SETL Newsletter # 155

J.T. Schwartz
October 7, 1975

Intermediate Result Recording and Other Techniques for Optimizing Recursions and Backtrack Programs.

1. General Explanation of the 'Memo Function' Technique.

Recursive routines will sometimes calculate required intermediate results repeatedly, and this inefficiency can have catastrophic effects on their performance. A well-known instance of this is the Fibonacci routine, which if written recursively as

```
definef fibon(n);  /* n-th Fibonacci number F_n */
return if n ∈ {1,2} then 1 else fibon(n-1) + fibon(n-2);
end fibon;
```

requires approximately 2^n steps to calculate F_n. This loss of efficiency can be avoided, at the cost of carrying more data, by recording intermediate results as they are calculated. In the preceeding example, this can be done by introducing a map *gotfib*, initialised to null, and by revising the preceeding algorithm as follows:

```
definef fibon(n);  /* improved Fibonacci calculation */
if gotfib(n) is val ne Ω then return val;;
/* if the value desired is not available, it must be calculated */
gotfib(n) = (if n ∈ {1,2} then 1 else fibon(n-1) + fibon(n-2))
                                                       is val;
return val;
end fibon;
```

This improved algorithm requires only cn steps to calculate F_n. The transformation (which we shall call *transformation by intermediate result recording*) that leads from the first of the two preceding functions to the second can clearly be applied mechanically.

One can indicate that it has been applied simply by attaching the keyword remember to the header line of the function, i.e., by writing exactly the first algorithm with the single change

definef fibon(n) remember; /* n-th Fibonnacci number F_n */

in its first line.

Transformation by intermediate result recording can actually be applied more generally than simply to functions free of side effects. To do this, one can, e.g., introduce general remember blocks having the form

```
remember;
    block /*this can be any block of code */
end remember;
```

The semantics of this construction are as follows: on entry to the block, the value of every variable that will be used in it is saved. Let *infoin* denote all this information. When the block is exited, either by a jump or by a normal exit, the final values of all variables changed within the block are recorded, and the target point of the exit jump or transition is recorded, also. Let *infout* denote all this information. A map *blockeffect* should be associated with each remember block RB; *blockeffect* will be initialised to null, and each time RB is exited *blockeffect* will be extended by executing blockeffect(infoin) = infout. Subsequently, before entering RB, one can construct *infoin*, retrieve the value of infout = blockeffect(infoin), and if *infout* is not Ω, simply execute a series of assignments that modify all variables in the same way that RB would modify them, and then jump to the same exit that RB would ultimately take.

The technique just outlined applies to both deterministic and nondeterministic programs. Let us agree to support non-terminism by adding the two customary primitives <u>ok</u> and <u>fail</u> to SETL (cf. SETL newsletter 153). Then in the first place, the *blockeffect* mappings associated with the various <u>remember</u> blocks RB in a program become multiple-valued (since one may enter RB several times, always with the same relevant entry information package *infoin*, and exit with several different exit information packages *infout*, which will reflect the several different sequences of values that can be supplied at the (dynamic) occurrences of <u>ok</u> within RB (and within procedures called within RB)). To take account of this situation, it is appropriate to associate an environment-valued subsidiary mapping *lastenvironment(infoin)* with RB (here we use the term *environment*, short for *data environment* (of a process), in the sense explained in Newsletter 153). The value e = lastenviroment(infoin) is the environment saved at the last evaluation of <u>ok</u> (which, since it leads to the saving of an environment, must yield the value <u>true</u>) before exit from RB, and after RB has been entered with *infoin* as its (relevant) entry information package. Suppose that we let

$$\text{weld (environment}_1\text{, environment}_2\text{, RB)}$$

designate the operation of modifying environment$_1$ by giving every variable (including the program instruction location counter) which is accessed within RB the value which it has in environment$_2$ (while other variables retain their environment$_1$ values). Let *perform (infout)* designate the operation of executing a series of assignments in the manner designated by one of RB's exit information packages *infout*, and then of transferring (past RB) to the RB-exit point that *inout* specifies. (As explained above, this duplicates the effect which actual execution of RB would have.)

Let *enter (environment$_1$)* designate the operation of transferring control to *environment$_1$*, and then letting execution proceed (at least until the next dynamic occurrence of an ok or fail.) Then entry to RB can be handled as follows:

(a) Let the relevant part of the environment at the moment of entry to RB be *infoin*. As long as there exist untried exit information packages *infout* belonging to *blockeffect(infoin)*, *perform*. A new exit information package will be *performed* whenever, as a consequence of a failure, control is return to the point (immediately before entry to RB) at which an unsucessful prior exit information package was tried.

(b) If all prior information packages have been tried, and if *lastenvironment(infoin)* is not Ω, then, immediately before entry to RB, generate

newenv = weld(currentenv,lastenvironment(infoin),RB)

as a descendent of *currentenv* (the currently executing environment) and *enter(newenv)*. Subsequently, when exit from RB occurs, we add a description of the exit status of all variables mentioned in RB to (the end of) *blockeffect(infoin)*, and also assign the parent environment p of the exit environment as the new value of *lastenvironment(infoin)*.

It is also reasonable either to attach an implicit remember action to each occurrence of ok in a nondeterministic program, or to allow such an action to be attached explicitly to selected occurrences of ok, perhaps by writing ok remember instead of ok. Then the parts of the data environment relevant to the code following an ok can be remembered when the ok is first executed in a set S attached to the ok (call this set *failed*). If this set is available, then before an ok is executed, its current data environment E can be tested for membership in *failed*. If E ∈ failed, then ok can return the value false instead of the customary true.

This technique suppresses both re-exploration of known dead ends and attempts to re-enter an environment that is already under exploration. Thus, for example, it allows a solution of the well-known 'buckets and wells' problem to be written in the following nicely 'rubbleized' form:

```
/* the following 'main' program is standard, and could be implicit */
(while ok  and v ne target) improve(v );;
                                  /* in the particular problem at hand,*/
                                  /* v is a vector of bucket contents  */
if v ne target then fail;;
/* now we use the fact that in a nondeterministic language, */
/* many routines all with the same name can be employed.    */
define improve(v);
(1 ≤ ∀n ≤ #v)
    if ok then
        v(n) = 0;           print 'pourout', n;
    else if ok then
        v(n) = C(n); /* C is a global vector giving the  */
                               /*capacities of the various buckets */
    print 'fillup', n;
    end if;
end ∀n;
return;
end improve;
define improve(v)
if 1 ≤ ∃n ≤ # v, 1 ≤ ∃m ≤ # v | ok then
    v(n) = (v(n) + v(m)) min C(n);
    v(m) = (v(m) - C(n) + v(n)) max 0;
    print 'pour', m, 'into', n;
end if;
return;
end improve;
```

Artifical intelligence algorithms which use heuristics are useful if they succeed with high probability (even if they do not work in every case). The following trick is available for handling algorithms of this class. At occurrences of ok, save a hash of the relevant data environment (e.g., a 64-bit hash) rather than a complete copy of the environment. Before executing the ok, recalculate this hash h, and check it for identity with some previously saved hash. If h occurred previously, let the ok return false rather than the customary initial true. If we use this trick, which can save very large amounts of memory, then calculations that might have succeeded will fail occasionally, but only very rarely.

A variant of this technique, suggested by M. Harrison,, is to discard 'old' hashes, e.g., by throwing away the oldest, i.e., least recently consulted element on each hash chain that grows to be more than one or two elements long. This latter variant is somewhat more reliable (i.e., likely to succeed if success is possible) than the technique suggested in the preceding paragraph, but will of course make certain computations that the former technique would avoid. In some cases, e.g., in the 'Fibonacci' example with which we begin, it will reduce the amount of required calculation very greatly. In other cases, e.g., in calculating the function defined recursively by

$$f(n) = f(n-1) + \ldots + f(2) + f(1),$$

it is clear that keeping a complete record of prior function values will be much more effective than keeping a partial record.

In many cases it will be inappropriate to save every detail of the (relevant portion of) the data environment at entry to a remember block or occurrence of ok, since there may be some mathematical reason, too deep for a compiler to discover, which ensures that only some limited aspect of the environment is relevant to the outcome of the block or to the success of the ok.

(E.g., important problem symmetries or localisations may be invisible to a compiler). For handling such situations, one may wish to allow remember to have a parameter, i.e. to allow the optimal form remember e, where e is a SETL value summarising all the environmental information which it is necessary to retain. Then *blockeffect* can be a function of e rather than of the whole (formally relevant) data environment on block' entry.

In nondeterministic situations allowing a heuristic treatment, one can retain a condensed hash of e rather than a full representation of e if e occurs in the context ok remember e, i.e., if e is associated with an occurrence of ok. As a matter of fact, a modification of this same technique can be used even if the inaccuracies implied by a heuristic treatment are not acceptable. The technique as modified can appropriately be called *statistical steering*, and has the following description:

(i) Associate a one-parameter mapping *pastexperience* with every occurrence of ok remember and ok remember e in a program P. The parameter h of this mapping is an appropriately defined hash of e (or in the case of a simple ok remember of the (relevant) data environment of the ok) The value of *pastexperience* is a pair of integers n_1, n_2, which respectively give the number of times (in the prior execution of P) that the same instance of ok has occurred (dynamically), with the same environment hash parameter h, and has been given the respective values true and false.

(ii) To assign a value to a newly encountered instance of ok, calculate its hash, and retrieve $<n_1,n_2>$ = pastexperience(h). If $n_2 > n_1$ try the value false for ok first; otherwise try true first. (Here note that an environment must be saved at the ok no matter what value of ok is tried first).

The tables of values *pastexperience(h)* accumulated during successive runs of a given program P might also be allowed to cumulate, which can improve the performance of P as it is used over a period of time.

If one tries to use the trick of saving only a hash h of the relevant environment e at the start of remember blocks which are not cases of ok remember, one runs the risk of having a program execute (rather than simply failing unnecessarily) but of executing wrongly. In such a case the program can actually come to a normal termination and print output, but the output can be wrong. At first sight it seems undesirable to permit calculations to proceed in this erroneous way; but in fact there is no compelling reason to take such an attitude. Suppose, e.g., that 32-bit hashes are used, and that each program is run three times (different hash functions being used each time) to give the effect of a 96-bit hash. Then even a program that makes 10^6 remember-block entries before completing should execute with apparent success except in one case out of 10^{-21}, a false-result rate which seems quite acceptable.

The semantic mechanisms suggested in the preceding pages serve nicely for the description of comprehensive searches of arbitrarily complex spaces, and ensure that pruning of these searches by programmed detection of search-path failure remains easy and convenient. However, in some cases, more general control mechanisms may be desirable. For example, even after well-programmed search-space pruning, spaces may remain to be searched, and investigation of these spaces may be impossible unless search is guided by some heuristic. Various heuristics have been considered in the literature:

i. *Depth-first search*, which is guided by some conjectured estimate of the distance between nodes y of a graph G being searched and the target node x. Such a search will prefer to examine the neighbors of that particular previously reached node which is felt to be closest to x before examining any other points.

Breadth-first search is a variant of depth-first search in which one assumes that the starting point x_o of the search lies near the search's target point x, and therefore estimates the distance of y from x simply to be y's (known) distance from x_o. This sort of search will examine all y near x_o before any nodes y' farther from x_o are examined.

ii. Symmetry pruning. In searching a graph G, it is important to avoid redundant re-examination of nodes that have already been examined. If both G and the predicate C(x) that one is attempting to satisfy have certain symmetries v, then one should not examine a point y if a symmetrical point σy, has already been examined, since if there exists a path from y to an x satisfying C(x) there will also exist a path from σy to x.

The ok remember mechanism described in the preceding pages allows redundant node examination to be avoided, and the more general ok remember *e* diction allows one to prune away nodes symmetric to a node already under examination. To allow expression of depth-first searches in a nondeterministic language, one can simply replace the fail statement that would otherwise be used by a statement estimate *e*, where *e* is real-valued and defines the amount of work that one expects to have to do before calculation forward from the node (i.e., environment) under examination leads to an x satisfying c(x). (Then fail comes to be equivalent to estimate ∞.)

2. General Backtrack Mechanisms, Some Illustrations of Their Use.

To allow the semantic mechanisms ok remember, estimate, and other similar constructs to be defined by programs, it is convenient (following Sussman's CONNIVER) to introduce general primitives allowing data environments to be treated as semantic objects. An adequate set of rules for this purpose is as follows (see NL 153):

(a) Environments become SETL objects which are treated rather like blank atoms. These objects can be set members and tuple components, and can be tested for equality. A copy e' of an environment e can be created by writing e' = copy(e).

(b) Only one environment is executing at any given moment. Inter-environment transition is accomplished using the coroutine-call-like primitive

(1) valuereturned = cocall (newenvironment, valuesent).

This *cocall* exits from the current environment *ce*, enters *newenvironment*, and leaves *ce* suspended 'halfway thru' the *cocall*, ready to receive a value back when and if control is eventually returned to *ce*. Note accordingly that, with the exception of the one environment that is currently executing, all other environments represent processes which are suspended awaiting a value to be returned by a *cocall*. The value transmitted by *ce* to *newenvironment* via the cocall (1) is the pair <ce, valuesent>.

(c) The special cocall

(2) cocall (Ω, valuesent)

generates a copy *env* of the currently running environment *ce*, and transfers control to *env*.

The system of primitives which have just been explained embodies the following semantic appreciations: to allow one environment A to manage other environments B, it must be possible for information to be transmitted from A to B and vice-versa. On the other hand, the internal structure of an environment can be opaque, and thus it is not reasonable to let A and B write directly into each other. Hence we insist that information passed between A and B be 'packaged' as a standard-form SETL object. The receiving environment, which knows its own structure, is responsible for installing this information into itself. To create new environments, the *copy* operation is provided, in an actual implementation, this operation should be implemented in a highly efficient way. The operation (2) gives an environment an appropriately standardised way of referring to itself.

We shall now describe another useful control structure, resembling that defined by the simple ok and fail primitives previously discussed, but supporting several additional features; this structure can easily be represented in terms of the very general *cocall* primitives which gave just been reviewed. The primitives of this control structure are as follows:

(1) A new primitive trials, intended for use in iterators having the form

$$(\forall\ t \in \underline{\text{trials}}\ (\mathit{label},\ \mathit{expn}))\ \mathit{block};$$

(2) A generalised variant of the fail statement, with the form

$$\underline{\text{fail}}\ \mathit{expn};$$

(3) A generalised ok statement, having the form

$$\underline{\text{ok}}\ (\mathit{name}_1, \ldots, \mathit{name}_k).$$

The intended semantics of these three primitives are as follows: whenever a trials iterator is (dynamically) encountered, a new sequence of backtracking explorations (or 'trials') is opened. Whenever a 'fail *expn*' statement (2) is executed during this exploration, *expn* is evaluated, the value which results is assigned to t (cf. (1)), and control is returned to the beginning of the *block* of (1). Before the start of the next iteration of (1), control is returned to the ok point which would normally have received control from the fail statement (2) (if this statement were a simple fail). But in cases in which this last rule implies that control would revert to an environment established prior to entry into the trials iterator (1), one simply falls out of the iteration.

Explicit exit from a trials loop (1), whether via a *quit* statement or by a go-to whose target label lies outside the loop, terminates the loop and erases all environments created in support of it.

The initial environment for the sequence of trials started by (1) is obtained by copying the environment in which (1) is executed, but then by transferring, in the copied environment, to *label*.

After control has been returned to the start of *block* (in (1)) by a fail and *block* has been executed, the value *v* of the *expn* of (1) will be calculated. Then, when at the start of the next iteration (1) control returns to an ok point, the calculated value *v* is made available, and if ok has the extended form (3), i.e., if it involves a name list, the multiple assignment

$$<name_1,\ldots,name_k> = v;$$

is executed immediately before anything else happens in the environment containing the ok.

The set of primitives just introduced stays close in concept to the easily comprehensible kind of backtracking defined by ok and fail in their simplest form, but allows a substantially greater measure of interaction between trials.

It is convenient to allow

(1') $(\forall t \in$ trials (label)) *block*;

as an abbreviation for

$(\forall t \in$ trials (label, Ω)) *block*;

To represent the primitives (1), (2), (3) in terms of the more general *cocall* operations introduced previously, it is convenient to begin by introducing an auxiliary control construct having the form

(4) <env,val> = try (label);

The intended semantics of this construct are as follows: it forms a new environment *e*, whose internal state is a copy of the internal state of the environment *ce* within which (1) is executed.

Control passes to *e*, which immediately transfers to *label*, from which its execution continues. In *e*, *env* has the value *ce*, and *val* the value Ω. Later, by executing *cocall(env, v)*, *e* can return to *ce*, transmitting <e,v> as the value of the function *try* in (1). The following code represents this handy control construct in terms of the *cocall* primitive.

```
<xenv,xval> = cocall (Ω,Ω); /* form e and transmit ce to it */
if xval eq Ω then           /* we are in e */
    <env, junk> = cocall (xenv,true); /* return control to ce */
    go to label;  /* when control given back, go at once to label */
else /* we are in ce, having received control back from e */
    <env, val> = cocall(xenv,Ω); /* return control to e */
end if;
```

Using this construct, we can represent the primitives (1), (2), (3) as follows. From an initially given environment *init*, we split off the first of a family of experimental environments *exp* by executing the statement

<master, val> = try (start);

Here, *master* is a global variable, which will have the value *init* in all 'experimental' environments. The primitives (1), (2), and (3), which will be executed only in 'experimental' environments, but not in *init*, are then represented as follows:

(a) The ok ($name_1$,...,$name_k$) primitive is represented as

```
[if not (hd (cocall(master,0)(2) is response) is okval) then
        <name_1,...,name_k> = response (2); end if;
            return okval;]
```

Note that *response* is here expected to have either the form <t, Ω> or the form <f, v>.

(b) The fail *expn* primitive is represented as

```
        junk = cocall(master, <false, expn>);
```

(c) The (∀t ∈ trials (label, expn)) *block*; primitive is represented as

```
        if cocall (master,1)(2) is save eq Ω then go to label;;
        /* when the 'master' environment init is called with    */
        /* parameter 1 by an experimental environment exp , it */
        /* will form two copies of exp, and pass Ω back to the */
        /* first of these; when control is returned to the other */
        /* copy, either a value of the form <false, t> (for final */
        /* iterations) or <okenv, t> (for iterations which are to */
        /* pass control to an environment that is in the midst  */
        /* of executing the ok primitive) will be transmitted. */
        go to enter;  /* jump to first trials loop iteration */
retry:  <junk, save> = cocall (okenv, expn); /* give control to okenv */
enter:  <okenv, t> = save;  /* decode parameter */
        block;              /* execute block */
        if okenv ne false then go to retry;; /* to attempt next trials */
```

(d) The code prologue executed within the environment *init* is as follows:

```
         <master, val> = try (start);
         exp = master;  /* within init  the value of the variable */
                        /* master is the initial experimental environment; */
                        /* within experimental environments it is init */
         envstack = nult; /* initialise stack of environments */
control: if val eq 0 then  /* we deal with an ok call */
              envstack (# envstack + 1) = <f, copy (exp)>;
              <exp,val> = cocall (exp, <true, Ω>); /* return true to exp */
         else if val eq 1 then
```

```
/* we deal with the cocall which initiates a new family of trials */
     envstack (# envstack + 1) = <t, copy (exp)>;
     /* note that the t flag distinguishes an environment in */
     /* which a trials loop is being executed. */
     <exp,val> = cocall (exp, Ω); /* return Ω to exp, thus */
                                  /* starting trials */
else if val eq 2 then
/* we deal with an explicit exit from the last trial loop entered */
     if # envstack ≥ ∃n ≥ 1 | hd envstack(n) then
     /* drop all environments since last trials loop entry */
          envstack = envstack (1: n-1);
     end if # envstack;
     <exp,val> = cocall (exp, Ω); /* now re-enter environment exp */
else  /* a fail operation has just been executed */
     if envstack eq nult then
          print 'attempted run results in total failure';
          stop;
else /* pop environment off envstack */
     <flag, env> = envstack (# envstack);
     envstack (# envstack) = Ω;
     if flag then /* trials environment. pass parameter */
                  /* indicating imminent loop exit */
     <exp, val> = cocall (env,val); /* note that hd val is false */
     else if # envstack ≥ ∃n ≥ 1 | hd envstack(n) then
          trier = envstack(n) (2); /* last preceding trials */
                                   /* environment */
          <exp,val> = cocall (trier, <env,val(2)>);
          /* this passes data from fail operation to trials */
          /* environment and also passes the environment which is */
          /* subsequently to receive control */
     else /* there is no preceding trials environment; control */
          /* returns to last preceding ordinary environment */
          <exp, val> = cocall (env,val);
     end if flag;
```

```
        end if val;
        go to control;
start:      /* the environment init begins to execute here */
```

Note that it is assumed in the preceding code that explicit exits from a trials loop, whether by *quit* or by *go-to* statements, are compiled as

```
                    junk = cocall (master,2); quit;
```

and as

```
                    junk = cocall (master,2); go to label;
```

respectively. If *label* is a variable, then the somewhat more complicated code pattern

```
    if label not ∈ labelset then junk = cocall (master,2);;
                    go to label;
```

is required. Here *labelset* is the collection of all labels which are internal to the trials loop in which the go-to appears. To handle nested trials loops, a yet more complicated treatment, which we leave it to the reader to work out, is required.

As an example illustrating the use of the semantic and syntactic mechanisms sketched in the preceeding pages, we shall now give a number of parsing algorithms. We begin with a variant of the 'nodal spans' algorithm, for which we assume that we are given a grammar of Chomsky normal form productions $\alpha \to \beta\ \delta$, represented as a set *gram* of triples $<\alpha;\beta;\delta>$, plus a set *termpro* of terminal productions $\alpha \to A$, represented as pairs $<\alpha,A>$.

```
definef splitspan (i,α,j) remember;
return if j eq i + 1 then
     if input(i) ∈ termpro{α}then i else 0
     /* here 0 is simply used to mark spans 'not present in the input'*/
          else if i < ∃n < j, <β,δ> ∈ gram {α} |
               splitspan (i,β,n) ne 0 and splitspan (n,δ,j) ne 0
                    then <β,n,δ> else 0;
end splitspan;
```

It is interesting that this algorithm very much resembles a top-down parsing algorithm in form.

The usual nodal span algorithm determines ambiguity, which the preceding algorithm does not; of course, an easy modification of the preceding will determine ambiguity also. Note that the preceding algorithm, like Earley's improved variant of the nodal span parse, never comes to consider any span (i,α,j) which is not relevant to the parse tree of some continuation of the left-hand context string input(1: i-1) of (i,α,j).

Next we shall use our general semantic mechanisms to represent an interesting, highly generalised variant of the bottom-up parse. In writing this algorithm, it is important to keep the following issues in sight.

(a) We want our algorithm to be able to deal with illformed input.

Thus it will have to distinguish between 'temporary' failures occurring during exploration of a parse tree,and 'real' failures caused by illformed input. In case of a 'real' failure, it should be able to pinpoint an error location, and specify a diagnostic message.

(b) We do not want our algorithm to be grossly inefficient. In particular, we want it to be linear in cases where a linear parse exists.

To ensure that condition (b) is met, we shall make use of the fact (discussed at greater length in Phil Owen's thesis, Courant Computer Science Report # 4) that the collection C of all potentially handle-free sentential strings of a context-free language L is a regular language, and may therefore be recognised by a finite state automaton AC. By using this automaton in our algorithm, we can ensure that we will not carry our exploration past any point at which the presence of a syntactic error can definitely be asserted. This implies that our algorithm, although general and nondeterministic, will in most cases be linear if the grammar with which it is working is LR(k) for some k.

We handle errors as follows. Possible parses are explored by generating new environments. During this exploration, we keep a record of the maximum number of input symbols accepted in each environment. If a definite failure is detected, i.e., if in every environment a failure occurs before the input is fully parsed, we use the environment E in which the maximum number of input symbols has been accepted to define an error location. A diagnostic message is then emitted, the particular message chosen being a function of the first symbol in the environment E, and also of the state σ of the automaton AC in this environment at the moment of rejection.

When a diagnostic is emitted, the part of the input which has been scanned in E is deleted, and analysis of the remainder RI of the input continues; of course, RI must be analysed using the grammar G'which describes tails of sentences rather than complete sentences. If the productions of the original grammar are $\alpha \rightarrow \beta_1 \ldots \beta_n$, then G' includes not only these productions by also all productions of the form $\alpha \rightarrow \eta\ \beta_j \ldots \beta_n$, where $2 \leq j \leq n$ and η is a dummy 'start of sentence' symbol. In the algorithm shown below, which has the standard 'shift/reduce' structure, we treat the *stack* vector (on which shifted symbols are placed) as if it had two copies of η appended to its left, provided that an error has already occurred. Likewise, since an error may occur, the automaton AC should be replaced by an automaton AC' whose states σ are subsets of the set of all states of AC. The transition and rejection rules for AC' are derived in an obvious way from those for AC; AC can be regarded as having a set of states identical with the set of all singleton states $\sigma = \{\alpha\}$ belonging to AC'. The initial state $\sigma 1$ of AC' is simply the set of all states of AC. If an error has occurred, we start AC' in the state $\sigma 1$; if no error has occurred, we start it in the state $\{\alpha_o\}$, where α_o is the initial state of AC. In this latter case, AC' simply mimics the action of AC. If no error has occurred, the stack vector is initialised with two copies of a sentence start symbol $\vdash$ instead of two η's. The extended grammar G' is always used for parsing, but none of the productions belonging to G'-G can be relevant unless an error has occurred and the stack initialised with η's. We assume that every input string is terminated with an end-of-input mark $\dashv$.

Detailed conventions are as follows. The *grammar* is given as a set of pairs $<\alpha,<\beta_1,\ldots,\beta_n>>$ which correspond to productions $\alpha \rightarrow \beta_1,\ldots,\beta_n$; 'tail of sentence' productions $\alpha \rightarrow \eta\beta_j,\ldots,\beta_n$ are also represented in *grammar*. The special symbols $\vdash$ and η are called *cleanstart* and *errorstart* respectively. The automaton AC' described in the preceding paragraph is represented by a mapping *transition(state,symb)*; the initial state of AC' is called *cleanstartstate*, and the state into which AC' passes each time an error is detected and diagnosed is *errorstartstate*. We assume that we have available a table *errormessage(state,nextsymbol)*, which selects messages depending on the state of AC' at the moment at which an error is detected and on the symbol following the point of error.

Since it suggests a technique allowing the efficiency of the following algorithm to be improved, we have written the ok operations appearing in the algorithm as ok remember c, where c is a pair consisting of the current state of the automaton AC', together with a few symbols (more precisely, *contextlength* symbols) of right-hand context. This is essentially the information which an LR-parser would use in making condense/no-condense decisions. The remember feature which is thereby assumed could be implemented using a *pastexperience* mapping in the manner sketched on page 7 above. (Note however that the code given above to implement the trio of control primitives ok, fail, trials does in fact not support any ok remember feature). It is interesting to observe that, if the ok remember construct were supported in this manner, the efficiency of the parser which now follows would improve steadily as it processed a stream of text, and in most cases would become asymptotically proportional to the efficiency of a deterministic LR parser.

```
       /* grammar, root and contextlength are global quantities of this */
       /* algorithm.Initialise startsymbol and startstate to indicate */
       /* that no error has yet occurred.                          */
       <startsymbol, startstate> = <cleanstart, cleanstartstate>;
       ipointer = 1;  /* initialise inout pointer */
       statefar = startstate;  /* note that auxiliary automaton begins */
       farthest = 0;           /* in initial state                    */
       (while ipointer lt # input)
            (∀t ∈ trials (start)|t(1) gt farthest)
                 <farthest, statefar> = t;
            end ∀t;
            print errormessage (statefar, input (farthest));
            ipointer = farthest + 1;  /* bypass bad input */
       /* reset startsymbol and startstate to show error */
            <startsymbol, startstate> = <errorstart, errorstartstate>;
            statefar = startstate;
       end while;
       print 'all syntactic errors have now been diagnosed';
       stop;
start:  /* entry point to begin each sequence of trials */
        stack = <startsymbol, startsymbol>;
       statestack = <startstate, startstate>;
       (while ipointer le # input and
                           stack ne <startsymbol, startsymbol, root>)
                 if  ∃pair ∈ grammar |(# stack) ge (#(pair(2) is rtside) is n)
                     andd stack ((# stack -n) is ilength + 1:) eq rtside
                     andd transition (statestack(ilength) is oldstate,
                                       pair(1))  is newstate ne Ω
                     andd ok remember <oldstate,input(ipointer:contextlength)>
                                       then  /* perform condensation action */
                           stack = stack (1: ilength) +
                                       if ilength gt 1 then <pair(1)> else
                                          <startsymbol, pair (1)>;
                           statestack = statestack (1: ilength) +
                                       if ilength gt 1 then <newstate>
                                           else <startstate, newstate>;
        else /* check legality of shift action */
            if transition (statestack(#statestack) is oldstate,
            input (ipointer)) is newstate ne Ω
            andd ok remember <oldstate,input(ipointer:contextlength)>
                then /* perform shift action */
                stack = stack + <input(ipointer)>;
                statestack = statestack + <newstate>;
        else /* neither condense or shift action is possible */
            /* therefore we fail, returning input position and state*/
            fail <ipointer, oldstate>;
        end if;
end while;
/* if we fall out of this loop, then tail of input is acceptable */
print  if startstate eq cleanstartstate then
        'string is syntactically acceptable' else
        'remainder of string is syntactically acceptable';
stop;
```

An optimization, based on global analysis, which applies in the semantic context defined by the three primitives ok, fail, and trials can be described as follows. Call an occurrence p of ok *bounding* if no occurrence of fail can be reached from p when ok is given the value false, or more generally and recursively if no such occurrence of fail can be reached without first passing through some other bounding occurrence of ok. Now note that if an occurrence of ok is bounding then whereas failure can propagate back to it, thereby requiring retrieval of the environment logically saved when the ok is first evaluated and given the value true, it can never fail completely, i.e., failure can never be propagated back to an environment saved before evaluation of a bounding occurrence of ok. Therefore on evaluating a bounding ok, one can destroy all the *envstack* entries ancestral to it, back to the last preceding occurrence of a trials entry on *envstack*.

This will generally save space, and, in implementations which represent environments differentially, may also speed up execution.

Certain global SETL-level optimizations carry over from the deterministic to the nondeterministic case without difficulty. Consider, for example, relationships of inclusion and membership iRo' and oRo' of the sort studied in Newsletter 130. In the presence of primitives permitting multiple environments, but where we assume that all these environments share the same ivariables and ovariables (this will be the case for all the systems of control primitives that we have discussed) we consider iRo' (resp. oRo') to hold if it holds in every environment. Since the variables internal to an environment e are modified only by statements executed within e, relationships of inclusion and membership can be deduced by fixing attention on some single environment and treating its cocalls to other environments as if they were *read* statements.

Similarly, if we assume that values transmitted by cocalls are always copied (this assumption is reasonable since to define most of the control structures considered above only boolean and integer values need to be transmitted by cocalls), then global value-flow analysis can be carried out simply by fixing attention on a single environment e. This same remark then carries over to copy optimization, operator-operand analysis, etc. However, to decide on the implications of all of this for data-structure choice, a closer study of data-structure choice issues in backtracking environments is required.

3. An additional Comment on the Optimization of Searches.

The mechanisms suggested in the preceding pages make it is easy to describe both comprehensive and depth - first searches of arbitrarily complex spaces, and to ensure that pruning of these searches by programmed detection of search-path failure remains easy and convenient. Moreover, these same mechanisms allow symmetry pruning, which is another important and general technique of search optimization, to be expressed easily. We shall now describe a third technique for guiding searches over large sets, which can be quite powerful in certain cases in which simple depth first and symmetry pruning techniques are insufficient. This technique can most readily be comprehended if one considers the problem of searching a product graph $G = G_1 \times G_2$ to find a target element $x = \langle x_1, x_2 \rangle$ satisfying a predicate $C(x)$ of the form $C_1(x_1)$ <u>and</u> $C_2(x_2)$. If G is searched without taking account of its product space structure, then $n_1 \times n_2$ elements may have to be examined, where G_1 contains n_1 and G_2 contains n_2 elements. On the other hand, suppose that one first searches for an element $\bar{x} = \langle \bar{x}_1, \bar{x}_2 \rangle$ satisfying the simple predicate $C_1(\bar{x}_1)$, treating elements as equivalent during this search if they have the same second component; and then starting from $\bar{x}$ searches for the desired x, now taking second components into account but confining the search to elements $y = \langle y_1, y_2 \rangle$ satisfying $C_1(y_1)$. Then it may not be necessary to examine more than $n_1 + n_2$ elements.

It is clear that the same remark applies in searching product graphs $G = G_1 \times \ldots \times G_n$ involving several factors. What is more interesting is that this remark can be generalised to apply when one searches graphs that are almost but not quite products. We shall assume that the 'almost product' structure of such a graph is defined by a sequence $\phi_1, \phi_2, \ldots, \phi_n$ of mappings (which in the product graph case would be the projections of G on $G_1 \times G_2 \times \ldots \times G_n$).

Suppose first that the predicate C defining the target point x of our search has the form $C(x) = C_1(x)$ <u>and</u> ... <u>and</u> $C_k(x)$. Then we might use $d(y) = \# \{j | \underline{\text{not}}\ C_j(y)\}$ as an approximate measure of the distance between a given point y and the search target x. Let f(D) be a monotone decreasing function of the distance variable D, and suppose that $f(1) = n$ while f(k) is substantially smaller, e.g., $f(k) = 1$. Suppose that a <u>remember</u> clause is added to each occurrence of <u>ok</u> in a program searching G, and specifically that we always write

<u>ok</u> <u>remember</u> $\Phi(1{:}\ f(D))$

at occurences of <u>ok</u>; where $\Phi = <\phi_1,\ldots,\phi_n>$, and where as above D counts the number of target predicates C_j which fail to hold in a particular environment (i.e., graph node) under exploration. This will have the effect of limiting the number of environments which are explored; e.g., we will never examine two environments e_1, e_2 unless either $\phi_1(e_1)$ and $\phi_1(e_2)$ differ or one of e_1, e_2 satisfies at least one of the target predicates C_j. In general, the more target predicates are satisfied in an environment e, the more detailed a view we take of it, i.e., the more of the functions ϕ_j we are willing to regard as'significant' attributes. (The feature $\phi_j(e)$ is 'significant' if we are willing to examine both e and an environment e' whenever $\phi_j(e')$ are different.) Note that by using the scheme just outlined and the auxiliary function $f(D) = n - D + 1$, we solve the prototype problem of connecting two points

$x_1^o \times\ldots\times x_n^o$ and $x_1 \times\ldots\times x_n$ in a product graph $G_1 \times\ldots\times G_n$ without difficulty, whereas more conventional search techniques, whether depth first or breadth-first, can be expected to fail when applied to this problem. Observe also that this scheme can remain useful even in connection with searches whose target predicate C is not a conjuction, provided that we use a distance heuristic function D having the property that $\# \{ x \in G \mid D(x) < \delta\}$ diminishes suitably with diminishing δ.

References

1. James R. Bitner and Edward M. Reingold, "Backtrack Programming Techniques", Comm. ACM (nov. 1975), 651-655.

2. Donald E. Knuth, "Estimating the Efficiency of Backtrack Programs", Tech. Rep. Stan-CS-74-442, Computer Science Department, Stanford University, August 1974.