

# Syntax-Directed Least-Errors Analysis for Context-Free Languages: A Practical Approach

Gordon Lyon  
Institute for Computer Sciences and  
Technology, National Bureau of Standards  
U.S. Department of Commerce

A least-errors recognizer is developed informally using the well-known recognizer of Earley, along with elements of Bellman's dynamic programming. The analyzer takes a general class of context-free grammars as drivers, and any finite string as input. Recognition consists of a least-errors count for a corrected version of the input relative to the driver grammar. The algorithm design emphasizes practical aspects which help in programming it.

**Key Words and Phrases:** arbitrary input strings, context-free grammars, parsing, dynamic programming, stored subanalyses, separability, state merging, least-errors correction

**CR Categories:** 4.12, 5.23, 5.42

## Introduction

There are many advantages to syntax-directed context-free analysis (Feldman, 1967 [10]), but in general, error recovery and correction are not among them. Feldman and Gries, 1968 [11], writing some years ago in an article on translator writing systems, specifically mentioned two general areas where automatic recovery or correction of syntax errors would be very

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This work was supported in part by the Mental Health Research Institute, University of Michigan, Ann Arbor. Author's address: U.S. Department of Commerce, National Bureau of Standards, A265 Tech, Washington, DC 20234.

useful—in translator writing systems and in relation to on-line control languages. Other possibilities might lie with computer-aided instruction systems and speech recognition.

Path-at-a-time pushdown methods can theoretically minimize errors in sentences, but at an enormous computation overhead. It is easy to trick such analyses into  $k^n$  execution times for input of length  $n$ , and constant  $k$  determined by a specific grammar. The algorithm which is developed here—based upon Earley's recognizer for context-free languages—maintains  $n^2$  storage bounds and  $n^3$  execution bounds, which Earley, 1970 [5], established.

An earlier algorithm by the author (Lyon, 1972 [22]) used Younger's (1967 [31]) recognizer as a framework for time  $n^3$  least-errors recognition. Principal drawbacks in that method were: (i) the limited Chomsky normal-form grammars; (ii) restricted errors (only mutations—replacement of a terminal by another terminal—were handled); (iii) inflexible use of storage (Younger's algorithm is matrix oriented). The new algorithm relaxes each above restriction.

## 1. Selected Prior Efforts

A brief mention of some previous approaches to error recovery and correction is helpful in placing results of this paper into perspective.

### Failure-driven Techniques

Conway, 1963 [4], presents early arguments on a (deterministic) parser which can indicate faulty input easily. Bollinger, 1968 [2], discusses a metacompiler for Conway's recursive transition diagrams; the mechanism includes a facility for specifying limited error-recovery actions.

Levy, 1971 [21], considers clusters of  $n$  or less errors at points in sentences of deterministic languages. Using an error-correction mode only when errors are detected, his scan proceeds left-to-right, treating each cluster independently. He shows that fiducial points may exist in the scan; these can help in determining the range of a cluster of errors.

Leinius, 1970 [20], presents a simple-precedence parser which automatically generates a stack discipline for recovering from error conflicts. The discipline does not interfere with normal processing. LaFrance, 1970 [18] and 1971 [19], covers other stacking strategies, paying attention to avoid severe error-induced backtracking. Adaptive strategies have been proposed by James and Partridge, 1973 [15].

Somewhat distinct from other failure-driven error-handling techniques, Eggers, 1972 [7], begins by examining regular expressions. Left-right and right-left scans are employed. Errors show as inconsistencies in states of the left-right and right-left analyses for a given input point. This dual-scan correction mecha-

nism is then incorporated (Eggers, 1972 [8]) into a system of recursive finite state automata.

Failure-driven analysis which requires serial exploration of multiple paths can get tangled easily in combinatorics of backtracking. A left-to-right corrector of Irons, 1963 [14], pursues concurrent analyses to diminish combinatorial effects. Although more general than most failure-driven strategies, Iron's corrector still assumes that errors lie within "local tolerance" of the last path to fail. In conjunction with Iron's ideas on concurrent analyses for error handling, Wegbreit, 1970 [29], recommends Earley's recognizer to handle an extensible language.

Local error correction may not lead to a least-errors correction. For example, consider a language  $\mathcal{L} = \{ab^k\text{efg} \vee db^k\text{xyz} / k \geq 5\}$ . Input "db<sup>k</sup>efg" has a good correction "ab<sup>k</sup>efg," but local corrections lead to "db<sup>k</sup>xyz." On the other hand, global correction can be extremely time-consuming if it is not approached carefully.

## Hypothesis Generators

Methods in this paper fall among techniques which use hypothesis generators internal to analysis mechanisms. Developing such methods, Hopcroft and Ullman, 1966 [12], discuss the increase in difficulty of recognition for formal languages, as input strings differ in various ways from well-formed sentences:

The main results are that  $\epsilon$ -tuple errors, burst errors, and the type<sup>s</sup> of errors corrected by recurrent codes preserve regular sets, context-free languages, and context-sensitive languages. However, deterministic context-free languages are not in general preserved under any of the above error types . . .

Levy, 1971 [21], includes a brief discussion on terminal symbol insertion, deletion, and mutation hypotheses applied to context-free analysis. Since  $k$  or less errors in sentences preserve context-free properties, Levy argues that Earley's recognizer can correct  $k$  or less errors in time  $n^3$ . The constructive approach in this paper shows that restriction to  $k$  or fewer errors is unnecessary—input may be any string over the set of terminal symbols  $T$ . A modified Earley's algorithm naturally limits execution times to  $n^3$ .

Peterson, 1972 [25, 26], develops a theoretical least-errors parser using Earley's algorithm and *extended* grammars. New rules supplement an original grammar to account for insertion, deletion, and mutation errors. For example, an extended Euler grammar will have quadrupled its original rules. Although Peterson's grammar extensions neatly support least-errors correction, they introduce problems as one swings toward a practical side: space is consumed; execution may be slowed.

Mention has been made that Younger's recognizer can be modified to correct mutation errors. Teitelbaum, 1973 [28], provides further results which relate such analyses to algebraic power series.

Work in syntax-directed pattern recognition has a

strong similarity to error-correction problems in context-free language parsing. A picture grammar uses pattern primitives and elaborates syntactic structures upon these basic forms. Imagine an analysis of hand-printed letters. Variations in printing take the form of disturbances on the primitives and amount to the addition of noise. Thus:

A question was asked [by Gorn] regarding how a graphical manipulation system interprets syntactical specifications for [pictorial] objects when the syntactical specifications refer to objects that are not well formed, hence, that have no pictorial representation. . . . [Discussion summary on graphical languages. *Comm. ACM* 9, 3 (Mar. 1966), 175].

Kovalevsky, 1968 [17], attacks this problem in noisy pictures. His goodness-of-fit function is separable into stages compatible with segments of grammatical analysis. As a consequence, dynamic programming can be used to obtain best-fits in a fairly efficient way.

## 2. Preliminaries

### Context-free Language Terminology

A *context-free language*, CFL, is generated by a four-tuple  $G = (N, T, S, P)$ ;  $N$  is an alphabet of variables;  $T$  a set of terminals; and  $P$  a set of context-free rules of the usual form.  $S$  is a distinguished start symbol, sometimes replaced by  $Z$  in the following text.

Let  $p$  be an index over rules  $P$  of grammar  $G$ . Rule  $p$  is written as  $L(p) = c(p,1)c(p,2) \cdots c(p,p)$ , where  $L(p)$  is the left-hand side, LHS, of rule  $p$ , and  $c(p, \cdot)$  are *components* in the right-hand side, RHS. Rule  $p$  defines a phrase of variable  $L(p)$ .  $RHS(p)$  references phrase  $p$  of variable  $V = L(p)$ .

A *substitution* replaces an occurrence of a rule's LHS with the RHS of the rule. A *derivation* is a sequence of substitutions: If  $x \rightarrow y \rightarrow z$ , write  $x \xRightarrow{*} z$ . A CFG is *grounded* if, for any variable  $V$  in  $N$ , there exists a string  $w$  in  $T^*$  and  $V \xRightarrow{*} w$ . Only grounded grammars are considered in subsequent discussion. String (word)  $w$  over  $T$  is a *sentence* in language  $\mathcal{L}(G)$  if there is a derivation from  $S$  to  $w$  via rules of grammar  $G$ .

### An "Order" Notation

Given  $f(x)$  over domain  $X$ , function  $f$  is of order  $g(x)$ , written  $f = O(g(x))$ , if for fixed constants  $k$  and  $c$ ,  $f(x) < k g(x) + c$ . Let  $fh$  denote  $f(x)h(x)$ , a product of two functions. If  $f(x) = O(x)$  and  $h = O(x)$ , then

$$fh < k_f k_h x^2 + c_f k_h x + c_h k_f x + c_f c_h$$

so that  $fh = O(x^2)$ . In general  $O(x^k)O(x) = O(x^{k+1})$ . For function  $f(x, y, \dots; a, b, \dots)$ ,  $f(x, \dots) = O(g(x))$  provided constants  $k$  and  $c$  can be chosen to maintain bound  $kg(x) + c$  for all parameter and argument values.

Fig. 1.

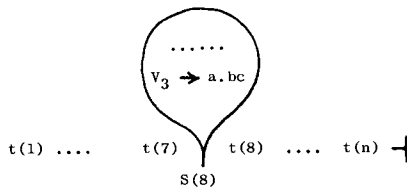
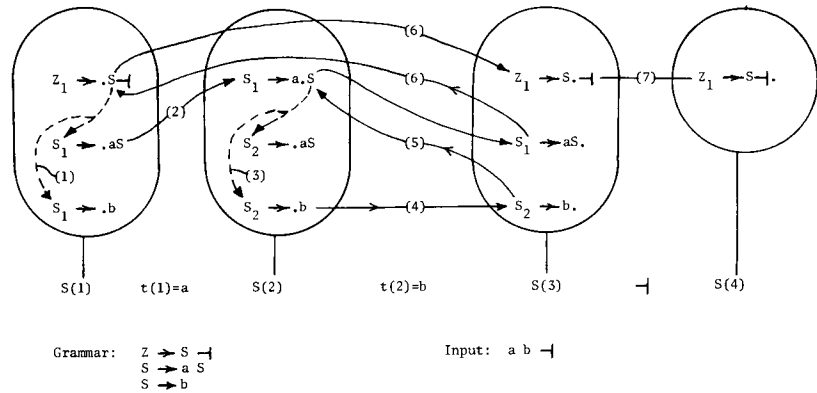


Fig. 2.



## Text symbols

$\dashv$	HALT (end) marker
P	set of productions for G
$\xi$	error map (strings to strings)
$ P $	number of productions
$L(p)$	left-hand side of rule $p$
$c(p, j)$	component $j$ of rule $p$ 's RHS
$\underline{p}$	number of components in rule $p$
$t = t(1) \cdots t(n)$	input string, length $n$
$i, j$	indices
$e, e', e''$	error counts
$f$	backpointer to statesets
N	set of variables for G
T	terminals
$S(i)$	stateset $i$ , composed of states
$(p, j, f, e)$	state, a four-tuple
*	Kleene star
$O( )$	"order"
G	a context-free grammar
$\mathcal{L}(G)$	language of G
$n$	length of input string $t$
$\alpha, \beta, \gamma$	strings
$\epsilon$	null (empty) character
Z	a new distinguished start symbol
$\Rightarrow$	derives via grammar G
$\Rightarrow_{\xi}$	derives via G and error map

## Earley's Algorithm Without Lookahead

Preparatory to discussion of the error corrector, details of Earley's context-free language recognizer are informally reviewed.

Any context-free grammar G with distinguished rule  $Z \rightarrow S \dashv$  is suitable, given that halt symbol  $\dashv$  and distinguished variable Z appear only in this first of productions,  $p = 1$ . Remaining rules are arbitrarily ordered.

Input  $t = t(1) \cdots t(n) \dashv$  is a member of  $T^* \dashv$ . Each position preceding and following input character

$t(i)$  is indexed as  $i$  and  $i+1$ , respectively. The index position falls between input characters. Location 1 marks a locus in front of the whole input string  $t$ . Index  $n+1$  falls between  $t(n)$  and  $\dashv$ .

A *state* is a triple  $(p, j, f)$  where  $p$  is a production number.  $j$  marks a position in  $RHS(p)$ , so that  $1 \leq j \leq \underline{p}+1$ .  $f$  is a pointer to some location in  $t$ : namely, a point on the left of  $t(f)$  as described above. State  $(p, j, f)$  is *final* provided that  $j = \underline{p}+1$ ; i.e. cursor  $j$  has reached maximum value. Final states represent phrases which have had all components recognized and have been recognized themselves. Final states signal certain substitutions which are to be discussed later.

A *stateset* is a set of states—where no particular constraint is imposed—other than order of arrival of states. (The changes to accommodate errors will include a stronger constraint on order.) A state may be added to the end of the list of states of some stateset unless it is already on the list. Each stateset  $S(i)$  is uniquely associated with position  $i$  of input  $t$ . There are  $n+2$  statesets for input  $t(1) \cdots t(n) \dashv$  of length  $n+1$ .

States in a stateset  $S(q)$  represent partial phrases that are known up to point  $q$  in a left-to-right scan of input  $t$ . If  $(p, j, f)$  is in  $S(q)$ , then components  $c(p, 1) \cdots c(p, j-1)$  of rule  $p$  have been discovered in substring  $t(f) \cdots t(q-1)$ . Backpointer  $f$  indicates where component  $c(p, 1)$  begins; i.e. stateset  $S(f)$  had state  $(p, 1, f)$  which initiated the search for  $L(p)$  represented by  $(p, j, f)$ .

A pictorial notation of state works well for illustrations. Backpointer  $f$  becomes a subscript to an LHS; cursor  $j$ , a dot in an RHS. Thus a rule  $V \rightarrow abc$  numbered  $p=13$ , and state  $(13, 2, 3)$  in stateset  $S(8)$  can be explicitly described as in Figure 1.

Figure 2 depicts the mechanisms of Earley's recognizer in labeled steps. To start, state  $Z_1 \rightarrow .S \dashv$  is placed at the head (of the empty) order list for stateset  $S(1)$ . Processing may now begin.

In Step 1 of Figure 2, a state is removed from the list of  $S(1)$  states. Only  $Z_1 \rightarrow .S \dashv$  is available. The cursor in this state indicates that the variable  $S$  is sought. The algorithm adds to  $S(1)$  all states which represent possible searches for  $S$ . Such states have back pointers  $f = 1$  and represent rules  $r$  such that  $L(r) = S$ . The dot cursor shows that no part of any RHS of any new state in  $S(1)$  has been matched. That is,  $j = 1$ .

In Step 2 of Figure 2, another selection from  $S(1)$ , viz.  $S_1 \rightarrow .aS$ , successfully matches its ".a" to  $t(1)$ .  $S_1 \rightarrow a.S$  is added to  $S(2)$ . A third selection from  $S(1)$  only exhausts the list of states for  $S(1)$ .

The first state chosen from  $S(2)$  predicts, in Step 3, a phrase  $S$ , and generates two new states on the list for  $S(2)$ . The first of these is of no value, but Step 4 puts a state into  $S(3)$ .

Only one state is available in  $S(3)$ .  $S_2 \rightarrow b.$  is a final state with a pointer back to  $S(2)$ . The algorithm returns to  $S(2)$  and advances into  $S(3)$  all states awaiting phrase  $S$ , appropriately advancing their cursors. Step 5 depicts this, which causes another final state to enter  $S(3)$ . Step 7 completes recognition.

The recognizer is easily changed to a parser. With provision for pointers to substituted phrases, a complete phrase structure is available.

The least-errors recognizer in the next section builds upon Earley's concepts by incorporating dynamic programming into recognition. This requires an augmented notion of state.

Because the algorithm attempts to use storage efficiently, *stateset* is redefined, as are orders and methods of processing states in a *stateset*.

### 3. A Least-Errors Recognizer

Sentences may have mutations, insertions, and deletions of terminals. Any grounded context-free grammar  $G$  is acceptable to the algorithm, provided there is a distinguished rule  $Z \rightarrow S \dashv$ , as discussed previously.

Recognition consists of an optimal, least-errors count  $e$  of errors in  $t$ , the input.  $t \in T^*$ . The techniques are easily adapted to supply more details of analysis. A version of the algorithm has been programmed which gives error counts, error loci, and a count of the number of least-errors solutions.

No detailed proof is given for the new algorithm. Readers interested in developing a proof might find Jones, 1972 [16] helpful.

The objective is to find the least-errors necessary for correction. Such error counting is separable, that is, subanalyses contribute to analysis without interactions (crossproducts). For example, consider a rule  $X \rightarrow YZ$ . In analysis, errors of  $X$  are the sum of those of  $Y$  and  $Z$ . If  $Y$  and  $Z$  are least-errors, then the composition resulting in  $X$  is least-errors.

Since the error analysis function is separable, minimizing errors in each subanalysis is an efficient ap-

proach. A least-errors parse must have least-errors components. Otherwise, it is not least-errors, since better components could be substituted. This is actually a statement of Bellman's (1957 [1]) dynamic programming technique, which is based on his principle of optimality:

If an objective function is separable into independent segments, then each segment may be individually optimized and the results combined into a final optimal solution.

Dynamic programming is simple but subtle. For instance, it tells nothing about how to compose a final solution from optimal segments. Consequently all possible compositions are tested; a final solution is chosen from alternatives. Each alternative is optimal for its particular composition, but some alternatives are usually better than others. Only a representative least-errors correction is returned. Such a representative may or may not correspond to an original error-free sentence, depending upon which sentences in language  $\mathcal{L}(G)$  can be changed into the actual input in a minimum number of errors.

If dynamic programming is to be effective in reducing combinatorics, a recognizer should save a particular optimal subanalysis of an input segment only once. The recognizer will develop many subanalyses independently and concurrently. Subanalyses which result in the same variable being assigned to the same segment of input can be merged; a least-errors subanalysis is chosen as representative and used later with all other larger analyses which require it.

Error-correction mechanisms (given shortly) are not embedded directly and simply into Earley's particular implementation. A phrase (or subanalysis) in the error-free case is without uncertainty of any kind. With errors, a realization of a phrase may not be optimal. Any phrase used as a component in another analysis should be least-errors, thereby preventing duplication of effort which would otherwise be necessary when an improved phrase is discovered. The new algorithm seeks all recognitions of a phrase over some segment and chooses a least-errors realization as representative. There must be a late binding time for use of phrases as components, requiring not merely that phrases be found, but that they be least-errors. To this end much of the later discussion will cover definite processing constraints which do not plague the usual Earley algorithm.

Along the same vein, Earley discusses various lookahead cases. For purposes of this paper, lookahead length  $k$  is set at  $k=0$  since the error mapping is so unrestricted there is actually no information in any lookahead. Given constraints upon errors, it is possible to make some sense of  $k \neq 0$ . Indeed under special conditions Earley's  $O(n)$  execution times for deterministic grammars are possible. This holds true only if errors preserve features in the grammar necessary for deterministic recognition and if certain lists which Earley has in his implementation are included. Combi-

nations of grammar and errors which preserve deterministic parsing appear to be rather unrealistic.

### General Algorithm for Least-Errors Recognition

Input  $t$  is taken from  $T^*$ . Endmarker  $\dashv$  is never subject to error: it always signals correctly an end of analysis. The notation for productions is the same as that used for Earley's algorithm. States have an error counter, and statesets a new ordering among their states.

A *state* is a quadruple  $(p, j, f, e)$ , where  $p$  is a production number,  $j$  marks a position in  $\text{RHS}(p)$ , so that  $1 \leq j \leq p+1$ ,  $f$  is a pointer to some location in  $t$ , and  $e$  the error count. Because grammar  $G$  is grounded, there is some shortest sentence of length  $k$  in  $\mathcal{L}(G)$ . It follows that no input of length  $n$  has more than  $n+k$  errors. One may simply assume that *all* of  $n$  input characters are insertions and all of  $k$  characters of a shortest sentence were deleted. Thus  $e \leq n+k$ , for input of length  $n$ . *Final state*  $(p, p+1, f, e)$  denotes recognition of phrase  $\text{RHS}(p)$  with  $e$  errors. A state is depicted as in Section 2, except that an error count follows:  $Z_j \rightarrow S, \dashv, e$ .

A *stateset* is an ordered set of states. States within a stateset are ordered by ascending values of  $j$  within  $p$  within  $f$ ;  $f$  takes descending values. Each stateset  $S(i)$  is uniquely associated with position  $i$  of the input string  $t$ . There are  $n+2$  statesets for input  $t(1) \dots t(n)\dashv$  of length  $n+1$ .

**Adding to Statesets.** In the left-to-right matching of input, states are added to statesets. If state  $(p, j, f, e)$  is a candidate for admission to a stateset which already has a similar member  $(p, j, f, e')$  and  $e' \leq e$ , then  $(p, j, f, e)$  is rejected because of the principle of optimality. However, if  $e < e'$ , then  $(p, j, f, e')$  is replaced by  $(p, j, f, e)$ .

### Processing

The algorithm processes  $n+2$  statesets on ascending index  $i$ , doing  $S(1)$  through  $S(n+2)$ . Each stateset  $S(i)$  is initialized with states  $(k, 1, i, 0)$  for  $k$  between 1 and  $|P|$ , where there are  $|P|$  rules in grammar  $G$ . Thus, at each locus of input  $t$ , all phrases are tentatively begun. Later discussion covers a more efficient predictive method. For clarity, this is omitted from early, more important details.

Processing of stateset  $S(i)$  begins after its initialization (above). A procedure **SCAN** is called for each state in  $S(i)$ . **SCAN** checks various correspondences of input token  $t(i)$  against terminal symbols in  $\text{RHS}$ s of rules. Only when **SCAN** has been called for all of stateset  $S(i)$  is further processing of  $S(i)$  performed.

Once **SCAN** is done, **COMPLETER** substitutes all final states of  $S(i)$  into all other analyses which can use them as components. **COMPLETER** is the difficult phase of this algorithm because it is intimately tied to implementation considerations, especially space conservation.

When the above are finished for  $S(i)$ , analysis begins similarly for  $S(i+1)$ . Processing terminates at stateset  $S(n+2)$ , which is treated as a special case.

### SCAN

**SCAN** handles states of  $S(i)$ , checking input  $t(i)$  with requirements of states in  $S(i)$  and various error hypotheses. These **SCAN** actions are given and explained below for a state  $(p, j, f, e)$  in stateset  $S(i)$ . "Add" has special meaning, as discussed previously.

1. If  $c(p, j) = t(i)$  then add—if possible— $(p, j+1, f, e)$  to  $S(i+1)$ . This case is a *perfect match*.
2. If  $c(p, j)$  is terminal but not equal to  $t(i)$ , then add  $(p, j+1, f, e+1)$  to  $S(i+1)$  when possible. This is a *mutation-error hypothesis*.
3. If  $c(p, j)$  is terminal, then add  $(p, j+1, f, e+1)$  to  $S(i)$ , if it is possible. This *deletion hypothesis* assumes that component  $c(p, j)$  is not available, that it was deleted from the input.
4. Add  $(p, j, f, e+1)$  to  $S(i+1)$  if possible. Here assume that  $t(i)$  should not be in the input: an *insertion hypothesis*.

All **SCAN** actions are *tried*. The recognizer is making guesses at any point as to which action is right. To be correct, the recognizer covers all possibilities. No context can help cut possibilities, since with errors context is not reliable. It is very crucial that Earley's algorithm limits combinatorics via merging and sharing of states.

The sorted order of states in  $S(i)$  becomes important in **SCAN**'s pass. Those states which may be added to  $S(i)$  from **SCAN** actions on  $(p, j, f, e)$  in  $S(i)$  lie ahead of  $(p, j, f, e)$  since ordering is via increasing values of  $j$ . This insures that a single **SCAN** pass is adequate.

*Two Examples.* Partial analyses depicted in Figures 3 and 4 illustrate **SCAN** actions. States shown are representative of those occurring with an augmented Earley's recognizer.

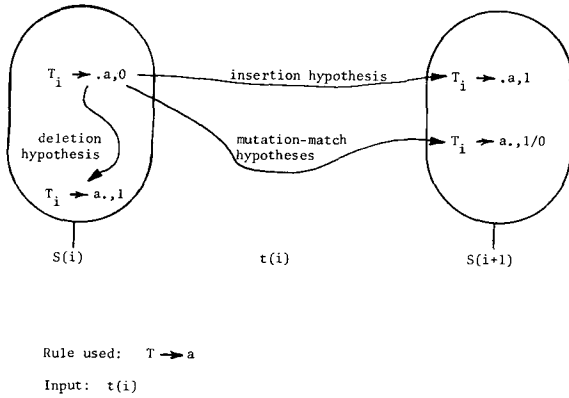
Figure 3 presents **SCAN** actions on a state representing  $T \rightarrow \cdot$ . a. Three paths represent three error hypotheses which can hold at one time, viz. deletion error, insertion error, and mutation/match. **SCAN** applies these possibilities to each state. Error counts appear after each  $\text{RHS}$  of a state.

Figure 4 partially depicts an error analysis of  $t(1)t(2) = a$  using a grammar from Figure 2. Many states are derived in more than one way. Some paths are discarded subsequently as nonoptimal, while others create new, diminished, error counts for states.

### COMPLETER

Upon completion of the pass of **SCAN** over  $S(i)$ , **COMPLETER** begins its run. **COMPLETER** handles substitution of final states in  $S(i)$ . Each final state represents recognition of some nonterminal (phrase). Final states have pointers back to the stateset of their origin. In these originating (or *prior*) statesets may lie stranded states awaiting recognition of some nonterminal component. (Observe that **SCAN** does nothing special for nonterminals.) **COMPLETER**'s task is to search back for stranded states awaiting nonterminals. For example, suppose  $U_p \rightarrow W.Va$ ,  $e$  is a stranded state in  $S(x)$ , and that **COMPLETER** substitutes final state  $V_x \rightarrow \gamma., e'$  which

Fig. 3.



is in  $S(i)$ ; this results in a state  $U_p \rightarrow WV.a.e+e'$  being added to  $S(i)$ . Notice that errors are additive.

A problem occurs among final states in stateset  $S(i)$  which have a common-valued backpointer  $f$ . COMPLETER should not repeat substitutions, yet Figure 5 demonstrates that unless final states are carefully selected, some COMPLETER substitutions may be wasted.

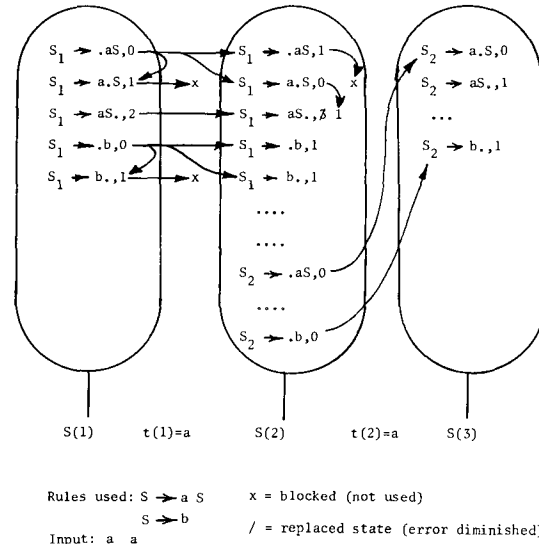
Consider details of Figure 5. If state  $Y_1 \rightarrow b.,1$  in  $S(2)$  is first substituted back into  $S_1 \rightarrow .Y.,0$  of  $S(1)$ , then state  $S_1 \rightarrow Y.,1$  is added to  $S(2)$ . This COMPLETER action will have to be redone, since state  $U_1 \rightarrow a.,0$  in  $S(2)$  is substituted back into  $Y_1 \rightarrow .U.,0$  of  $S(1)$ , and this result into  $S_1 \rightarrow .Y.,1$  of  $S(2)$  is replaced by  $S_1 \rightarrow Y.,0$ .

To avoid futile COMPLETER substitutions of nonoptimal final states, COMPLETER first substitutes final states in  $S(2)$  whose errors are fewest. A least-errors final state of  $S(2)$  will never have its error count diminished by states entering  $S(2)$  via COMPLETER substitution of yet unselected final states in  $S(2)$ . This follows because state  $s$  entering  $S(2)$  via COMPLETER has errors  $e+e'$  at least equal to errors  $e'$  in the final state in  $S(2)$ , which satisfied the variable it was waiting for.

**Final State Selection Given Pointer  $f$ .** A rule for COMPLETER's selection of final states may be stated: Among yet unused final states in stateset  $S(i)$  with common backpointers  $f$ , COMPLETER selects a final state whose error count  $e$  does not exceed error counts of any other. The selected final state is removed from the list of unused eligibles, and substituted. Substitutions are guaranteed sound. If a selected final state is modified by any other final state, this modification has been accomplished already. Alternately, since a selected final state is least-errors among those awaiting substitution (having  $f$  in common), there is no chance that it will be modified to fewer errors by any yet-to-be selected final state in  $S(i)$ .

**Selecting Backpointer  $f$ .** Until now discussion has assumed that COMPLETER selects unused final states from groups with common backpointers  $f$ ; constraints upon values of  $f$  have been skirted. Backpointer  $f$  determines where a subanalysis began. This is very im-

Fig. 4.



portant since groups of final states in  $S(i)$  can be ordered by their ability to influence one another *across values of backpointer  $f$* . Given two final states  $s = (p, p+1, m, e)$  and  $s' = (q, q+1, m+1, e')$  in  $S(i)$ , it is possible for  $s'$  to be substituted into a state with backpointer  $f = m$  in stateset  $S(m+1)$  and thereby modify  $s$  in  $S(i)$ . There is no possibility of  $s$  influencing  $s'$ . To do so,  $s$  would have to satisfy a component in the RHS of the rule for  $s'$ . This is impossible since  $s$  spans more input than does  $s'$ .

COMPLETER for  $S(i)$  works first with a group of final states with the highest possible backpointer ( $f=i$ ) and then successively considers each group of lower-valued backpointers until all have been used. In this manner, any interaction of  $S(i)$ 's final states across backpointer values  $f$  is kept in front of COMPLETER—it will not miss pertinent final state error changes in  $S(i)$  on its single COMPLETER pass, nor will any substitutions have to be redone.

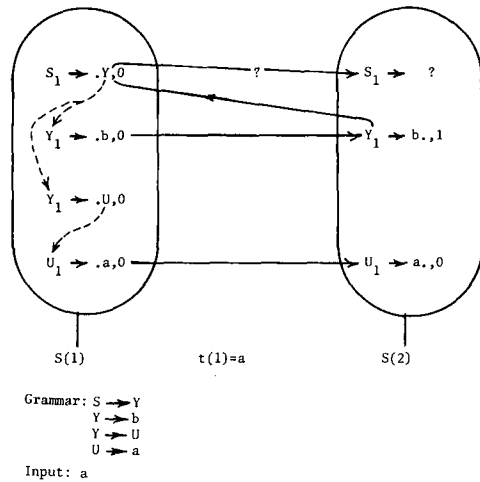
### COMPLETER in Detail

This section describes exactly how COMPLETER of  $S(i)$  functions on each group of final states in  $S(i)$  with common backpointers  $f=x$ . Values for  $f$  are chosen in descending order as indicated above, ranging from  $f=i$  to, finally,  $f=1$ . Each value  $f=x$  determines an associated prior stateset  $S(x)$ .

Two ways that COMPLETER can substitute final states of  $S(i)$  into states of  $S(x)$  are:

**S1.** Select final states  $(p, p+1, x, e)$  in  $S(i)$  and check in stateset  $S(x)$  for stranded states awaiting  $L(p)$ . Earley's algorithm does this efficiently with the help of lists which link all states in stateset  $S(x)$  awaiting a particular variable  $L(p)$ . Least-errors substitutions for COMPLETER have been mentioned in this context.

Fig. 5.



**S2.** Search through  $S(x)$  and with each stranded state awaiting nonterminal  $W$ , check for final states  $(p, p+1, x, e)$  in  $S(i)$  such that  $L(p) = W$ . Substitute a least-errors final state of those available. This converse strategy dispenses with lists for  $S(x)$ , although at a cost of always running in time  $O(n^3)$ . If a grammar is unambiguous, Earley's linked lists allow  $O(n^2)$  execution times. However, unrestricted errors force  $O(n^3)$  times anyway. S2 conserves memory otherwise used for pointers.

COMPLETER strategy for substituting final states of  $S(i)$  into  $S(x)$  is mixed. A first Stage S1 (a relaxation) is performed to insure that final states in  $S(i)$  have least-errors when substituted into  $S(x)$ . Discussion on Figure 5 contained some ideas on this matter, which will be amplified in detail. Stage S2 performs substitutions into  $S(x)$  via the second strategy. Principal motives for dropping Earley's algorithm here involve a desire to save space (no pointers in  $S(x)$ ), and an attempt to push much of storage into a sequential organization. These assumptions underlie the whole COMPLETER philosophy.

**Stage S1: Stabilization.** Let  $S(x)$  be typical of the  $i$  statesets associated with a backpointer value  $f = x$  for  $S(i)$ . COMPLETER begins by substituting (details given shortly) all final states in  $S(i)$  with common pointers  $f = x$  into all possible stranded states in  $S(x)$  with pointers  $f = x$ . The result is a set of final states in  $S(i)$ ,  $f = x$ , which have minimal-error counts  $e$ .

Let  $m$  be a state with fewest errors among yet unselected final states in  $S(i)$  with pointers  $f = x$ . Where possible, COMPLETER substitutes  $m$  into states  $(p, j, x, e)$  of  $S(x)$ . Let a successful result be  $V_x \rightarrow \alpha.\beta, e$ . COMPLETER computes additional errors necessary to delete part  $\beta$  of the RHS, that is, to map (via deletions)  $\beta$  into  $\epsilon$ . This is easily accomplished with a table NULLVR of least-errors counts for deletion of any variable  $W$

in grammar  $G$ . Let  $e'$  be the null cost of  $\beta$ . COMPLETER builds final state  $V_x \rightarrow \alpha\beta, e+e'$  and adds it to  $S(i)$ , if possible. This stage of COMPLETER is concerned only with least-errors counts in final states ( $f=x$ ) of  $S(i)$ . Other possible results are deferred for S2. Stage S1 has a finite bound since states with  $f=x$  are limited in both  $S(i)$  and  $S(x)$ . S1 continues until all final states ( $f=x$ ) in  $S(i)$  have been selected.

**Stage S2: Substitution.** Stage S1 assures that all final states with  $f = x$  in  $S(i)$  are of least-errors. COMPLETER stage S2 then searches through stateset  $S(x)$  and with each stranded state awaiting non-terminal  $W$ , checks for final states  $(p, p+1, x, e)$  in  $S(i)$  such that  $L(p) = W$ . S2 substitutes a least-errors final state of those available in  $S(i)$ . The result is added to  $S(i)$ .

**Other COMPLETER Actions.** SCAN is used on COMPLETER's "add" actions to  $S(i)$  since the scanner will not be called again for  $S(i)$ . For example, if COMPLETER Stage S2 adds a state of the form  $A \rightarrow aV.cXY$  to  $S(i)$ , then SCAN is used to process the "c" relative to input  $t(i)$ . States for  $A \rightarrow aVc.XY$ ,  $A \rightarrow aVcX.Y$ , and  $A \rightarrow aVcXY$ . are added to  $S(i)$ . The latter two states are derived from SCAN of  $A \rightarrow aVc.XY$  using table NULLVR of null string matchings (explained below). SCAN is applied to *any* state added to  $S(i)$  during COMPLETER, including final states derived in stage S1.

Matchings which span no input are represented in stateset  $S(i)$  by final states of the form  $(p, p+1, i, .)$ . Although COMPLETER could handle null-string final states, such null matching is duplicated work since it is independent of input and done identically at each stateset. Table NULLVR of least-errors variable deletions should be built prior to regular processing. With such a table, SCAN of  $S(i)$  can handle null-phrase cases; COMPLETER of  $S(i)$  need not even consider final states in  $S(i)$  with pointer  $f=i$ . SCAN's deletion-error mechanism then handles all null-matched variables via a rule: If possible, add  $(p, j+1, f, e+NULLVR(c(p, j)))$  to  $S(i)$ .

This completes discussion of a basic least-errors algorithm. As detailed, recognition is bottom-up. However, table NULLVR introduces flexibilities which can be exploited: By converting the algorithm into a top-down recognizer, one may avoid inefficiencies in blanket initializations of statesets.

### A Top-Down Version

Once COMPLETER of stateset  $S(i)$  is freed from doing final states with pointers  $f=i$ , this group of states with  $f=i$  can be handled at any time during processing of stateset  $S(i)$ . Other SCAN and COMPLETER actions on stateset  $S(i)$  are independent of SCAN actions upon states in  $S(i)$  with pointers  $f=i$ : Table NULLVR includes any cases which might arise.

The recognizer runs SCAN and then COMPLETER over all states  $f \neq i$  in  $S(i)$ , adhering to the order just established. All variables which could be useful to states in  $S(i)$  become known. For example,  $W$  is useful to

$Y \rightarrow a.Wb$ . The recognizer marks vector location  $PRED(W)$  for each useful phrase  $W$ .

Prior to parsing, the recognizer builds an  $|N| \times |N|$  *predictive matrix*, where there are  $|N|$  variables in grammar  $G$ . If  $W \xRightarrow{*} \alpha X \beta$  and via errors  $\xi: \alpha \rightarrow \epsilon$ , then  $W \xRightarrow{*}_{\xi} X \beta$ . Thus  $W$  predicts (via grammar  $G$  and errors) phrase  $X$ . An appropriate entry is placed in the predictive matrix. Then during recognition, whenever  $PRED(W)=1$  is made,  $PRED(X)=1$  should be set also.

Omitting states with  $f=i$ , the recognizer has performed SCAN and COMPLETER passes for  $S(i)$ . It then initializes group  $f=i$  with all requests from request vector  $PRED$ : If  $PRED(W)=1$ , it adds states  $(p,1,i,0)$  to  $S(i)$ , provided that  $L(p) = W$ . SCAN is then performed over (only) this  $f=i$  group of states in  $S(i)$ . COMPLETER is not necessary because, in conjunction with SCAN, NULLVR replaces it in this special case of  $f=i$ . These initializations for group  $f=i$  and associated SCAN actions constitute the PREDICTOR phase for stateset  $S(i)$ .

Only phrases which could be useful to stranded states in  $S(i)$  have been started in  $S(i)$ . Analysis is "predictive." This PREDICTOR is very much weaker than usual since errors allow many more derivational possibilities. Discussion in Section 4 will include easy ad hoc constraints on the error map  $\xi$ ; such constraints allow PREDICTOR to perform rather effectively.

### The Algorithm

State  $Z_1 \rightarrow .S \dashv, 0$  is added to an empty  $S(1)$ . Proper  $PRED(\cdot)$  entries are made using the predictive matrix on  $S$ . SCAN for  $f=i=1$  is performed. Thus  $S(1)$  begins with PREDICTOR. Usual SCAN and COMPLETER passes are unnecessary for  $S(1)$ .

SCAN, COMPLETER, and PREDICTOR are performed for statesets  $S(2)$  through  $S(n+1)$ .

The final result is state  $Z_1 \rightarrow S \dashv, e$  in stateset  $S(n+2)$ . This corresponds to matching phrase  $Z$  to input  $t(1) \dots t(n) \dashv$ . The actual four-tuple is  $(1,3,1,e)$ , and  $e$  is least-errors for recognition.

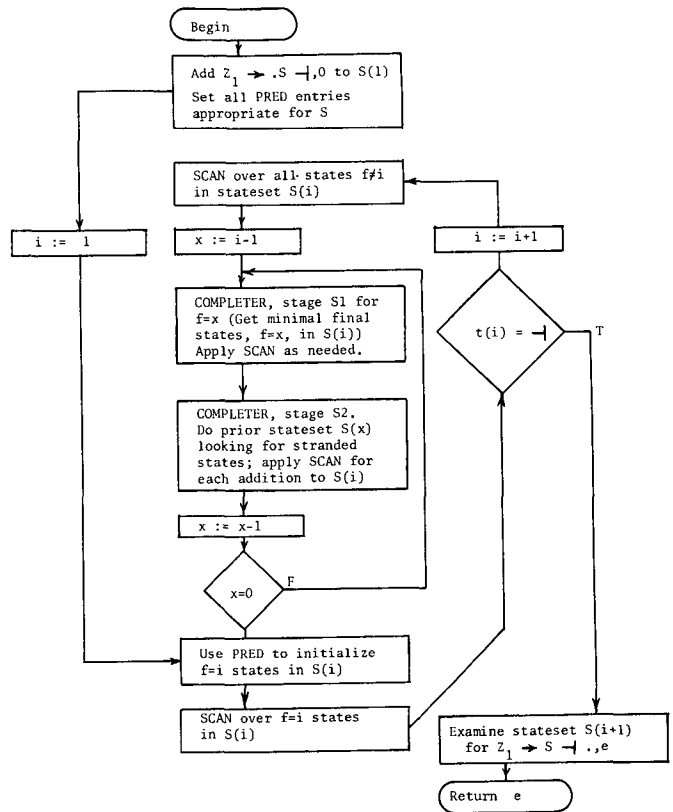
Figure 6 presents an outline of the algorithm.

**Implementation.** This particular implementation assumes random access memory. Each state occupies one word. States  $(p,j,f,\cdot)$  are sequentially ordered in a stateset,  $j$  within  $p$  within  $f$  ( $f$  descends from  $i$  to 1). Each sequential block of states  $(p,j,f,\cdot)$  with common backpointer  $f$  has a header node which gives the backpointer value  $f$ . This header occupies one word: No backpointer  $f$  is explicitly stored in any state's storage word. Headers serve adequately and conserve memory. A vector points to the origin of each stateset. Storage for the stateset uses ascending addresses from the vector indication.

A number of practical factors influence the organization:

(i) Sequential blocks eliminate any need for pointers in states. This is quite important when programming the algorithm since it does make heavy demands upon storage. General unrestricted errors change the problem

Fig. 6.



quite a bit from error-free analysis. Pointers are necessary for unambiguous and deterministic cases which Earley's implementation can take advantage of, but unrestricted errors preclude special cases.

(ii) A large portion of storage— $O(n^2)$ —can be put onto magnetic tape, leaving only a linear fraction— $O(n)$ —in random access memory. If  $S(i)$  is current, then it and  $S(i+1)$  should be in random access memory. Stage S1 should be done with states  $f=x$  of  $S(x)$  in a fixed size random access buffer.

(iii) Sequential methods can be very fast.

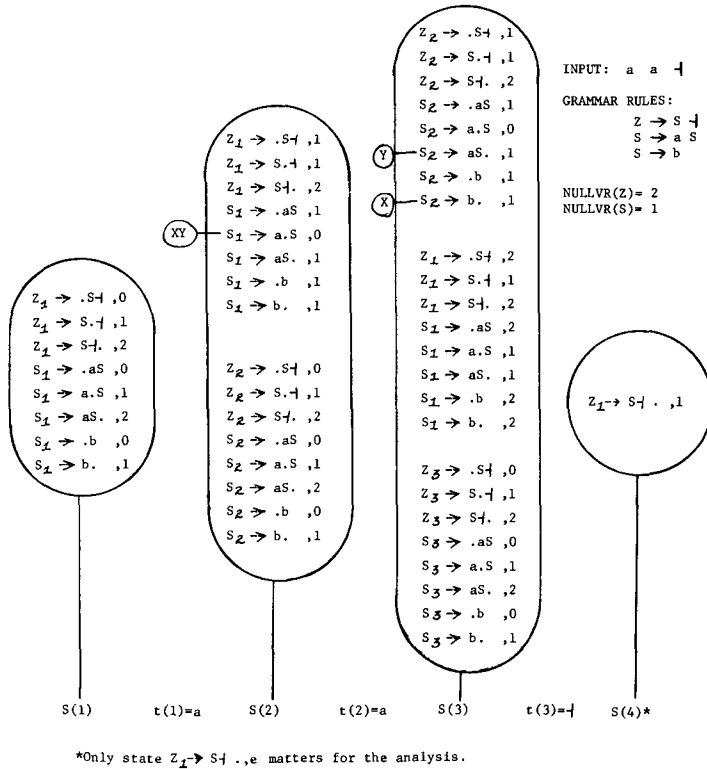
One theoretical consequence of using sequential storage organizations is that a multitape Turing machine is fairly easy to program for the algorithm (Lyon, 1972 [23]). Such an implementation runs in time  $O(n^3 \log n)$ . A  $\log n$  factor arises because tape fields must carry error counts  $e$  which in an ordinary  $O(n^3)$  context-free recognition hold either 0 or 1. Error is  $O(n)$ . Sufficient cells must exist in each tape field for a binary number of similar magnitude to the error. Thus tapes are  $O(\log n)$  longer than usual, and so are run times.

States in  $S(i)$  and  $S(i+1)$  are addressed associatively via  $p, j$ , and  $f$ . There is no search to find any state in either stateset. This is not strictly necessary for prior statesets  $S(i-1) \dots S(1)$ .

Headers actually become useful only when a stateset



Fig. 7.



$S(i)$  is completely processed by SCAN, COMPLETER, and PREDICTOR. At this point many states in  $S(i)$  are eliminated, since only those awaiting variables, "stranded," are worth saving. Stateset  $S(i)$  is compressed to conserve storage. Contents of  $S(i)$  will not change from this point forward.

Table NULLVR gives least-errors deletion counts for each variable  $V$ , as in the error/grammar derivation  $V \xrightarrow{*} \epsilon$ . Naturally if a rule  $V \rightarrow \epsilon$  exists or a true grammatical derivation  $V \xrightarrow{*} \epsilon$  is possible, then  $\text{NULLVR}(V) = 0$ .

**Organization Comments.** The SCAN pass over  $S(i)$  establishes a basis for COMPLETER. If SCAN actions on some states resident in  $S(i)$  were deferred until after S1 phases of  $S(i)$ , these S1's could be invalid because of new, least-errors final states from SCAN. SCAN actions are also applied in stage S2 of COMPLETER. However, any special cases which could influence S1 have been accounted for in S1 through use of NULLVR. SCAN associated with phases S1 and S2 differs from usual SCAN. In the former cases there is a definite lower bound on errors, namely, the errors of the back-substituted state. States which get into  $S(i)$  through routes other than COMPLETER of  $S(i)$  are not so easily assigned lower error bounds. Consequently a SCAN pass occurs prior to COMPLETER so that these cases are removed as a problem to COMPLETER.

Part S1 of COMPLETER is actually a relaxation method. Each selected final state  $m$  sets a minimal error count for results of substitutions which use  $m$ . Since each selected final state  $m$  has at least as many errors as any prior selections of S1,  $m$  cannot influence them. Furthermore, only final states enter  $S(i)$  via actions of S1; S1 only stabilizes error counts for final states in  $S(i)$ . Other S2 actions could be taken, but are deferred until S2.

S2 uses least-errors final states in  $S(i)$  to perform substitutions. Supplementary SCAN actions add to  $S(i)$  and  $S(i+1)$  those states which a SCAN pass failed to provide because the necessary seed state was not in  $S(i)$ .

*Example.* Figure 7 presents a full analysis of the input and grammar shown in partial detail in Figure 4. For input "a a -" and grammar  $Z \rightarrow S -$ ,  $S \rightarrow a S$ ,  $S \rightarrow b$ , there are two corrections possible. One is "ab -" with one mutation-error assumed and the other "aab -", which assumes that a "b" was deleted. These two solutions share a state labeled (XY) in  $S(2)$  and are completed by either states (Y) or (X) in  $S(3)$ .

### An Upper-Bound on Storage

Each state  $(p, j, f, e)$  has bounded parameters  $p, j, f$ , and  $e$ . For fixed values of  $p, j$ , and  $f$ , only one state exists in stateset  $S(i)$ . Parameter  $f$  ranges from 1 to  $i$  in  $S(i)$ . Grammar  $G$  determines finite parameters  $p$  and  $j$ . Storage for  $S(i)$  is thus  $O(i)$ . It follows that, for  $n+2$  statesets of a complete analysis, total storage is  $O(n^2)$ .

### A Bound on Execution

For any stateset  $S(i)$ , the SCAN pass runs over  $O(i)$  states. There are  $i-1$  COMPLETER S1 stages, each taking time proportional to  $|P|^2$ , where  $|P|$  is the number of productions in  $G$ . COMPLETER for  $S(i)$  also passes over  $i-1$  prior statesets,  $S(i-1)$  through  $S(1)$ . Each prior stateset  $S(j)$  has  $O(j)$  members, and each "stranded" state in  $S(j)$  may require a bounded number of computations in the substitution process. Thus COMPLETER for  $S(i)$  executes in  $O(i^2)$ . Total COMPLETER time for all analysis is  $O(n^3)$ . COMPLETER limits recognition speed.

### 4. Practical Constraints

There are ad hoc limitations which can be attached easily to the error hypothesis mechanism.

**Local and Global Limits.** Errors which involve components of RHSs which are terminals are called local errors. For example, mutation errors are always local, because they involve a terminal in an RHS of a rule which a state represents. Counter  $e$  in  $(p, j, f, e)$  is a global error count.  $e$  is a total accumulation of errors. Define a supplemented state  $(p, j, f, L, e)$ , where all but  $L$  are as before. Local error  $L$  is incremented to  $L+1$  if and only if  $e := e+1$  via a mutation or character deletion hypothesis. A state representing  $X \rightarrow V$  can

Fig. 8.

INPUT: "v" = ( "vc" \* "vc" + + ) -

GRAMMAR:

```

Z → [PG] -
[PG] → [PG] ; [ST]
[PG] → [ST]
[FC] → [TM]
[FC] → [FC] * [TM]
[AE] → [FC]
[AE] → [AE] + [FC]
[AS] → "v" = [AE]
[UC] → [AS]
[UC] → goto "L"
[UC] → "L" : [UC]
[IC] → if "be" then
[IS] → [IC] [UC]
[CS] → [IS]
[CS] → [IS] else [ST]
[CS] → "L" : [CS]
[ST] → [UC]
[ST] → [CS]
[TM] → "vc"
[TM] → ( [AE] )

```

PERFORMANCE:	Local errors	States	Time (seconds)
	6	2800	17.3
	5	2690	16.6
	4	2531	14.8
	3	2337	13.9
	2	2062	11.8
	1	1300	6.1
	0	—	0.6

have no local errors. Clearly  $L \leq e$  for any state.

Limits can be set upon how many local and global errors a state can have. An attempt to add a state which exceeds a local or global error limit—called a cutoff—results in a failure. Whatever analysis the state represents is lost.

Cutoffs for local and global errors thin numbers of states generated at any stage of analysis. Justifications for cutoffs arise from observation of actual practice. If a programmer meant to write a conditional statement, it is unlikely that he would omit IF, THEN, and ELSE. Consequently, a local cutoff of  $L \leq 2$  is not unreasonable for a rule

*conditional ::= IF boolean THEN statement ELSE statement*

Any action which serves to limit states will augment efficiency. Figure 8 demonstrates that changes in numbers of states linearly influence execution times, which is hardly surprising, given the way COMPLETER works. Global errors in Figure 8 are set to  $e \leq 32$ . Six runs were made with  $L \leq r$ , and  $r = 1, \dots, 6$ . An additional run established a time and state census for a correct input with no local errors. The first admission of errors entails a ten-fold increase in execution time for that particular grammar.

**Fiducial Characters.** Another practical approach reserves certain terminals as privileged. There are four areas of concern: the range and the domain of mutation hypotheses, the range of insertion errors, and the domain of deletion mappings. By restricting terminals which can occur in these areas, some characters become fiducial (error-free) markers. The result is to partition analysis into much smaller pieces.

Fiducials are fairly commonplace. FORTRAN is an example of a card oriented language where end-of-state-

Fig. 9.

INPUT: LABL : BEGN DCLR ; DCLR ; DMMY ; END -

GRAMMAR:

```

Z → PRGM -
PRGM → BLOK
PRGM → CMST
UBST → ASGN
UBST → GOTO
UBST → DMMY
UBST → PROC
BCST → UBST
BCST → LABL : BCST
UNCL → BCST
UNCL → CMST
UNCL → BLOK
STMT → UNCL
STMT → CNDL
TAIL → STMT END
STMT → FORS
TAIL → STMT ; TAIL
BKHD → BKHD ; DCLR
BKHD → BEGN DCLR
ULCD → BEGN TAIL
ULBK → BKHD ; TAIL
CMST → ULCD
CMST → LABL : CMST
BLOK → ULBK
BLOK → LABL : BLOK

```

PERFORMANCE:	Fiducials (no errors)	States	Time (seconds)
	---	2295	13.3
	;	1567	9.3
	; DCLR	984	5.7
	; DCLR :	730	4.5

CONSTRAINTS: Local error cutoff = 2  
Global error cutoff = 5

ment coincides with the end of a card. Figure 9 presents experimental results with an ALGOL-like grammar and sets of fiducials. Fiducials indicated in the table are prohibited from entering any errors whatsoever. Results are fairly predictable: As input characters are constrained performance improves, quickly at first. Common separators such as “;” in ALGOL have the most fiducial value.

## 5. Observations

With reasonable constraints upon errors, the recognizer might serve as a diagnostic scanner. First, a very fast syntactic analyzer would be called to test for correctness. Upon discovery of discrepancies, input would be passed for a least-errors analysis. Corrected text would be returned as output.

The algorithm has been programmed. Error loci are saved in bit strings which are carried with each state. COMPLETER actions include the ORing of a final state's bit string into the string associated with a stranded state. This simple arrangement works well for short inputs. Unlike a system of pointers to components, the bit strings are complete upon reaching stateset  $S(n+2)$ . Error loci are available without backtracking along pointer paths.

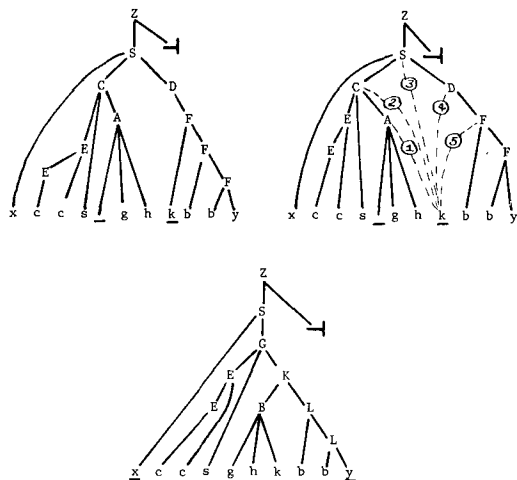
One can speed things up. For instance, in any S1 relaxation final states  $(p, p+1, x, e)$  and  $(p', p'+1, x, e')$  such that  $L(p) = L(p')$  and  $e' \leq e$  can be collapsed into one final state  $(p', p'+1, x, e')$ . This reduces relaxation effort from  $O(|P|^2)$  to  $O(|N|^2)$ .

Counting solutions requires attention to yet finer details. A counter  $a$  is inserted into each state, so the

Fig. 10.

Grammar rules	Input and representative error loci (underline)	Total errors	Possible analyses
Z $\rightarrow$ S $\rightarrow$	<u>—</u> c s <u>—</u> b y $\rightarrow$	4	1
S $\rightarrow$ xCD	x c <sup>2</sup> s f g h b <sup>2</sup> y $\rightarrow$	0	1
S $\rightarrow$ uG	<u>x</u> c <sup>2</sup> s g h k b <sup>2</sup> z $\rightarrow$	1	1
C $\rightarrow$ EsA	x c <sup>2</sup> s <u>—</u> g h <u>—</u> b <sup>2</sup> y $\rightarrow$	2	7
D $\rightarrow$ F			
G $\rightarrow$ EsK			
E $\rightarrow$ Ec			
E $\rightarrow$ c			
A $\rightarrow$ fgh			
F $\rightarrow$ bF			
F $\rightarrow$ by			
K $\rightarrow$ BL			
B $\rightarrow$ ghk			
L $\rightarrow$ bL			
L $\rightarrow$ bz			

Fig. 11.



form is  $(p, j, f, e, a)$ . If a state  $(p, j, f, e, a)$  is to be added to  $S(i)$  which has  $(p, j, f, e, a')$ , the result is  $(p, j, f, e, a + a')$ . Since this changes the rule of adding (normal adding would reject), care must be exercised during Stage S2 ( $f = x$ ) of any COMPLETER that no final states with  $f = x$  are generated since these states have been accounted for already in Stage S1.

Counts of components of a state are multiplied to get the state's ambiguity. Null phrases (NULLVR) must have a table NULL# which gives their ambiguity, or count. Naturally no cyclic variables  $A \rightarrow B \rightarrow C \rightarrow A$  can be allowed; otherwise ambiguities could be unbounded. This prohibition is used in the algorithm: In an S1 relaxation, among unselected least-error final states with  $f = x$ , there will be at least one which cannot be influenced by the others. This final state is selected, since its ambiguity counter  $a$  will not change.

Here is an example. Suppose there is a rule  $A \rightarrow B$ . Let  $A_j \rightarrow B, e, a$  and  $B_j \rightarrow abc, e, a'$  be final states in  $S(i)$ . Let  $S(j)$  contain state  $A_j \rightarrow B, 0, 1$ . In  $S(i)$ , state  $B_j \rightarrow abc, e, a'$  cannot be influenced by  $A_j \rightarrow B, e, a$ , so the former is selected first even though error counts  $e$  are the same. The end result is a final state  $A_j \rightarrow B, e, a + a'$  in  $S(i)$ . If chosen in the other order, improper counts for phrase A would result.

As an example of an analysis, consider the LR(1)

grammar and results in Figure 10. Leinius, 1970 [20], devotes some attention to recovery from errors with this grammar, which is also simple precedence. Sentences in Figure 10 should have a form  $xc^nsfghb^mz$  or  $uc^nsghkb^mz$ . Error loci are marked via underline. A locus may contain more than one error.

Analyses for input  $xc^2sghkb^2y$  are interesting and varied. Figure 11 depicts seven interpretations. First, one assumes that an "f" has been deleted and a "b" changed into "k". Next, assume an "f" deletion and the "k" to be inserted. The "k" can associate to five different phrases, given interpretations shown in the center tree of Figure 11. And last, the pair " $x \cdots y$ " could be wrong, i.e. mutations of the correct set " $u \cdots z$ ".

No grammars larger than those which appear in text examples have been tested. With longer inputs and larger grammars both memory and execution times grow uncomfortably. A compiled version—tailored to a specific grammar—might provide further efficiencies. Earley suggests such an approach.

This research has shed some light on attempts to fashion recovery and correction strategies for syntactic scanners. Experience with the programmed algorithm indicates that ALGOL-like phrase structures are extremely fragile. A least-errors global correction is often worse in reconstructing an original sentence than a correction constrained as in Section 4. This suggests that distances between ALGOL sentences are not great. Least-errors recognition then implies whole new sentences, rather than reconstruction of originals. "Robust" grammars might gracefully sustain syntactic damage in sentences without bizarre disintegration of structure; the problem is to find robust features while preserving usefulness.

*Acknowledgments.* Professor Bruce Arden suggested to the author that, for general-error correction, Earley's algorithm might work better than Younger's. The relaxation of S1 of COMPLETER resulted from a discussion with Professor E. Lawler.

Received September 1972, revised April 1973

## References

1. Bellman, R.E. *Dynamic Programming*. Princeton U. Press, Princeton, N.J., 1957.
2. Bollinger, H. The HELP metacompiler, a compiler writing tool. 1968 Annual Spring Meeting, General Motors Committee on Engineering Computations.
3. Chomsky, N. On certain formal properties of grammars. *Information and Control* 2,2 (Feb. 1959), 137-167.
4. Conway, M.E. Design of a separable transition-diagram compiler. *Comm. ACM* 6, 7 (July 1963), 396-408.
5. Earley, J. An efficient context-free parsing algorithm. *Comm. ACM* 13,2 (Feb. 1970), 94-102.
6. Earley, J. An efficient context-free parsing algorithm. Unpubl. Ph.D. diss. Carnegie-Mellon U., 1968.
7. Eggers, B. Zur Theorie und Praxis Selbstkorrigierender Regulaerer Sprachen. Dr.rer.nat. dissertation. Technischen Universitaet Hanover, 1972.
8. Eggers, B. Error reporting, error treatment and error cor-

rection in ALGOL translation, Part II. Second Annual Meeting, G.I. Karlsruhe, Oct. 1972.

9. Evans, A., Jr. An ALGOL 60 compiler. In *Annual Review in Automatic Programming*. Pergamon Press, New York, 1964.
10. Feldman, J.A., and Curry, J.E. The compiler-compiler in a time-sharing environment. Summer Engineering Conf. on Advanced Computer Organization. U. of Michigan, Ann Arbor, Mich., 1967.
11. Feldman, J.A., and Gries, D. Translator writing systems. *Comm. ACM* 11,2 (Feb. 1968), 77-108.
12. Hopcroft, J.E., and Ullman, J.D. Error correction for formal languages. Digital Syst. Lab. Rep. 52. Princeton U., Princeton, N.J., 1966.
13. Hopcroft, J.E. *Formal Languages and Their Relation to Automata*. Addison-Wesley, Reading, Mass., 1969.
14. Irons, E.T. An error-correcting parse algorithm. *Comm. ACM* 6,11 (Nov. 1963), 669-673.
15. James, E.G., and Partridge, D.P. Adaptive correction of program statements. *Comm. ACM* 16,1 (Jan. 1973), 27-37.
16. Jones, C.B. Formal development of correct algorithms: An example based on Earley's recognizer. SIGPLAN Notices 7 (Jan. 1972), 150-169.
17. Kovalevsky, V.A. Sequential optimization in pattern recognition and pattern description. Preprint supplement booklet I, IFIP Cong., Edinburgh, 1968, pp. 1146-1151.
18. La France, J.E. Optimization of error recovery in syntax-directed parsing algorithms. SIGPLAN Notices 5 (Dec. 1970), 2-17.
19. LaFrance, J. E. Syntax-directed recovery for compilers. Ph.D. diss., U. of Illinois at Urbana-Champaign, 1971.
20. Leinius, R.P. Error detection and recovery for syntax-directed compiler systems. Unpubl. Ph.D. diss. U. of Wisconsin, 1970.
21. Levy, J.P. Automatic correction of syntax errors in programming languages. Ph.D. diss., Cornell U., 1971.
22. Lyon, G.E. Least-errors recognition of mutated context-free sentences in time  $n^3 \log n$ . Proc. Sixth Princeton Conf. on Inform. Syst. and Sci. Princeton, N.J., 1972, pp. 115-118.
23. Lyon, G.E. Time  $n^3 \log n$  least-errors recognition of ungrammatical sentences of context-free languages. Mental Health Res. Inst. Comm. # 292. U. of Michigan, Ann Arbor, Mich. 1972.
24. Lyon, G.E. A syntax-directed least-errors recognizer for context-free languages. Unpubl. Ph.D. diss. U. of Michigan, 1972.
25. Peterson, T.G. From a private discussion with the author, March 22, 1972.
26. Peterson, T.G. Syntax error detection, correction and recovery in parsers. Ph.D. diss., Stevens Inst. of Tech., 1972.
27. Peterson, W.W. *Error-Correcting Codes*. The MIT Press, Cambridge, Mass., 1961.
28. Teitelbaum, R. Context-free error analysis by evaluation of algebraic power series. Proc. ACM-SIGACT Fifth Ann. Conf. on Theory of Computing. U. of Texas, Austin, Texas, 1973, pp. 196-199.
29. Wegbreit, B. Studies in extensible programming languages. Ph.D. diss. Harvard U., 1970.
30. Wirth, H., and Weber, H. EULER: A generalization of ALGOL and its formal definition; Parts I and II. *Comm. ACM* 9,1-2 (Jan.-Feb. 1966), 13-35, 89-99.
31. Younger, D.H. Recognition and parsing of context-free languages in time  $n^3$ . *Information and Control* 10,2 (Feb. 1967), 189-208.

Added in proof. Readers may also find the following recent contribution of interest. Graham, S.L., and Rhodes, S.P. Practical syntactic error recovery in compilers. ACM SIGACT-SIGPLAN Symp. on Principles of Prog. Languages, Boston, 1973.

Numerical  
Mathematics

R.A. Willoughby  
Editor

# A Fast Method for Solving a Class of Tridiagonal Linear Systems

Michael A. Malcolm  
University of Waterloo  
and  
John Palmer  
Stanford University

The solution of linear systems having real, symmetric, diagonally dominant, tridiagonal coefficient matrices with constant diagonals is considered. It is proved that the diagonals of the  $LU$  decomposition of the coefficient matrix rapidly converge to full floating-point precision. It is also proved that the computed  $LU$  decomposition converges when floating-point arithmetic is used and that the limits of the  $LU$  diagonals using floating point are roughly within machine precision of the limits using real arithmetic. This fact is exploited to reduce the number of floating-point operations required to solve a linear system from  $8n - 7$  to  $5n + 2k - 3$ , where  $k$  is much less than  $n$ , the order of the matrix. If the elements of the subdiagonals and superdiagonals are 1, then only  $4n + 2k - 3$  operations are needed. The entire  $LU$  decomposition takes  $k$  words of storage, and considerable savings in array subscripting are achieved. Upper and lower bounds on  $k$  are obtained in terms of the ratio of the coefficient matrix diagonal constants and parameters of the floating-point number system.

Various generalizations of these results are discussed.

**Key Words and Phrases:** numerical linear algebra, linear systems, Toeplitz matrices, tridiagonal matrices  
**CR Categories:** 5, 5.1, 5.11, 5.14, 5.17

Copyright © 1974, Association for computing machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.