again (otherwise, the search process would be potentially infinite under the depth-first strategy of backtracking).

Two beneficial implications of this fact are capitalized upon in our scheme:

i) any original searcher for (G,S) will search to completion before any regeneration requests on (G,S) can arise, and

ii) any regeneration requests for (G,S) will be serviced thoroughly by its original searcher before any subsequent regeneration requests for (G,S) can arise.

4. A LINGUISTIC SETTING

From the logic of figure 1, it is clear that coroutines are an essential ingredient in any implementation of non-forgetful backtracking. In fact, coroutines alone suffice if the programmer is willing to manage explicitly the state saving and restoration inherent in backtracking (see [Ln76]).

However, if both ND control (with systematic trial executions) and coroutine control (for suspending and reactivating original searchers) are available, a much cleaner implementation of this strategy can be obtained.

To provide such a setting for our sample application in the next section, we present here a PASCAL extension including both SIMULA-like coroutines and primitives for ND control. (Further discussion of the merits of such a control combination may be found in [Ln77a].)

4.1 Coroutine control extensions.

The coroutine manipulation facilities of our PASCAL extension have been selected from those found in Coroutine PASCAL [Lm76]. We will need the following primitives:

i) the data type *ref*, which is the set of names of dynamically created coroutine instances;

ii) the function *CREATE(<procedure call>)*, which dynamically creates a new coroutine instance of the given procedure. Parameters are evaluated and bound, but execution of the instance does not yet commence. A value of type *ref* referring to the created instance is returned as the value of this call on *CREATE* (and is available as *SELF* within the coroutine);

iii) the function *CALL(<ref exp>)*, which passes control to the coroutine instance referred to by the given expression. If that instance is newly created, it begins execution at its first statement. If the instance currently is *DETACH*ed (see (iv)), it resumes following the statement that caused that *DETACH*ment;

iv) the procedure *DETACH*, which suspends the most tightly surrounding coroutine instance (in the sense of *CALL/DETACH* nesting), and returns control to its most recent *CALL*er, with control resuming just following the statement doing that call, and

v) the procedure *TERMINATE* (equivalent to exiting from the code body of the most tightly surrounding coroutine instance), similar to *DETACH* except that the coroutine instance is no longer *CALL*able.

4.2 Nondeterministic control.

Of the many linguistic formulations of backtracking and ND control that have appeared in the literature (e.g. [Jh67], [Ch75], and [Hn76]), we find the early work of Floyd [Fl67] to offer the best basis for our needs here. Our particular ND primitives, defined in terms of their effect under ordinary backtracking execution, are:

i) the function *NDCREATE(<procedure call>)*, which creates a coroutine instance of the given procedure operating as an independent ND system. This means:

   a) the instance may be manipulated (e.g. *CALL*ed and *DETACH*ed) as an ordinary coroutine instance, but in addition:

   b) one may use the special ND control primitives *CHOICE* and *FAILURE* within its dynamic scope.

ii) the function *CHOICE(<exp>)*, delivering successive integer values from 1 to the value of *<exp>*, in that order, and

iii) the function *FAILURE*, signalling detection of a blind alley. This causes the following backtracking actions to occur in the most tightly surrounding ND system:

   a) the ND system's control state is reset to that in effect at the time of the most recently executed *CHOICE* call within the dynamic scope of that system. If at least one value remains to be generated by that *CHOICE* operation, a new value is selected for generation. Otherwise, if that *CHOICE* operation is exhausted, then the system's control state is reset to that associated with the next most recent *CHOICE* operation, etc. If all previous *CHOICE* operations in this system have been exhausted, then a *FAILURE* is done in the dynamically surrounding ND system, if it exists. Otherwise the current system simply terminates.

   b) the <u>local</u> date state of the selected ND system (i.e. the set of all variables created within its dynamic scope)

is reset to that associated with the selected *CHOICE* point. Note that variables outside the ND system are left unchanged by a *FAILURE* action.

c) rule (b) notwithstanding, a programmer may declare selected variables within a ND system to be nonreset upon *FAILURE* via the *VAR* prefix *NONRESET*.

4.3 Combining coroutines and ND control.

The primitives of these two control regimes may be mixed in any semantically meaningful execution sequence. While a complete specification of the semantics of such a mixed usage is beyond the scope of this paper, we will simply observe the following useful facts:

i) ND systems can be created within one another. However, as long as no *DETACH*es are used within them, the *FAILURE* effects are as though only one overall system were created.

ii) If a ND system becomes *DETACH*ed and is reactivated by a *CALL* from another ND system, then its subsequent ND control actions (*CHOICE* and *FAILURE*) have the same effect as they would had its new caller been its initial caller.

iii) Finally, the control sequence {*DETACH*, *FAILURE* (in the dynamically surrounding system)} is a useful control combination that cannot conveniently be programmed due to the context change immediately following the *DETACH*. Consequently, we assume the availability of a special command *FAILDETACH* performing this two-step action.

5. NON-FORGETFULNESS IN A TOP-DOWN PARSER

We will now illustrate our notion of generalized backtracking through the familiar problem of fully general top-down parsing.

Top-down parsing is attractive for our purposes here because:

i) it exemplifies rule-driven searching, a familiar programming paradigm;

ii) it is a well-understood process;

iii) it is computationally non-trivial (involving true backtracking not directly expressible by ordinary recursion [Ln77b]), and

iv) its "forgetfulness" leads to a dramatic decline in speed (i.e. exponential run time) for certain grammars and string sequences.

5.1 Floyd's top-down parser.

Floyd [F164] has elegantly formulated an approach to top-down parsing in ND control terms. That approach, cast into our PASCAL in figure 2, assumes the grammar has been put into the following normal form:

a) the non-terminal symbols are taken from the upper case alphabet $\{A, \ldots, Z\}$;

b) the terminal symbols are taken from the lower case alphabet $\{a, \ldots, z\}$, and

c) the grammar is non-left-recursive (this restriction can be eliminated by a variety of methods all complicating exposition), and

d) there is only one rule for each nonterminal symbol α, and that rule obeys one of the following three forms:

(alternation) $\alpha \to \beta_1 | \beta_2$, with β_1 and β_2 nonterminals;

(concatenation) $\alpha \to \beta_1\beta_2$, with β_1 and β_2 nonterminals, or

(terminal) $\alpha \to \tau$, with τ a terminal symbol.

5.2 Floyd's algorithm in non-forgetful form.

Floyd's top-down parsing algorithm may be recast into our non-forgetful backtracking framework by the following interpretations on figure 1:

i) A controlling state is simply the current global string pointer *ptr* (since the string *str* itself is constant). Goals, of course, are non-terminal symbols. Thus a task is a pair (G,*ptr*).

ii) The associative memory for retrieving the name of the original searcher on a task (G,*ptr*) is simply a table indexed by G and *ptr*, since each is drawn from a compact range.

iii) The net state change resulting from a successful search on a task (G,*ptr*) is simply the *ptr* value marking the end of the spanned substring, along with the *p* value pointing to the constructed subtree.

iv) Instituting a net state change amounts to simply setting *ptr* and *p* to those values (string position beyond end of spanned substring, and root of spanning subtree, respectively) saved under that success;

v) Authentication of a global success is accomplished by printing (via *printtree*) the overall parse tree as previously constructed incrementally.

With these modifications, Floyd's parser now has the following properties:

i) the underlying search strategy is unchanged, with each distinct new goal arising in the same order as before;

ii) when a repeated goal occurs, each success found originally for that goal is simulated, one at a time, by direct assignment to *ptr* and *p* in time independent of the complexity of the spanning subtree;

iii) when the BOSS routine detects à global success, then the subtrees associated with each participating subparse are outputted by traversal in linear time, and

iv) the parser accommodates the general case of regenerated successes themselves involving regenerated successes at lower levels, with proper subtree outputting at all levels.

5.3 An implementation.

Figure 3 gives code for Floyd's parser converted to non-forgetful form. Notice that neither BOSS nor SUBORD are altered in any way other than to replace calls of the form:

SUBORD(G)

with calls of the form:

CALL(NDCREATE(SEARCH(G))).

This ensures two salutary effects:

i) calls on SUBORD are now systematically done through SEARCH, which does success logging and regeneration, and

ii) each SEARCH routine is a separate ND subsystem, so it can be *DETACH*ed and re-*CALL*ed when success regeneration must be done for the same task in a subsequent context.

It is particularly interesting to note how *CHOICE* is used in three novel ways within SEARCH:

<u>first usage</u> ("*CHOICE*(2)=1"): to intercept the final *FAILURE* done by its original SUBORD searcher;

<u>second usage</u> ("*CHOICE*(2)=2"): to intercept the exhaustion of success regeneration, and

<u>third usage</u> ("*RESTOREGLOBALS(CHOICE(*n))"): to select each saved original success in turn for regeneration. This greatly simplifies the coding of SEARCH and renders uniform its interface with its *CALL*er in both original search and regeneration modes.

Figures 4 and 5 illustrate the revised parser's operation on a sample grammar and input string.

5.4 <u>Parser performance</u>.

Our non-forgetful top-down parser possesses the following desirable characteristics:

i) full generality, including exhaustive parsing on ambiguous strings;

ii) distributed (i.e. non-global) parse state representation, with local success data associated with each original task searcher;

iii) a top-down strategy that attempts only globally plausible subparses;

iv) polynomial run time, and

v) tree outputting by direct traversal in time proportional to tree size.

Time behavior is as follows. Denote the number of non-terminals in our grammar by $|G|$. Let n be the length of the input string, and b its degree of ambiguity. Then the total time spend doing searching (exclusively on original tasks) is bounded by a number proportional to the maximum number of original

successes, i.e. (number of substrings) x (number of goals) x (ambiguity), or

$$OS(n) = \frac{n^2}{2} |G| b$$

The space required can be estimated as follows. Clearly, the global task array is of size $|G|$ n. Overall, the sum of D table sizes is proportional to the number of original successes, OS(n). Moreover, the heap space required to represent all retained parse subtrees is also proportional to OS(n), for the local path length of each such success is 3 or less.

Thus both space and time behavior for the non-forgetful parser are of order OS(n). Since this bound represents the minimum amount of time that could be consumed by an exhaustive parser, "non-forgetfulness" must in fact be attained despite the reoccurrences of subtasks under the global top-down strategy.

6. CONCLUSION AND FUTURE WORK

This paper has presented a fundamental notion of non-forgetfulness in backtracking along with its illustration through a particular case study. The results encourage further research to bring this technique into wider explicit use in general programming. Areas suggested include:

i) formalization of this method into general linguistic primitives suitable for application in any "standard" backtracking situation;

ii) extension of non-forgetfulness to search strategies beyond classical backtracking, where original searching and regeneration are not locally disjoint phases (e.g. a dynamic, incremental alpha-beta pruning search for moves in games);

iii) further study of the controlling state notion, aimed at methods of minimizing such states and readily recognizing their reoccurrence in general, and

iv) analysis of the empirical impact this particular control regime has on individual control implementation strategies, expecially in the area of storage management [PSW72], [SE73]. Some preliminary work of this kind may be found in [SL76].

References

[Br76] Berry, G., "Bottom-up computation of recursive programs", Revue Francaise d'Automatique, Informatique, Recherche Operationnelle 10,3 (Mar. 1976) 47-82.

[Ch75] Cohen, Jacques, "Interpretation of non-deterministic algorithms in higher-level languages," Inf. Proc. Ltrs. 3,4 (March 1975) 104-109.

[Fl64] Floyd, R.W., "Syntax of programming languages: a survey," IEEE PGEC 4 (1964), p. 346. Also in Rosen, Programming Languages and Systems, McGraw-Hill.

[Fl67] Floyd, R.W., "Nondeterministic algorithms," JACM 14,4 (Oct. 1967) 636-644.

[FWW76] Friedman, Daniel P., David S. Wise, and Mitchell Wand, "Recursive programming through table look-up," Tech. Rpt. 45, Indiana Univ. Computer Science Dept. (March 1976).

[Gs77] Gaschnig, John, "A general backtrack algorithm that eliminates most redundant tests", Proc. IJCAI-77, Boston (Aug. 1977) p. 457.

[GY76] Gerhart, Susan L., and Lawrence Yelowitz, "Control structure abstractions of the backtracking programming technique", IEEE Trans. Soft. Eng. (Dec. 1976).

[GB65] Golomb, S. W. and L. D. Baumert, "Backtrack programming," JACM 12 (1965), 516-524.

[Hn76] Hanson, David R., "A procedure mechanism for backtrack programming," Proc. ACM Annual Conf. (Oct. 20-22, 1976), Houston, Texas, 401-405.

[Jh67] Johansen, Peter, "Non-deterministic programming," BIT 7 (1967) 289-304.

[Kn75] Knuth, D., "Estimating the efficiency of backtrack programs", Math. of Comp. 29-129 (Jan. 1975) 121-136.

[Lm76] Lemon, Michael, "Coroutine PASCAL: a case study in separable control," M.S. thesis, Tech. Report 76-13, Dept. of C.S., Univ. of Pittsburgh (Dec. 15, 1976). 68 pp.

[Ln76] Lindstrom, Gary, "Non-forgetful backtracking: an advanced coroutine application," Tech. Report 76-8, Dept. of C.S., Univ. of Pittsburgh (Dec. 6, 1976) 42 pp.

[Ln77a] Lindstrom, Gary, "Backtracking in generalized control settings", Technical Report UUCS 77-105, Dept. of Computer Science, Univ. of Utah (July 6, 1977).

[Ln77b] Lindstrom, Gary, "Control structure aptness: a case study using top-down parsing," Dept. of Computer Science, Univ. of Utah (July 18, 1977) 26 pp.

[Mr70] Marsh, David, "Memo functions, the Graph Traverser, and a simple control situation," Machine Intelligence 5, pp. 281-300. Meltzer, B. & D. Michie, eds., New York: Am. Elsevier (1970).

[Mc68] Michie, D., "'Memo' functions and machine learning," Nature 218 (1968) 19-22.

[PSW72] Prenner, Charles J., Jay M. Spitzen, and Ben Wegbreit, "An implementation of backtracking for programming languages," Proc. ACM Nat'l. Conf. (1972), 763-771.

[SE73] Smith, D.C. and H.J. Enea, "Backtracking in MLISP2," Proc. IJCAI-73, Stanford (1973).

[SL76] Soffa, Mary Lou, and Gary Lindstrom, "Describing and testing generalized control regimes through implementation modeling", Univ. of Pittsburgh C. S. Dept. Tech. Rpt. 76-11 (December 1976).

```
procedure search(G,S,A);
    {G is given goal;
     S is controlling state;
     A is set of globals altered}
begin if we have performed (G,S) before then
            pass request on to original (G,S) searcher
    else
    begin preserve values of globals specified in set A;
          do search on G;
          while successful do
          begin save values of globals specified in set A;
                suspend until next success is requested
                continue search
          end;
          repeat restore initial values of globals in set A;
                report failure and suspend;
                {now have regeneration request}
                while saved successes remain do
                begin restore A value set for this success;
                      suspend until next success is requested
                      move to next saved success
                end
          until false {original searchers never die}
    end
end {search}
```

Figure 1. General strategy for non-forgetful searching.

```
{Floyd's parsing algorithm in conventional ND form}

const   strmax = 50;                {maximum string length}

type    ptrval = 1..strmax;         {range of string pointers}
        symb = 'a' .. 'Z';          {vocabulary of grammar}
        ntsymb = 'A' .. 'Z';        {nonterminal symbols}
        termsymb = 'a' .. 'z';      {terminal symbols}
        pcellptr = ↑pcell;          {pointer to print tree cell}
        valtype = (locval,subref);  {tags on print tree cells}
        pcell = record link: pcellptr;              {link to next cell}
                       case valtag: valtype of
                            locval: (val: ntsymb);  {subtree root label}
                            subref: (ptr: pcellptr) {pointer to subtrees}
                end;

var     rule: array [ntsymb,1..2] of symb;          {rules of grammar}
        ruletype: array [ntsymb] of (alt,conc,term); {type of each rule}
        ptrlim: ptrval;                             {length of test string}
        root: ntsymb;                               {root of grammar}
        p: pcellptr;                                {root of print tree}
        str: array [ptrval] of termsymb;            {test string}

procedure printtree(p: pcellptr; d: integer);  {print tree p indented d levels}

var     i: integer;

begin   case p↑.valtag of
             locval:  begin for i:=1 to d do write("  "); {indent}
                            writeln(p↑.val)
                      end;
             subref:  begin printtree(p↑.link,d);
                            printtree(p↑.ptr,d+1)
                      end
        end {case}
end {printtree};

procedure splice(var result: pcellptr; v,w: pcellptr);  {add subtree ref cell}

begin   new(result,subref);
        result↑.ptr:=v;  result↑.link:=w
end {splice};

procedure output(G: ntsymb);  {add subtree root cell}

begin   new(p,locval);
        p↑.link:=nil;  p↑.val:=G
end {output};
```

```
procedure boss;   {overall supervisor of parsing process}

var     ptr: ptrval;   {parser's pointer into test string}

        procedure subord(G: ntsymb);   {general rule-driven searcher}

        var     psave: pcellptr;

        begin output(G); psave:=p;
                case ruletype[G] of
                     alt:   begin subord(rule[G,choice(2)]);
                                  splice(p,p,psave)
                            end {alt case};
                     conc:  begin subord(rule[G,1]);
                                  splice(psave,p,psave);
                                  subord(rule[G,2]);
                                  splice(p,p,psave)
                            end {conc case};
                     term:  if ptr<=ptrlim and str[ptr]=rule[G,1] then
                                  ptr:=ptr+1
                            else failure
                end {case}
        end {subord};

begin {body of boss}
        ptr:=1;
        subord(root);
        if ptr=ptrlim+1 then printtree(p,0);
        failure {get exhaustive list of all parses}
end {boss};

begin   {main program}
        {read rules, root, string, and ptrlim}
        call(ndcreate(boss))
end. {program}
```

Figure 2. Floyd's top-down parser in conventional ND form.

(added to var section of main program:)

```
task: array [ntsymb,ptrval] of ref;   {initialized to nil values}
```

(in BOSS and SUBORD, each call subord(α) *changed to:)*

```
call(ndcreate(search(α)))
```

(code added for new procedure:)

```
procedure search(G: ntsymb);

const sucmax = 10;                      {maximum number of local successes}

type  Avals = record p: pcellptr;      {A set values for subord}
                     ptr: ptrval
              end;
      sucnr = 0 .. sucmax;              {local success serial numbers}

nonreset var n: sucnr;                          {local success counter}
             D: array [sucnr] of Avals;         {array of A set values}

      procedure saveglobals(n: sucnr);

      begin {save current A set values under success name n}
            D[n].p:=p;  D[n].ptr:=ptr
      end {saveglobals};

      procedure restoreglobals(n: sucnr);

      begin {restore A set values associated with success name n}
            p:=D[n].p;  ptr:=D[n].ptr
      end {restoreglobals};

begin if task[G,ptr]≠nil then  {(G,ptr) searched before}
            call(task[G,ptr])  {ask for success regenerations & pass back}
      else {have original search instance}
      if choice(2)=1 then {log successes on original search}
      begin task[G,ptr]:=self;         {enter name of searcher under (G,ptr)}
            n:=0;  saveglobals(n);     {save A set values on entry}
            subord(G);                 {call subord for actual searching}
            n:=n+1;  saveglobals(n)    {must have new success, so save it}
      end
      else {have intercepted final failure within subord call}
      repeat restoreglobals(0);  {restore A set values from initial entry}
            faildetach;          {report failure & suspend}
            {now have regeneration request from new searcher on (G,ptr)}
      until choice(2)=2; {cycle back to faildetach when regens stop}
      restoreglobals(choice(n))  {pick a success & return to new caller}
end {search}
```

Figure 3. Modifications to obtain non-forgetful parser.

Sample grammar:

$R \to Y \mid Z$ (alternation)

$Y \to Z\ X$ (concatenation)

$X \to R\ R$ (concatenation)

$Z \to a$ (terminal)

Trees produced for sample string $a\ a\ a\ a\ a$:

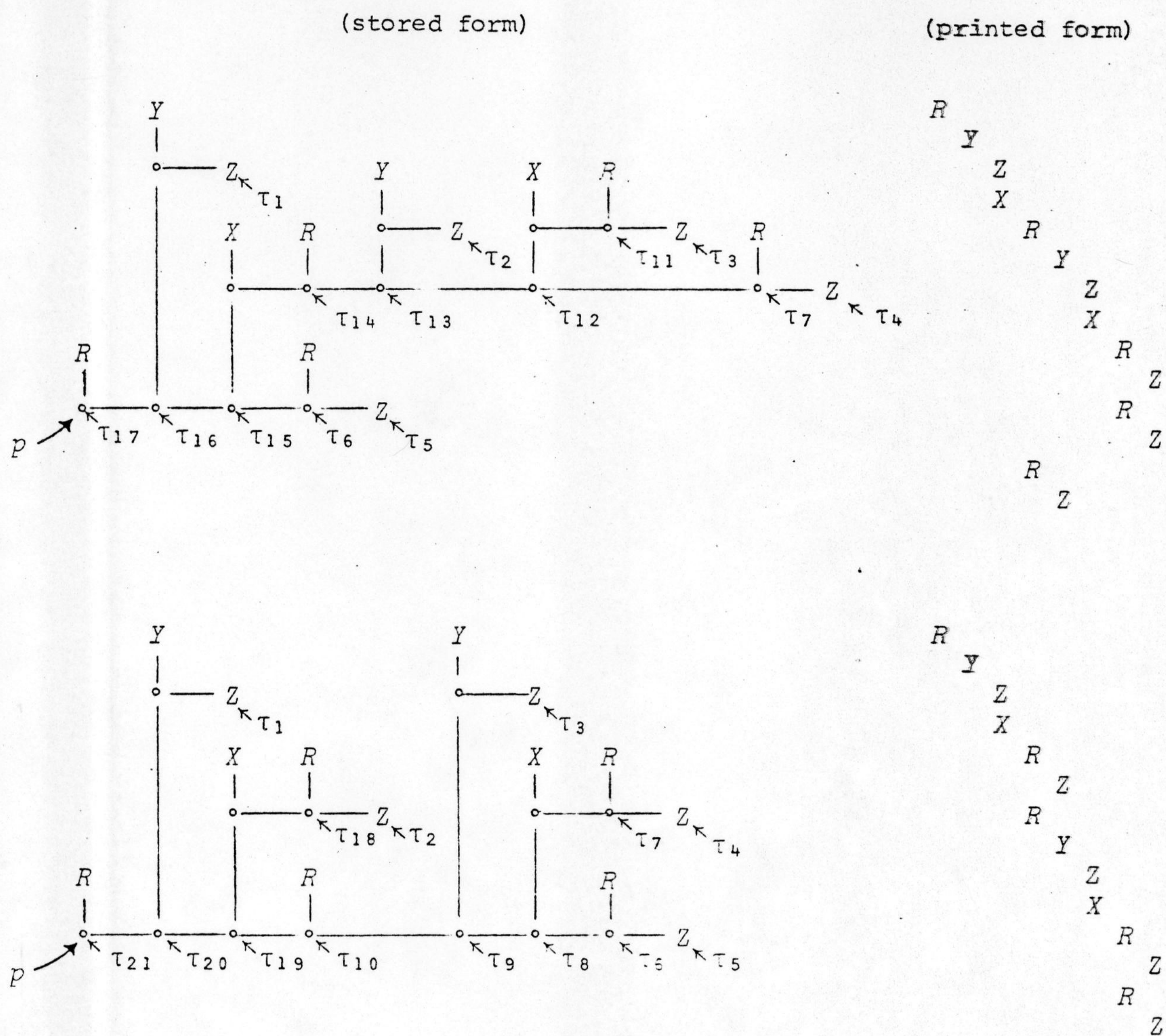

Figure 4. Sample results from non-forgetful parser.

	1	2	3	4	5
R	R τ_{16} τ_{20} τ_{23} τ_1 p: τ_{17} τ_{21} τ_{24} τ_{25} ptr: 6 6 4 2	R τ_{13} τ_2 τ_{14} τ_{18} 5 3	R τ_9 τ_3 τ_{10} τ_{11} 6 4	R τ_4 τ_7 5	R τ_5 τ_6 6
Y	Y τ_1 τ_{15} τ_{19} τ_{22} τ_{16} τ_{20} τ_{23} 6 6 4	Y τ_2 τ_{12} τ_{13} 5	Y τ_3 τ_8 τ_9 6	*(none)*	*(none)*
X	*(untried)*	X τ_{14} τ_{18} τ_6 τ_{10} τ_{11} τ_{15} τ_{19} τ_{22} 6 6 4	X τ_{11} τ_7 τ_{12} 5	X τ_7 τ_6 τ_8 6	*(none)*
Z	Z τ_1 2	Z τ_2 3	Z τ_3 4	Z τ_4 5	Z τ_5 6

Figure 5. Task table for figure 4 example, showing saved global values for each success.