On Failure of the Pruning Technique in "Error Repair in Shift-Reduce Parsers"

EBERHARD BERTSCH
Ruhr University, Bochum
and
MARK-JAN NEDERHOF
University of Groningen

A previous article presented a technique to compute the least-cost error repair by incrementally generating configurations that result from inserting and deleting tokens in a syntactically incorrect input. An additional mechanism to improve the run-time efficiency of this algorithm by pruning some of the configurations was discussed as well. In this communication we show that the pruning mechanism may lead to suboptimal repairs or may block *all* repairs. Certain grammatical errors in a common construct of the Java programming language also lead to the above kind of failure.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*parsing*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Error repair

1. INTRODUCTION

Error repair techniques have always been essential components of compiler systems. There is a wealth of literature on various approaches to constructing repair modules automatically from the grammar of a given programming language. The technique presented in the article by McKenzie et al. [1995] (ACM TOPLAS 17, 4) stands out as an elegant and efficient, generic method which has been claimed to produce locally least-cost repairs for incorrect strings with respect to arbitrary LR grammars. To speed up the search for possible repair configurations, a technically simple pruning mechanism was also introduced in McKenzie et al. [1995]. The purpose of the present communication is to give evidence that such pruning is unsafe, both in

Both authors are supported by the German Research Foundation (DFG), under grant Be1953/1-1. The second author is further supported by the Priority Programme Language and Speech Technology (TST). The TST-Programme is sponsored by NWO (Dutch Organization for Scientific Research).
Authors' addresses: E. Bertsch, Ruhr University at Bochum, Faculty of Mathematics, Universitätsstraße 150, D-44780 Bochum, Germany; email: eberhard.bertsch@lpi.ruhr-uni-bochum.de; M.-J. Nederhof, University of Groningen, Faculty of Arts, P.O. Box 716, 9700 AS Groningen, The Netherlands; email: markjan@let.rug.nl.

procedure ClearQueue:
"initialize queue to be empty"

procedure Enqueue(< *stack* , *input* >, *inserted*, *deleted*, *cost* , *ptr* , *height*):
"insert new configuration into queue"

function DeleteMin():
"get least-cost configuration and remove it from queue"

function CreateGlobal(value):
"create global variable with initial value *and return pointer to this variable"*

function ↑(ptr) :
"return the variable pointed to by ptr*"*

Fig. 1. Auxiliary routines.

terms of the cost of repairs obtained and in terms of finding any repair at all.

We begin our discussion by restating the algorithms presented in McKenzie et al. [1995]. Then several example grammars and specific cases of error repair applicable to erroneous strings with respect to those grammars are described in detail. It will become apparent that the pruning technique presented in McKenzie et al. [1995] fails in those cases.

2. THE REPAIR ALGORITHM

In this section we present the algorithm from McKenzie et al. [1995]. The original description was given in two phases. First, an informal algorithm was presented which computes the locally cheapest repair of a given error configuration. It does so by generating new configurations that can be reached from the error configuration by means of inserts and deletes. Configurations are maintained in a priority queue ordered by the sum of the costs of the applied inserts and deletes, in such a way that cheapest repairs are considered first.

The second phase is a refinement of this algorithm which restricts the number of configurations that are inserted in the priority queue. Quite roughly, this restriction means that a configuration is ignored if some earlier configuration had the same state on top of the stack. The motivation is that this newer configuration would not lead to any cheaper repairs than the older one, and therefore does not warrant consideration. (In the following sections we will investigate to what extent the assumption is justified.)

We give a high-level description of a combination of both phases.

The second phase was only sketched in McKenzie et al. [1995], but an implementation of the complete repair algorithm (embedded in the framework of Bison) was made available at the Web site mentioned in that paper. The algorithm given here is a compromise between the published algorithm, which is relatively simple but comprises only the first phase in a formal way, and the C code we retrieved from the Web, which comprises both phases but is much more complicated due to several optimizations outside the scope of this communication.

Figure 1 describes some auxiliary operations. They include creation of variables that can be shared and handling of pointers to such variables.

In the present article, the values of such variables will be subsets of LR states. In

```
function Recover(< stack , input >):
begin
  ClearQueue;
  let ptr_init = CreateGlobal(∅);
  Enqueue(< stack , input > , ϵ , ϵ , 0 , ptr_init , 0);
  while queue is not empty do
    let (< q_n ··· q_2 q_1 , t_1 t_2 ··· t_m > , Γ , Δ , cost , ptr , h) = DeleteMin();
    if δ(q_1 , t_1) ≠ error then
      return (Γ, Δ)
    end;
    if Δ = ϵ then
      if q_1 ∉ ↑(ptr)
      then let ↑(ptr) = ↑(ptr) ∪ {q_1};
           for each t ∈ T do
             TreatInsert(< q_n ··· q_2 q_1 , t_1 t_2 ··· t_m > , t , Γ , Δ , cost , ptr , h)
           end
      end
    end;
    let ptr_new = CreateGlobal(∅);
    Enqueue(< q_n ··· q_2 q_1 , t_2 ··· t_m > , Γ , Δt_1 , cost + D(t_1) , ptr_new , 0)
  end;
  return "no repair found"
end Recover.
```

Fig. 2. Main routine based on McKenzie et al. [1995].

actual implementations, such sets may be represented by bitmaps, with a one-to-one correspondence between bits and LR states.

We also assume assignments $\mathrm{I}(t)$ and $\mathrm{D}(t)$ to all input symbols t, representing costs of inserting and deleting individual symbols t, respectively. The cost of inserting a string of symbols is defined to be the sum of the costs of inserting the symbols individually: $\mathrm{I}(t_1 \cdots t_m) = \mathrm{I}(t_1) + \cdots + \mathrm{I}(t_m)$; similarly we define $\mathrm{D}(t_1 \cdots t_m) = \mathrm{D}(t_1) + \cdots + \mathrm{D}(t_m)$. The set of all input symbols is denoted by T. The function δ represents the LR tables [Aho et al. 1986].

The main routine is given in Figure 2. It is to be called with the stack and input as they were right after the last successful shift before detection of an error. It consists of a loop which selects minimal elements from the queue and derives new elements from them by considering possible insertions and then deletions.[1] Such elements are 6-tuples consisting of

—a configuration, derived from the error configuration by means of inserts and deletes, and — where appropriate — shifts and reductions,

—the list of inserts used,

—the list of deletes used,

—the sum of the costs of those inserts and deletes,

—a pointer to a set of LR states that have appeared on top of the stack, and

[1] Allowing insertions and deletions to alternate arbitrarily causes solutions to be found more than once.

```
procedure TreatInsert(< q_n ··· q_2 q_1 , input > , t , Γ , Δ , cost , ptr , h):
begin
  case δ(q_1, t) of
    shift q_s:
      if q_s ∉ ↑(ptr)
      then Enqueue(< q_n ··· q_2 q_1 q_s , input > , Γt , Δ , cost + I(t) , ptr , h + 1)
      end
    reduce A → α:
      DoReduce(< q_n ··· q_2 q_1 , input > , t , A → α , Γ , Δ , cost , ptr , h)
    otherwise:
      skip
  end
end TreatInsert.

procedure DoReduce(< q_n ··· q_2 q_1 , input > , t , A → α , Γ , Δ , cost , ptr , h):
begin
  let len = length of α;
  if len ≤ h
  then let h_new = h − len + 1;
         let ptr_new = ptr
  else let h_new = 0;
       let ptr_new = CreateGlobal(∅)
  end;
  let q_new = δ(q_{len+1}, A);
  TreatInsert(< q_n ··· q_{len+1} q_new , input > , t , Γ , Δ , cost , ptr_new , h_new)
end DoReduce.
```

Fig. 3. Secondary routines based on McKenzie et al. [1995].

—the number of stack symbols by which the stack has grown since creation of the lowest stack in one of the ancestor configurations.

The loop is terminated when a configuration is found for which the LR table contains a nonerror entry. We call such configurations *repair configurations*. Then the associated lists of inserts and deletes are given as the result. The loop may also terminate when the queue becomes empty, in which case no error repair is found.

The main routine makes use of the mutually recursive procedures TreatInsert and DoReduce in Figure 3, which investigate the reductions and subsequent shift that can be performed on a configuration resulting from an insert.

Initially, an empty set of LR states is created at the beginning of Recover. We maintain a pointer to that set. For each configuration derived from the error configuration, we investigate whether its top-of-stack is already in the set of LR states. If so, the configuration is pruned. If not, the top of the stack is added to the set.

There are two occasions when we create a new, initially empty set of LR states. The first one is when we perform a delete. The other one is when the height of the stack shrinks below the height it had at the last time when a new set of LR states was created, or equivalently, when regions of the stack are addressed which have not been investigated before during the current repair attempt. For additional explanation we refer to McKenzie et al. [1995].

3. DEFECTIVENESS OF PRUNING

In this section we show that the pruning mechanism may lead to suboptimal repairs or may cause failure to produce any repair even if one exists. To simplify the presentation, we will assume that the technique used to build LR tables is the SLR(1) technique [Aho et al. 1986]. Our examples below could, however, be adapted to obtain comparable results for LALR(1), LR(1), etc.[2]

The case of suboptimal repair is exemplified by the context-free grammar with the following rules:

$$\begin{aligned} S &\rightarrow AAa \mid b \\ A &\rightarrow cd \end{aligned}$$

where S and A are nonterminals, and a, b, c, and d are input symbols. We assume $\mathrm{I}(a) = \mathrm{I}(c) = \mathrm{I}(d) = 1$ and $\mathrm{I}(b) = 6$; deletion costs are irrelevant to this example. The sets of LR(0) items and the FOLLOW-sets are given in Figure 4. We assume that an SLR(1) parser has been constructed from these data.

Assume that the incorrect input to the parser is the empty string, or more precisely, "#", where # is the end-of-sentence marker. The error configuration is given by $<0\,,\,\#>$. Possible repairs are presented in the form of a search tree in Figure 4. The arrows represent insertions. The input symbols enclosed in parentheses denote the inserted symbols (cf. parameter t in routine TreatInsert); inserted symbols that do not allow any shift or reduction and therefore immediately lead to failure in the next step were omitted from the figure.

A sequence of configurations between a pair of arrows represents a series of reductions and a subsequent shift over the inserted symbol.

Two repairs can be found: one by means of inserting "b", which has cost $\mathrm{I}(b) = 6$, and another one by means of inserting "$cdcda$", which has cost $\mathrm{I}(cdcda) = 5$. The latter is therefore the cheaper repair, but it is never found if the algorithm from the previous section is used, since the corresponding search path is pruned due to the occurrence of state 1 on top of the stack in $<031\,,\,\#>$, after the earlier occurrence in $<01\,,\,\#>$.

That suboptimal repairs are produced may not be a serious problem in practice. However, it may also occur that no repair at all is found even though one does exist. This becomes clear if we remove the second alternative of S from the above grammar:

$$\begin{aligned} S &\rightarrow AAa \\ A &\rightarrow cd \end{aligned}$$

For this new grammar the search tree is similar to that in Figure 4, except that now only one repair exists, with cost $\mathrm{I}(cdcda) = 5$, and obtaining this repair depends on allowing two occurrences of state 1. Therefore, the pruning mechanism will prevent any repair from being found.

The previous pair of examples deals with two occurrences of a state which arise in distinct configurations. There are, however, also examples where pruning due to double occurrences in the *same* configuration leads to blocking of the cheapest repair. (Failing to find *any* repair if one exists is however not possible in this

[2]The approach in McKenzie et al. [1995] uses default reductions in addition to LALR tables.

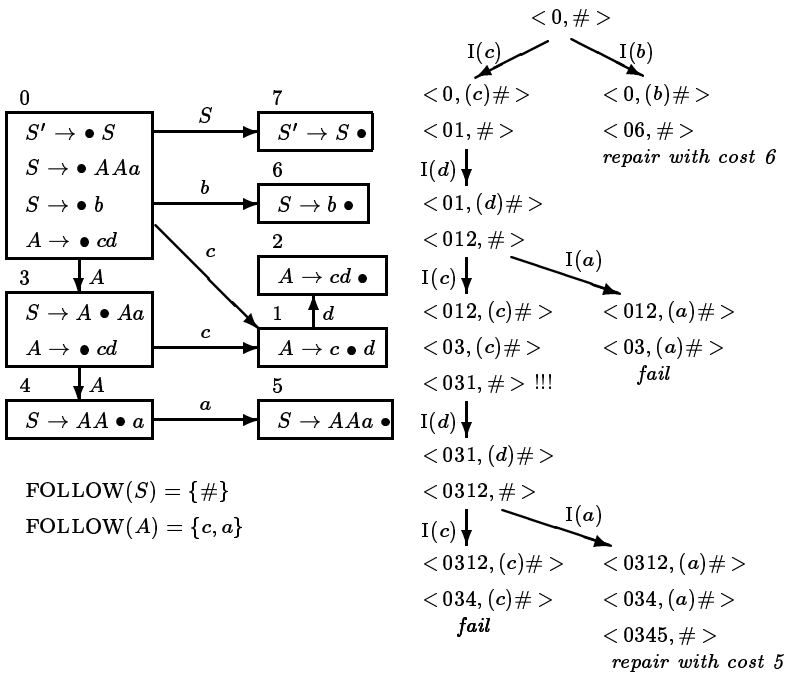

Fig. 4. The collection of sets of LR(0) items, in pictorial form, the FOLLOW-sets, and the search tree of error repairs.

situation, except in the case of validation, as will be explained later on in this section.)

One simple example is the grammar

$$S \;\to\; aa \mid aSbc$$

and the incorrect input "$ac\#$", where we assume $\mathrm{D}(c)$ is sufficiently high. The cheapest repair is then only reached if there are two occurrences of the state corresponding with the set of items $\{S \to aa \bullet, S \to a \bullet a, S \to a \bullet Sbc, S \to \bullet aa, S \to \bullet aSbc\}$.

We will not discuss this example in depth, but instead we demonstrate that the phenomenon can be generalized to arbitrary $k > 1$, i.e., that for any k we can find a grammar, an input string, and a cost assignment, such that the optimal repair cannot be reached unless some configuration is encountered which has k occurrences of the same state. Such a grammar is given by

$$\begin{aligned} S &\to A_1 \mid A_2 \mid \ldots \mid A_{k-1} \\ A_1 &\to aA_2bc \mid a \\ A_2 &\to aA_3 \mid \\ &\ldots \\ A_{k-1} &\to aA_1 \end{aligned}$$

We consider the input "$c\#$" and assume the cost assignment is such that the costs of deletions are high enough for repairs with only insertions to be cheaper

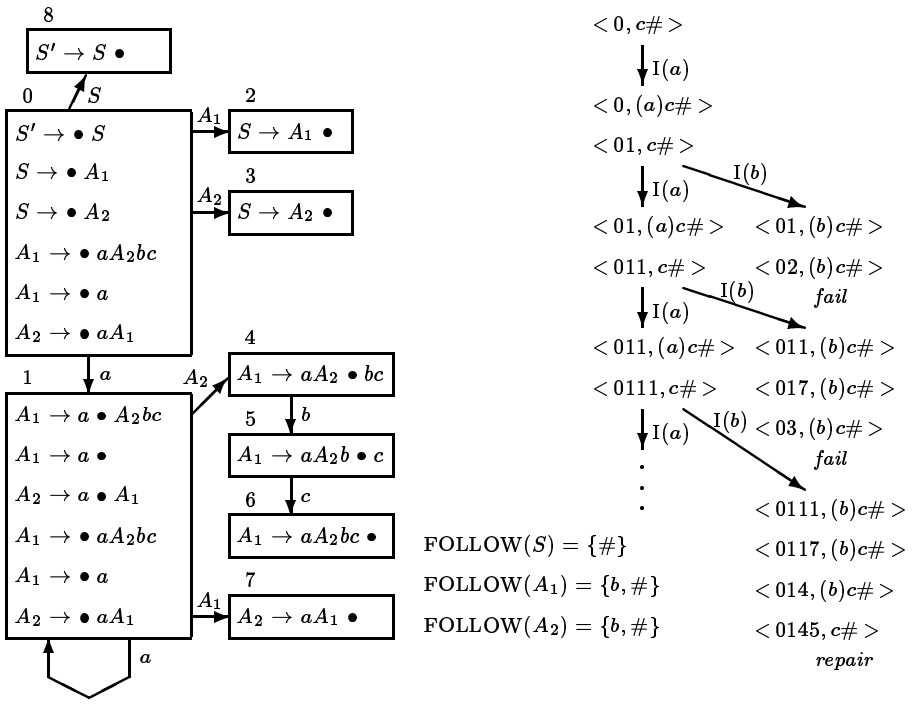

Fig. 5. $k = 3$ occurrences of a state in a single stack.

than those with both deletions and insertions.

Figure 5 shows the LR automaton and the search tree for $k = 3$; we have omitted parts of the search tree involving deletions, since deletions are regarded to be prohibitively expensive. The cheapest repair involves $k = 3$ occurrences of state 1 in a single configuration.

Until now, we have not considered *validation*, which entails that commitment to a repair is made only after a certain number of tokens have been processed successfully. If however a further error is encountered too soon, the original error configuration is revisited; the second cheapest repair is applied; and again an attempt is made to validate, etc.

In the case of validation with as little as one further input token, more examples of failure of the pruning mechanism are observed, when validation does not succeed and control is returned to the repair algorithm. (This is mentioned as a "subtle" interaction in McKenzie et al. [1995].) For example, with grammar $S \to aSb \mid c$ and incorrect input "$bb\#$", the state reached by reading a is needed twice for producing a repair that can be validated.

Validation complicates matters considerably. Its full treatment is beyond the scope of the present article, and therefore further discussion is omitted.

All examples discussed so far involved multiple occurrences of states in a single path in the search tree. Next, we discuss the interaction between different paths that share variables. In such cases, the pruning mechanism may lead to either

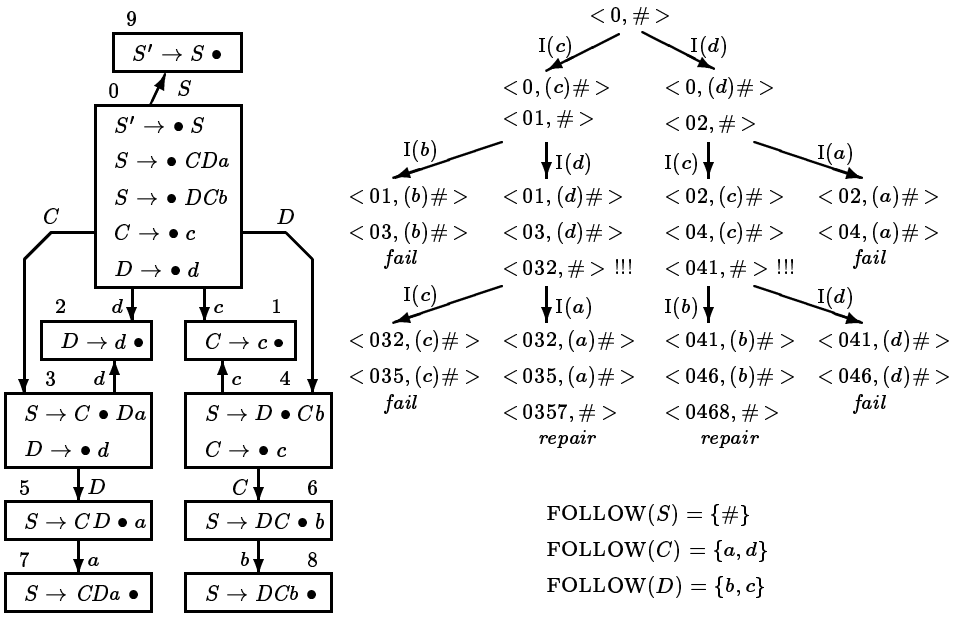

Fig. 6. A case of two search paths blocking each other.

suboptimal repairs or even failure to produce any repair. The latter situation occurs for the grammar with the rules

$$\begin{aligned} S &\to CDa \mid DCb \\ C &\to c \\ D &\to d \end{aligned}$$

the input "#", and a cost assignment such that $\mathrm{I}(a) = \mathrm{I}(b) = \mathrm{I}(c) = \mathrm{I}(d) = 1$. Two repairs exist with equal costs: either "*cda*" or "*dcb*" may be inserted. The LR automaton and the search tree are given in Figure 6.

The two alternatives for S are tried in an interleaved manner, due to the priority queue. The occurrence of state 1 after inserting c pertaining to the first alternative for S *blocks* further processing of the second alternative at $< 041, \# >$. In a similar way, the first alternative is blocked due to the second alternative at $< 032, \# >$, since state 2 has then already occurred at $< 02, \# >$. Hence, no repair at all is found.

We summarize our findings:

—Pruning due to two occurrences of a state in the same search path may lead to suboptimal repairs or to failure of the repair algorithm.

—Pruning due to two occurrences of a state in the same configuration may lead to suboptimal repairs. This case of pruning may lead to failure of the repair algorithm only if repairs are validated.

—Pruning due to two occurrences of a state in distinct search paths, by means of shared variables, may lead to suboptimal repairs or to failure of the repair

algorithm.

4. FURTHER WORK

As reported in McKenzie et al. [1995], the kind of pruning discussed in the previous sections filters out a large number of configurations that are not considered further.

A consequence of this kind of pruning is however that it is very difficult to analyze the behavior of the pruning mechanism for concrete grammars, to predict at compiler-generation time whether it will lead to suboptimal repairs, and possibly to adapt the mechanism so that it dispenses with pruning in cases where otherwise optimal repairs would be lost. We have therefore investigated alternative forms of pruning which are less effective in terms of the decrease in the number of configurations that have to be investigated, but which have useful properties allowing us to perform substantial analysis at compiler-generation time.

One such form of pruning relies on bitmaps that are not shared among different configurations. The bitmaps record the states that occur at certain heights in the parse stack. Second occurrences at the same height or higher lead to pruning. If the parse stack is popped below a certain height, the information about the states that have occurred at that height is discarded.

We have implemented this kind of pruning and formulated a sufficient condition for it to be safe, i.e., to preserve all optimal repairs. We found that in almost all practical cases, pruning was safe. However, because this form is so much weaker than the original one, its effectiveness is very low. This was established by measuring the number of configurations constructed with and without pruning for several hundred randomly introduced grammatical errors in public-domain Java applets [Gosling et al. 1996]. The average reduction of the number of configurations typically amounts to a few percent. It might therefore be stated that this form of pruning, taken for itself, is not really worth the additional effort at generation time.

Another approach is to consider a weaker form of the notion of "safe": we no longer demand that optimal repairs result, but merely that some repair will result if one exists. A sufficient condition for pruning to be safe can be formulated as follows. For each state q and each symbol a we consider the development of the repair algorithm given an error configuration with q on top of the stack and a as next input symbol, once with and once without pruning. (If pruning is not applied, tabulation (cf. Earley [1970]) is needed to ensure termination.) No deletion of a is allowed. We only consider the configurations up to a point where either q is popped off the stack, or a is a valid next input symbol for the current top-of-stack. In the former case we note among other things how many more symbols beside q are popped, and in the latter case we merely note that there was a repair for q and a.

If there are no differences in what we may achieve with and without pruning for such "subcomputations" of the repair algorithm, then this means that pruning in general is safe, i.e., it cannot lead to loss of repairs. This is because any computation of the repair algorithm is composed of a number of such subcomputations. (Note that if the stack is popped to below the original height, then a fresh bitmap is introduced, according to DoReduce in Figure 3.)

This approach will be tested with programming language data in the future.

We expect that blocking of all repairs will happen very seldom, if at all, in practical programming languages. No such case was found in the Java examples. This expectation is further supported by the observation that even cases where pruning leads to more expensive repairs are by no means frequent. Again using the above set of errors in Java source texts, we found that very few errors are repaired in a more expensive way when the pruning facility is turned on. One notable exception was due to the following Java construct which is very similar to our first example of Section 3.

The Java conditional expression syntax

$$\mathit{ConditionalOrExpression}\ ?\ \mathit{Expression}\ :\ \mathit{ConditionalExpression}$$

forces a repetition of states in any repair that inserts all of what follows behind the question mark, because *two* expressions are required. So if the erroneous text is "$a = (b > 0)$? ;" the repair string "*identifier* : *identifier*" cannot be produced because the pruning mechanism prevents it.[3]

The usefulness of the pruning heuristic from McKenzie et al. [1995] appears to be founded on the almost ubiquitous validity of a very simple set of assumptions, which can be expressed as follows: when an error is detected, there is some repair string with the following properties:

(1) No other repair string is less expensive than this one.
(2) It does not require any second occurrences of LR states in the process of its construction.
(3) No other competing repair attempt will encounter one of the states used in this attempt at an earlier point of time.

In spite of our findings concerning the formal defects of the pruning heuristic, it is fair to state that unrestricted pruning is practically useful in a large majority of cases. This study of shortcomings of the pruning mechanism can therefore be concluded by recommending it with only minor reservations.

ACKNOWLEDGMENTS

Frank Gleim and Markus Koetter provided valuable assistance to us by performing experiments with the implementation mentioned in McKenzie et al. [1995]. Ralf Armbruster implemented an earlier version of the generation-time analysis. Three referees and the associate editor were very helpful in finding an appropriate form for this article. Some essential results were in fact stimulated by their critical remarks in an earlier round of reviewing.

REFERENCES

Aho, A., Sethi, R., and Ullman, J. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass.

Earley, J. 1970. An efficient context-free parsing algorithm. *Commun. ACM 13,* 2 (Feb.), 94–102.

Gosling, J., Joy, B., and Steele, G. 1996. *The Java Language Specification.* Addison-Wesley, Reading, Mass.

[3]This does not lead to blocking of all repairs if there is for example an identifier *behind* the semicolon. In that case, deletion of ";" will lead to a repair.

McKenzie, B., Yeatman, C., and De Vere, L. 1995. Error repair in shift-reduce parsers. *ACM Trans. Program. Lang. Syst. 17,* 4 (July), 672–689.

Received March 1996; revised July 1997; accepted December 1997