

Error Repair in Shift-Reduce Parsers

BRUCE J. MCKENZIE, COREY YEATMAN, and LORRAINE DE VERE

University of Canterbury

Local error repair of strings during CFG parsing requires the insertion and deletion of symbols in the region of a syntax error to produce a string that is error free. Rather than precalculating tables at parser generation time to assist in finding such repairs, this article shows how such repairs can be found during shift-reduce parsing by using the parsing tables themselves. This results in a substantial space saving over methods that require precalculated tables. Furthermore, the article shows how the method can be integrated with lookahead to avoid finding repairs that immediately result in further syntax errors. The article presents the results of experiments on a version of the LALR(1)-based parser generator *Bison* to which the algorithm was added.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—*compilers; parsing; translator writing systems and compiler generators*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Bison, error recovery, least cost, shift-reduce, Yacc

1. INTRODUCTION

The problem of correcting and recovering from syntax errors during context-free parsing has received much attention [Feycock and Lazarus 1976; Fischer and Mauney 1980; 1992; Fischer et al. 1980; Grosch 1990; Irons 1963; James 1972; LaFrance 1971; Leinius 1970; Levy 1971; Lyon 1974; Ripley and Druseikis 1978; Röhrich 1980; Sippu and Soisalon-Soininen 1980a; 1980b; 1980c]. Given a context-free grammar (CFG) and an invalid string $s_{err} = t_1 t_2 \dots t_{e-1} t_e t_{e+1} \dots t_n$ we can identify an “error symbol” t_e which is the first symbol at which the error could be detected by a left-to-right scan of the string. This means the substring $t_1 t_2 \dots t_{e-1}$ is the prefix of some valid string $t_1 t_2 \dots t_{e-1} \dots$ of the language while there is no valid string $t_1 t_2 \dots t_e \dots$ that includes the error symbol.

Recovery methods that “repair” the input to construct a string that is valid according to the grammar can be divided into “local” or “global” error repair methods. A local repair only involves changes to the substring $t_e t_{e+1} \dots t_n$ while a global repair allows the possibility that the symbols before the error symbol may also be changed by the repair. Local repairs have less impact on the parser environment as there is no need to revoke earlier parsing decisions such as syntax tree construction during the repair process. The repair method investigated in this article involves only local repair.

Authors’ address: Department of Computer Science, University of Canterbury, Private Bag 4800, Christchurch, New Zealand; email: B.McKenzie@cosc.canterbury.ac.nz.

Permission to make digital/hard copy of all or part of this material without fee is granted provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery, Inc. (ACM). To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee
© 1995 ACM 0164-0925/95/0700-0672 \$03.50

More than a single repair of an invalid string is usually possible, and some way of deciding among these is required. A particular class of repair involves minimal-distance or least-cost recovery [Aho and Peterson 1972; Anderson and Backhouse 1981; Anderson et al. 1983; Backhouse 1981; Dion 1978]. These involve associating an insert/delete cost with each symbol that is inserted in or deleted from the invalid string. The repair with the lowest total cost is then chosen. Such a strategy is applicable to both local and global repair, but the overheads of minimizing the cost globally over the whole invalid string are so high that it has only been applied to local repairs.

Error repair methods associated with parser generation systems require differing levels of interaction by the user of the generator. Some are completely automatic or require the user only to supply insert/delete costs. At the other extreme the user may be required to provide extra information such as the “error productions” available in *Bison* [Corbett and Stallman 1991] and *Yacc* [Johnson 1975]. Unfortunately, providing such productions requires extensive knowledge of shift-reduce parsing and is something of a “black art.”

The approach of Fischer and Mauney and others [Fischer and LeBlanc 1988; Fischer and Mauney 1980; 1992; Fischer et al. 1979a; 1979b; 1980; Mauney and Fischer 1980] which was used in their LL(1)-based FMQ and LALR(1)-based ECP parser generators is closest in spirit to the method proposed in this article. They employ user-supplied symbol insert/delete costs to precalculate for a given CFG the tables $S(A)$ and $E(A, a)$ for each nonterminal A and terminal a . These tables represent the least-cost string of terminals¹ that can be generated from the nonterminal A and the least-cost prefix string² (if any) that can be generated from the nonterminal A that contains the terminal a .

For an LL(1) parser the parsing stack contains a prediction of the remainder of the input as a stack of terminal and nonterminal symbols. For each symbol X on the stack, the entry $E(X, t_{err})$ will give the least-cost insertion (if any) that can be inserted before the error symbol to allow a correct repair. If no such string exists then $S(X)$ can be inserted and the next stack symbol used. After the least-cost insertion (if any) is found that allows the parser to continue with symbol t_{err} , the whole process can be repeated with symbol t_{err+1} , adding the deletion cost of the error symbol to the insertion possible with the error symbol deleted. This process is repeated with subsequent symbols until least-cost recovery is determined. In this way the least-cost combination of inserting and deleting symbols to allow the parse to continue can be found and thus used to repair the error. A similar method is used for LALR(1)-based parsers, but in addition to $S(A)$ and $E(A, a)$, tables giving the predecessor state for each item in a parse state must be stored enabling all possible paths through the parsing DFA to be followed using the state stack of the LALR(1) parser.

The tables that need to be precomputed and stored³ to assist in the repair are quite large, especially for the LALR(1)-based parser generators, owing to the large

¹e.g., $S(A) = x$ such that $A \Rightarrow^* x$ and $\text{Cost}(x)$ is minimized.

²e.g., $E(A, a) = x$ such that $A \Rightarrow xay$ and $\text{Cost}(x)$ is minimized.

³Alternatively $S(A)$ and $E(A, a)$ can be calculated on demand and cached to avoid unnecessary recomputation [Fischer and Mauney 1992].

number of states. The ECP implementation, for example, was forced to hold these on external storage rather than hold them in memory.⁴ A more serious problem is that there is no validation of the error repair. So, although the repair is guaranteed to accept the next input symbol, another error may occur immediately on the following symbol, and potentially an avalanche of errors can result.

The problem of avalanche errors can be lessened by validating the correction. This is known as *regionally least-cost repair* [Mauney 1982], where the cost is minimized within a fixed-sized region of the program. Effectively this requires the repaired string to be correct for at least L tokens after the error. Choosing a region size of say $L = 5$ was found to improve significantly the quality of error repair. An extension of the FMQ algorithm to incorporate such validation is possible [Fischer and Mauney 1992].

This article describes an algorithm for shift-reduce parsers that produces repairs identical to that produced by the Fischer algorithm and allows the repair to be validated using an arbitrary region size of L symbols, but requires no additional tables. The algorithm is presented in the next section with details of the implementation within the LALR(1) parser generator *Bison* [Corbett and Stallman 1991] presented in Section 3. Detailed results and timings are presented in Section 4, and the final section presents our conclusions.

2. NEW ALGORITHM

A shift-reduce parser for a string of symbols from a CFG consists of a deterministic parsing automata (DPA) and a stack, initially empty, of states from the DPA. Variations such as SL(1), LR(1), or LALR(1) result in different DPA for the same grammar, but these variations are not important in our context as the algorithm we present is equally applicable to all of these. The DPA is a function, $\delta(state, symbol)$, consisting of entries:

shift q	shift to state q
reduce $A \rightarrow \alpha$	reduce using a production
accept	accept input and halt
error	error in input

The basic shift-reduce algorithm uses the symbols to make shift and reduce moves based on the DPA until the string is accepted (i.e., it can be generated by the CFG) or an error occurs. Algorithm \mathcal{SR} in Figure 1 expresses this using operations on a configuration $\langle \text{stack} \nabla, \text{remaining input} \vdash \rangle$.⁵ In any practical implementation the remaining input will be implicit, and each successive symbol will be supplied when required by calling a “scanner.” Algorithm \mathcal{SR} is expressed in this somewhat unconventional manner to simplify the explanations of the error recovery algorithm. When an error is encountered the basic algorithm terminates the parse with a suitable error message.

If the CFG is augmented by error productions⁶ then when an error is encountered the stack is investigated for states allowing a shift or (nondefault) reduction on

⁴For example the tables for a Modula-2 grammar require 111KB when stored in their binary form.

⁵ ∇ and \vdash represent the bottom of stack and end of input respectively.

⁶These are productions containing a special “error” symbol such as “**decl**→**error** ;.”

```

config := <  $q_0 \nabla, s \dashv$  >,
loop
  let <  $q_1 q_2 \dots \nabla, t_1 t_2 \dots \dashv$  > = config;
  case  $\delta(q_1, t_1)$  of
    shift  $q_s$ :
      config := <  $q_s q_1 q_2 \dots \nabla, t_2 \dots \dashv$  >;
    reduce  $A \rightarrow \alpha$ :
      let  $l$  = length of  $\alpha$ ;
      let  $q_r = \delta(q_{l+1}, A)$ ;
      config := <  $q_r q_{l+1} q_{l+2} \dots \nabla, t_1 t_2 \dots \dashv$  >;
    accept:
      HALT;                                     -- accept input
    error:
      HALT;                                     -- error in input
  end;
end

```

Fig. 1. Algorithm \mathcal{SR} : Basic shift-reduce parsing.

“error.” The stack is cut back to such a state, the shift or reduction on “error” performed, and then symbols in the input (including the error symbol) skipped until one is found that allows a shift or reduction in this state before the parse is allowed to continue. In practice this is usually augmented to require not just a single symbol, but a number (say three) can be shifted successfully after the recovery. If another error occurs before this number of valid shifts occurs then the stack is again stripped back to an error state and further symbols discarded. This is repeated if necessary.

In this article we propose that the recovery be obtained by searching the DPA itself to find a suitable configuration to allow the parse to continue. For each terminal symbol t from the CFG we assume the existence of positive insert ($I(t)$) and delete ($D(t)$) costs. The recovery technique is presented in algorithm \mathcal{REC} of Figure 2. It maintains a priority queue of configurations augmented with the symbols inserted ($i_1 i_2 \dots$), symbols deleted ($d_1 d_2 \dots$), and the total insert/delete cost ($\sum_j I(i_j) + \sum_k D(d_k)$). The queue is initialized and emptied with `ClearQueue`. Configurations are inserted into the queue with the procedure `Enqueue`, and `DeleteMin` removes the least-cost configuration.⁷

The major features of the algorithm involve beginning with a configuration of the stack and remaining input when the error is detected (cost zero) on the queue and successively investigating “moves” from the least-cost configuration in the queue. This ensures that (a) the “state space” (in the sense of Kreutzer and McKenzie [1991]) is investigated in a “breadth-first” and “least-cost” manner and (b) a least-cost recovery will be obtained.

For each configuration retrieved from the queue without deletions, all possible transitions from the state at the top of the stack are added to the queue for sub-

⁷For small queues a sorted list with the minimum at the front would suffice, while for larger queues a heap or some other priority queue would be more efficient.

```

-- Routines to manipulate a priority queue of augmented configurations
procedure ClearQueue; -- initialize queue and set empty
procedure Enqueue( $\langle \text{stack} \nabla, \text{input} \rceil \rangle$ , inserted, deleted, cost); -- insert new configuration into queue
function DeleteMin() : ( $\langle \text{stack} \nabla, \text{input} \rceil \rangle$ , inserted, deleted, cost); -- remove least-cost configuration from queue

function Recover( $\langle \text{stack}_{err} \nabla, \text{input}_{err} \rceil \rangle$ ) : ( $\langle \text{stack}_{new} \nabla, \text{input}_{new} \rceil \rangle$ ,
begin
  ClearQueue;
  Enqueue( $\langle \text{stack}_{err} \nabla, \text{input}_{err} \rceil \rangle$ ,  $\epsilon, \epsilon, 0$ );
  while queue is not empty do
    let ( $\langle q_1 q_2 \dots q_n \nabla, t_1 t_2 \dots t_m \rceil \rangle$ ,  $\Gamma, \Delta$ , cost) = DeleteMin();
    if  $\Delta = \epsilon$  then
      foreach  $t \in T$  do
        case  $\delta(q_1, t)$  of
          shift  $q_s$ :
            Enqueue( $\langle q_s q_1 q_2 \dots q_n \nabla, t_1 t_2 \dots t_m \rceil \rangle$ ,  $\Gamma, t, \Delta$ , cost + I(t));
          reduce  $A \rightarrow \alpha$ :
            DoReduce( $q_1 q_2 \dots q_n, A \rightarrow \alpha, t, t_1 t_2 \dots t_m, \Gamma, \Delta$ , cost);
          otherwise: ; -- do nothing for error and accept
        end;
      end;
    end;
    Enqueue( $\langle q_1 q_2 \dots q_n \nabla, t_2 \dots t_m \rceil \rangle$ ,  $\Gamma, \Delta, t_1$ , cost + D( $t_1$ ));
    if  $\delta(q_1, t_1) \neq \text{error}$  then
      return  $\langle q_1 q_2 \dots q_n \nabla, t_1 t_2 \dots t_m \rceil \rangle$  -- inserted 'T' and deleted 'Δ' to recover
    end;
  end;
  HALT; -- unable to recover – empty queue
end Recover;

procedure DoReduce( $q_1 q_2 \dots q_n, A \rightarrow \alpha, t, t_1 t_2 \dots t_m, \Gamma, \Delta$ , cost);
begin
  let len = length of  $\alpha$ ;
  let  $q_{new} = \delta(q_{len+1}, A)$ ;
  case  $\delta(q_{new}, t)$  of
    shift  $q_s$ :
      Enqueue( $\langle q_s q_{new} q_{len+1} \dots q_n \nabla, t_1 t_2 \dots t_m \rceil \rangle$ ,  $\Gamma, t, \Delta$ , cost + I(t));
    reduce  $A' \rightarrow \alpha'$ :
      DoReduce( $q_{new} q_{len+1} \dots q_n, A' \rightarrow \alpha', t, t_1 t_2 \dots t_m, \Gamma, \Delta$ , cost);
    otherwise:
      return ; -- error—add nothing to queue
  end;
end DoReduce;

```

Fig. 2. Algorithm \mathcal{REC} : Recovery by “searching” the DPA for least-cost insertions and deletions

sequent investigation. For shifts the new configuration is obtained by pushing the new state onto the stack, adding the symbol to those inserted, and adding the insert cost of the symbols. Reductions are treated by following any sequence of reductions first and then making a final shift before the new configuration is queued again, adding the symbol to those inserted and adding the insert cost.

Finally, a delete configuration is constructed from the least-cost configuration by deleting the first symbol of the remaining input, adding it to those deleted, and accounting for the delete cost of the symbol. A configuration already involving deletions is not investigated for further possible insertions to avoid duplication of effort. For example the sequence “insert i_1 , delete d_1 , and then insert i_2 ” results in exactly the same configuration as the sequence “insert i_1 , insert i_2 , and then delete d_1 .” When there are a number of least-cost recoveries with the same cost, the recovery chosen will vary depending on the order in which `DeleteMin` returns configurations with the same cost. If, for example, it maintained a sorted queue and inserted such configurations *after* configurations of the same cost then the recovery involving fewer deletions would be favored. If, however, the opposite strategy of inserting before configurations with the same cost was chosen, recoveries with more deletions would be favored. In the implementation discussed in the next section the former approach is taken, discriminating against deletions.

After all these new configurations have been queued then the least-cost configuration is checked to see if it is a possible recovery. If it is then we return the resulting configuration as a recovery and allow the parse to continue.

In practice a couple of useful improvements can be made to algorithm *REC*. First, instead of HALTING when the queue is empty it is possible to revert to the error production recovery method.

The second improvement is to augment the algorithm with a validation of repairs similar to that employed by the error production method. If another error occurs too soon after returning from the error recovery procedure then the algorithm is reentered to find the next least-cost recovery. This is repeated until a recovery that allows the parser to continue for at least (say) three symbols is found. This requires that the queue is kept until such a recovery is located.

A small example is useful to show the steps in the recovery of a simple error. Consider the LALR(1) DPA for the simple-expression grammar $E \rightarrow E+E \mid E * E \mid E \wedge E \mid \text{int}$ in Figure 3.⁸ When the input 1+2 3 4 is parsed using the shift-reduce algorithm of Figure 1 the successive configurations are: $\langle 0\nabla, 1 + 234\mid \rangle$, $\langle 10\nabla, +234\mid \rangle$, $\langle 20\nabla, +234\mid \rangle$, $\langle 320\nabla, 234\mid \rangle$, and $\langle 1320\nabla, 34\mid \rangle$ at which point an error is detected. Let us assume that we have insert costs of $I(+) = 1$, $I(*) = 2$, $I(\wedge) = 3$, $I(\text{int}) = 4$, and delete costs equal to the insert costs, ($D(x) = I(x)$), for each symbol. Upon detection of the error an initial configuration is added to the queue by the call

`Enqueue($\langle 1320\nabla, 34\mid \rangle$, ϵ , ϵ , 0).`

When this configuration is removed from the queue it generates four new configurations which will be queued in the order shown:

$\langle 1320\nabla, 34\mid \rangle$, ϵ , ϵ , 0) \rightarrow `Enqueue($\langle 320\nabla, 34\mid \rangle$, “+”, ϵ , 1)`

⁸Precedence and associativity of the operators have been used to resolve shift-reduce and reduce-reduce conflicts arising from the ambiguity of the grammar.

```

→ Enqueue(< 46320∇, 34⊢ >, "*", ε, 2)
→ Enqueue(< 56320∇, 34⊢ >, "^", ε, 3)
→ Enqueue(< 1320∇, 4⊢ >, ε, "3", 4).

```

Removing the least-cost configuration from the queue generates:

```

(< 320∇, 34⊢ >, "+", ε, 1) → Enqueue(< 1320∇, 34⊢ >, "+int", ε, 5)
                             → Enqueue(< 320∇, 4⊢ >, "+", "3", 5).

```

And then because it is possible to shift "3" in state 3 a recovery is attempted. This will result in a further configuration $\langle 1320\bigtriangledown, 4\vdash \rangle$, and another error is detected before three symbols have been successfully shifted. As a result this recovery is abandoned, and the next configuration from the queue is tried instead. Similar unsuccessful recoveries will result from this and the next configuration which will also add configurations with total costs 6 and 7 to the queue. When the next configuration is removed it generates:

```

(< 1320∇, 4⊢ >, ε, "3", 4) → Enqueue(< 1320∇, ⊢ >, ε, "3 4", 8).

```

There are now two configurations with a cost of 5 in the queue, namely, $\langle 1320\bigtriangledown, 34\vdash \rangle$, "+int", ε, 5) and $\langle 320\bigtriangledown, 4\vdash \rangle$, "+", "3", 5). The first of these generates four new configurations:

```

(< 1320∇, 34⊢ >, "+int", ε, 5) → Enqueue(< 320∇, 34⊢ >, "+int+", ε, 6)
                                → Enqueue(< 46320∇, 34⊢ >, "+int*", ε, 7)
                                → Enqueue(< 56320∇, 34⊢ >, "+int^", ε, 8)
                                → Enqueue(< 1320∇, 4⊢ >, "+int", "3", 9).

```

The other configuration with cost 5 generates the single configuration:

```

(< 320∇, 4⊢ >, "+", "3", 5) → Enqueue(< 320∇, ⊢ >, "+", "3 4", 9)

```

and then discovers the possible recovery from state 3 on input symbol "4". This results in successive configurations $\langle 320\bigtriangledown, 4\vdash \rangle$, $\langle 1320\bigtriangledown, \vdash \rangle$, $\langle 6320\bigtriangledown, \vdash \rangle$, $\langle 20\bigtriangledown, \vdash \rangle$, $\langle 920\bigtriangledown, \vdash \rangle$. The final configuration is an ACCEPT configuration, and so a least-cost recovery has been found which involved deleting the "3" and inserting a "+" correcting the input from "1+2 3 4" to "1+2+4."

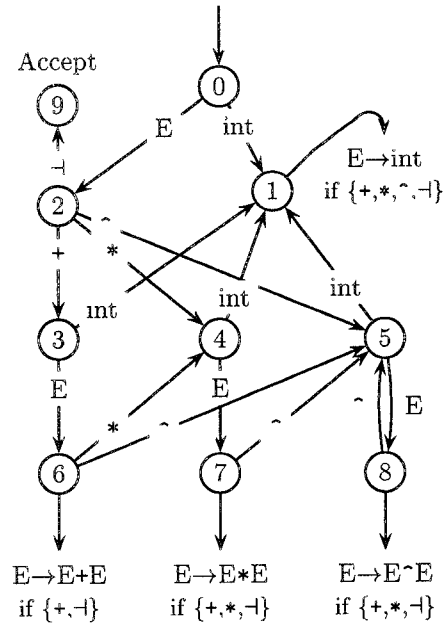
It is not difficult to prove that the algorithm *REC* results in a valid least-cost recovery. The algorithm investigates every possible series of symbols that can be inserted and deleted at the point of the error. The only transitions made are exactly those that would be made if such symbols had actually been encountered in a string. Because the queue returns configurations in order of increasing cost, a least-cost recovery will be found.

It is interesting to note that the algorithm will be able to recover from any possible error. This is because it has the potential to delete every symbol after the error and replace them (insert) with any sequence of symbols. Because the symbols up to the error symbol are a valid prefix of some valid string, then it is always possible to insert symbols that would allow recovery. In spite of this it may be that the computation and space requirements of this are excessive in practice.

3. IMPLEMENTATION

In this section we give details of incorporating the error recovery algorithm into *Bison* [Corbett and Stallman 1991], a widely available LALR(1)-based parser generator, to determine the practicality of the algorithm. This parser generator consists of a program to process the grammar and output the DPA and a fragment

Fig. 3. LALR(1) parsing DFA for Simple-Expression Grammar.



of constant code that implements the shift-reduce and error production parsing algorithms. The scanner is supplied separately by the user. It was possible to experiment with our error recovery method by only changing the constant code section and not making any alteration to the program that generates the DPA. A consequence of this is that it will be a simple matter to update any existing *Bison*-generated system to the new error recovery method.

The description of both the shift-reduce (algorithm \mathcal{SR}) and error recovery (algorithm \mathcal{REC}) algorithms given in the previous section is somewhat idealized. An implementation based on these descriptions would be extremely expensive in both space and time. *Bison*, like many shift-reduce parsers, does not maintain an explicit $\langle \text{stack} \nabla, \text{remaining input} \nabla \rangle$ configuration. Rather the stack is treated as global, and the remaining input is implicit. At any stage only the current input symbol is available, and only after a shift operation is the scanner called to yield the next input symbol. Both of these considerations clearly complicate the implementation of the error recovery algorithm, which assumes such configurations exist and are easily queued and manipulated.

A further complication (which is also applicable to many other shift-reduce systems) is that the DPA function, $\delta(q, a)$, is held in a highly compressed state to save space. A variation of row displacement encoding [Aho et al. 1986, pp.144–146], combined with the use of defaults [Aho et al. 1986, pp.244–247] is used. Although this yields a very significant compaction of the shift-reduce parsing tables without a significant penalty in parse time, it does significantly complicate the extraction of the DPA function. In particular, it is not possible to limit investigation to only “valid” transitions from a state. Rather it is necessary to try all possible values of a to find the valid moves $\delta(q, a)$. Experiments were conducted to determine if

it was worthwhile to extract the complete DPA or at least extract and cache the transitions for a state at a time during recovery. However only a few states were normally needed for most recoveries, and around half the states have only a single default reduction. Furthermore, because decoding the DPA is fast,⁹ it is easier to decode each state as required.

Another consequence of the compaction of the DPA table by means of default reductions is that error detection can be delayed. In particular if the input $t_1 t_2 \dots t_e \dots$ is parsed using a DPA without default reductions the error t_e will be detected in state q where $\delta(q, t_e) = \text{error}$. If, however, there is a default reduction associated with state q then the reduction will be made (and possible further default reductions), and when the error is detected¹⁰ the stack will have been altered. For example consider the parsing DFA for the simple-expression grammar of Figure 3. If this was compacted then all four reductions would become default reductions. In the previous section when the example string $1+2\ 3\ 4$ was parsed an error was detected in the configuration $\langle 1320\nabla, 34\vdash \rangle$. With default reductions the error would not be detected until after the reductions in state 1 (giving $\langle 6320\nabla, 34\vdash \rangle$) and in state 6 (giving $\langle 20\nabla, 34\vdash \rangle$) had been made.

When the error recovery algorithm is started it is important that the stack be the one existing before the default reduction(s) was (were) made. Although for the $1+2\ 3\ 4$ example described above it would make no difference; it would have done so if the insert cost for “+” had been higher than that of “*” rather than the reverse. In such a case the least-cost recovery chosen would have been to delete “3” and insert “*” giving $1+2*4$. However the reductions made would correspond to $(1+2)*4$ rather than the correct $1+(2*4)$. Although this example merely results in a semantically incorrect recovery, in other examples default reductions can mean the least-cost recovery is missed.¹¹ Hence, to allow all possible error recoveries to be considered it is necessary to begin recovery from the configuration that existed before the last chain of default reductions was made.

This requires that the shift-reduce algorithm be augmented to save the state of the stack before each default reduction is made in case an error is detected. This is unfortunate as it will tend to slow the parsing of error-free inputs. Figure 4 shows the time taken to parse Modula-2 programs of various sizes (measured in symbols) with and without saving the stack before default reductions. The trend lines in this figure show that the parsing speed is reduced from 14600 to 10300 tokens/sec.

An alternative to saving the stack before a series of default reductions would be to avoid them altogether by not compressing the parse tables in this way. This has a significant space penalty. For example the parsing LALR(1) DFA tables for Modula-2 have 400 default reduction entries, and if these are removed the 457 action entries (shifts and explicit reductions) increase to 3332. Of more concern,

⁹Decoding a single state requires about 70 microseconds on a Sparc 10 running at 40MHz, averaged over all states in DPAs for the Pascal, Modula-2, and C languages. Decoding states contributes only 1% of the error recovery time.

¹⁰The error will always be detected before the error symbol t_e is shifted.

¹¹For example, consider a small addition to the expression grammar so the grammar rule $E \rightarrow E+E$ became $E \rightarrow E+optE$ with $opt \rightarrow []|\epsilon$. When parsing the input $1+]2$ with default reductions the error is not detected until after the reduction $opt \rightarrow \epsilon$ has already been committed, and so the recovery of inserting $[$ is no longer available.

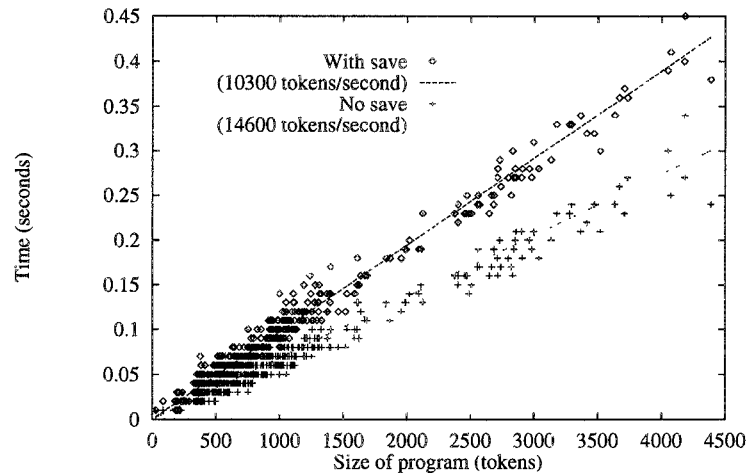


Fig. 4. Time (in seconds) to parse error-free Modula-2 programs of various sizes (in numbers of tokens) both with and without saving the stack before default reductions.

however, is that even without default reductions it is still possible for a reduction to occur because the next input symbol occurs in the lookahead set¹² yet there is no shift possible on this symbol from the state entered after the reduction. This is a consequence of the LALR(1) heuristic of merging “similar” states and combining their lookahead sets. This means it would still be necessary to save the stack before a chain of reductions. Finally the time for error recovery in the absence of default reductions is significantly longer. This is because when a state contains a default reduction the chain of reductions is followed, and then each possible shift from the resulting state is investigated, while without default reduction each symbol in the reductions lookahead set will result in a (usually similar) chain of reductions before making a single shift.

The queue of configurations to investigate can become quite long, so it is important that it is implemented efficiently. As specified by the algorithm *REC* we might expect to store a copy of the stack, the remaining input, along with inserted/deleted symbols and total cost.

Clearly it would be difficult to store the remaining input because it is not available—only the current symbol is available, and then further symbols that the algorithm may consider deleting will be supplied by the scanner. The solution chosen is to store any symbols returned by the scanner during error recovery in an array. Each configuration stores the symbols deleted, and when a further deletion is considered it can be obtained from the array, forcing a further symbol to be returned from the scanner and stored if necessary. When the recovery is complete, and control returns to the shift-reduce parsing algorithm, it will need to reuse any symbols read into this array but not regarded as being deleted by the successful

¹²See Aho et al. [1986, p.230] for an explanation of lookahead sets.

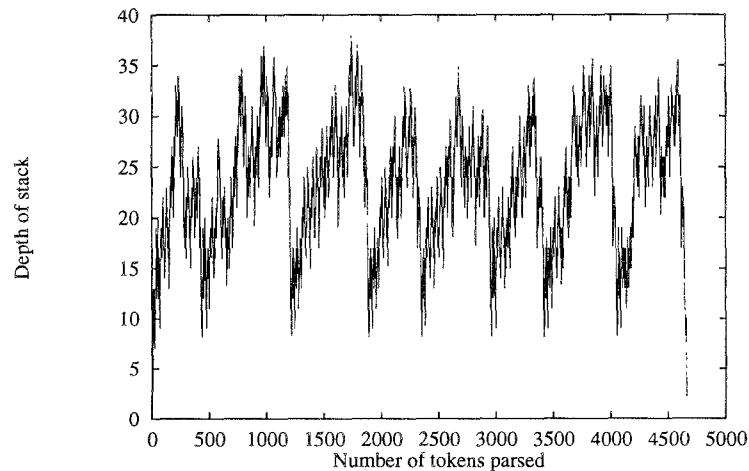


Fig. 5. Size of the parse stack during the parsing of a 216-line, 4659-token Modula-2 program.

Table I. Maximum Size of Stack when Parsing C and Pascal Programs

Language	State stack size		
	Average	Std Dev	Max
Pascal	20.4	9.6	60
C	13.5	10.5	54

recovery.

To assist in deciding how to represent the saved stack in each configuration a number of experiments were conducted to determine the distribution of stack sizes encountered during parsing. Figure 5 shows the size of the stack at each point during the parsing of a 216-line, 4659-token Modula-2 program. For this example the maximum stack size was 38 states with a mean size of 23 ± 6 states.¹³ Table I shows the result of parsing a large number of Pascal and C programs, recording the maximum size of the stack for each. This indicates that on average the state stack is not extremely large, even though there is a reasonable amount of variation from the mean.¹⁴ This suggests that a complex scheme which avoids duplication of the similar stacks is not necessary. Although it has the potential to reduce the space requirements of the algorithm, the added complexity is not justified in the light of the reasonably small stacks that need be saved. The validation of recoveries is achieved by retaining the contents of the priority queue and any symbols used to lookahead when control is returned to the parser. The parser is then allowed to proceed normally, and each successful shift is counted. If another error occurs before

¹³Throughout this article we use the notation 23 ± 6 for a mean of 23 with a standard deviation of 6.

¹⁴*Bison* allocates an initial stack size of 200 configurations.

the required number of validated symbols have been shifted, the error recovery process is restarted with the same priority queue and saved symbols.

A final change was made to the implementation of algorithm \mathcal{REC} to improve its efficiency. This change is not required to guarantee least-cost repairs, but it does result in significantly better repair speeds. During a recovery it is possible for a sequence of insertions to form a cycle within the DPA. Looking again at the expression example of the previous section when the erroneous input `1+2 3 4` was parsed, we see that one of the first few configurations generated by the error recovery was `Enqueue(< 320∇, 34+ >, "+", ε, 1)`. At a later stage the following configuration was queued `Enqueue(< 320∇, 34+ >, "+int+", ε, 6)`. This configuration is identical but of higher cost, and so it (and consequently any of its children) could be pruned from the priority queue. It could never generate a least-cost recovery because if it resulted in a recovery the previous configuration would have yielded one of lower cost. It is also possible that configurations are generated in which the stack is not identical but has exactly the same top state as a previous configuration as a result of investigating a cycle in the DPA. These could also be pruned from the queue because the recovery without the cycle would have a lower cost.

Detection of such unproductive paths was implemented by means of a bitmap indicating which states had been entered. The bitmap is shared by all configurations with the same symbols deleted. The bitmap has a bit for each state in the DPA with all bits initially cleared and a pointer to it associated with the initial configuration. When a new configuration is created by inserting another symbol (via a shift or reduce) the bitmap associated with the configuration is consulted to see if the bit corresponding to the new top of the stack state is set. If it is set then the new configuration is discarded. If it is not set then that bit is set, and when the new configuration is queued a pointer to the same bitmap is associated with it. This changed bitmap will be shared by all those configurations queued with a common ancestor. When a configuration involving a fresh deletion is constructed, a pointer to a fresh bitmap with all bits cleared is associated with the queued configuration. A fresh bitmap is also begun if a reduction reduces the length of the stack below that which it was for the ancestor configuration when the bitmap was created. In these later two situations, reentering the same state again does not necessarily indicate a loop has been detected.

Removing these unproductive configurations from the queue can have a tremendous influence on the efficiency of recovery. Using C with empty input as an example then the least-cost recovery involves inserting `Ident { }`. Given insert costs related to the length of tokens, using the bitmap technique, this recovery is found after investigating 43 and queuing 366 configurations. If, however, bitmap loop detection is not used then the same recovery requires investigating 8586 and queuing 45,276 configurations. The results and timings in the next section are with bitmap loop detection turned on.

There is a subtle¹⁵ interaction between removing loops and validation. For example, a least-cost repair of the C fragment `if c = getchar()) != 0))` (assuming sensible insert/delete costs) with a validation of fewer than seven symbols would insert a single left parenthesis before the `c`, and a further error would be detected at

¹⁵The authors are indebted to one of the reviewers for the following example.

Table II. Statistics of CFG for Programming Languages Used in Tests Giving Number of Terminals (T), Nonterminals (N), and Productions (P)

	T	N	P
Pascal	61	79	177
Modula-2	72	126	248
C	89	102	313

Table III. Statistics of DPAs for Programming Languages Used in Tests Giving Number of States (Q), Shifts (S), Normal Reductions on symbols (R_s), and Default Reductions (R_d)

	Q	S	R_s	R_d
Pascal	323	662	5	195
Modula-2	400	481	72	275
C	517	2398	1	354

the $!=$. If a validation of seven to nine symbols is used the least-cost repair inserts two left parentheses, and the extra closing parenthesis would result in another error and be deleted. Finally if ten or more symbols are used for validation then the least-cost repair of inserting three left parentheses is not chosen as this is eliminated by the bitmap loop detection mechanism.¹⁶ In spite of this somewhat unfortunate interaction between the loop detection heuristic and long validations, the dramatic improvement in the speed of error repair yielded by the heuristic makes it well worth while to add to any implementation of the algorithm. Furthermore, examples such as this are not at all common when a lower validation length is used. In practice a validation of three symbols appears to give good results—significantly reducing the number of repairs that immediately result in a further error. Too high a value runs the risk of combining two distinct errors together.

4. RESULTS AND TIMINGS

In this section we present the results of tests on the algorithm implemented in *Bison* for a number of programming languages, using student programs containing syntax errors. All tests were run on a Sparc 10 running at 40MHz. The programming languages used in the tests were C, Pascal, and Modula-2. Table II gives statistics regarding the CFGs used, and Table III gives statistics regarding the corresponding LALR(1) DPAs produced by *Bison*.

A measure of the complexity of the search is the branching factor, which is the average number of alternatives that need be followed from each state during a recovery. An analytical estimate can be derived from the number of states (Q), shifts (S), normal reductions on symbols (R_s), and default reductions (R_d), and is given by

$$BFA = \frac{S + R_s}{Q} \left(1 + \frac{R_d}{Q} + \frac{R_d^2}{Q^2} + \dots \right) = \frac{S + R_s}{Q - R_d}.$$

An empirical estimate can be obtained by averaging the number of new configurations generated from each state while a collection of sample programs is parsed. If

¹⁶The third but not the second parenthesis is regarded as a loop as the first is associated with the `if` syntax and the next two as parenthesized expressions.

Table IV. Analytical and Empirical (Both with and without Configurations Discarded by the Bitmap Tests) Estimates of the Branching Factors for the Programming Languages Used in the Tests

	BF_A	BF_E	BF'_E
Pascal	5.2	4 ± 2	3 ± 2
Modula-2	4.4	8 ± 3	7 ± 4
C	14.7	20 ± 10	18 ± 11

the new configurations that are discarded by the bitmap test are included, the resulting value (BF_E) should be more comparable with the analytical estimate, while the value excluding these (BF'_E) is a more realistic measure of the complexity. Table IV shows the various branching factors for the languages used in the tests. The large branching factors for the language C are a consequence of (a) the large number of operators available in that language and (b) that most configurations of the language can be expressions.

A collection of 70 Pascal, 135 Modula-2, and five C programs were selected from programs containing syntax errors that students had submitted for compilation. For each program, the algorithm was used to attempt an error recovery, and various properties were measured. For the purpose of these experiments the insert and delete costs were taken to be the length of the symbol. For variable-length tokens such as identifiers and strings, a value that was representative of the average length of such symbols was chosen. Although static costs were used for both insert and delete costs for efficiency, it would be possible to choose a delete cost for, say, an identifier, based on its spelling. Table V shows the statistics for a large number of variables during the compilation of the Modula-2 programs. These generated 273 errors of which 240 could be recovered from and 33 reverted to panic mode recovery after 1000 configurations had been queued without finding a recovery. A validation region of three symbols was used. An analysis of the errors that did not require panic mode processing showed that the majority (63%) produced a least-cost recovery that was successfully validated immediately. However, some errors resulted in recoveries being rejected by validation a number of times before a successful recovery was found. Although the number of rejections before successful validation was typically low (87% of errors required fewer than nine attempts) some required a large number (up to 393 attempts) before a recovery that satisfied the validation criteria was found. The mean number of retries was 10 ± 40 . For the errors that reverted to panic mode recovery this was usually only after a large number of potential recoveries had been rejected by validation. The mean number of retries for this group was 108 ± 45 .

As can be seen the mean recovery time is quite modest (0.02 seconds) which corresponds to parsing around 200-300 tokens without error. For the 12% of errors where the limit of 1000 configurations queued was reached and panic mode entered, the recovery time for these averaged 0.20 ± 0.03 seconds with a maximum time of 0.29 seconds. If this cutoff point is increased to allow more configurations to be queued before reverting to panic mode then reverting to panic mode is less common, but the mean and maximum recovery time increases. There is no upper limit to the recovery time for most realistic grammars. In the worst case it may be necessary to delete all remaining symbols after the error symbol and then insert symbols that

Table V. Statistics Collected During Recovery of 240 Syntax Errors in 135 Modula-2 Programs. These include the number of symbols inserted and deleted in a recovery, the number of states of the DPA expanded, and the number of these that were unique including the percentage of all states ever expanded. Also included is the time to recover in seconds, the number of configurations queued along with the maximum size of the queue during the recovery, and finally information regarding the number of state bitmaps created, the number of bits tested, and those that were found to be already set.

	Symbols		DPA States		Time (sec)	Qued	Max Qlen	Bitmaps		
	Ins	Del	Expand	Visited(%)				Created	Tested	Set
Mean	1.1	0.9	30.0	28.1 (7.0)	0.02	120.5	83.3	52.6	261.7	61.1
Stddev	0.9	4.2	67.8	23.9 (6.0)	0.04	162.2	98.5	137.6	437.1	132.2
Max	6	62	661	125 (31.2)	0.34	961	746	1461	3353	603
Min	0	0	2	1 (0.2)	0.00	1	1	2	5	0

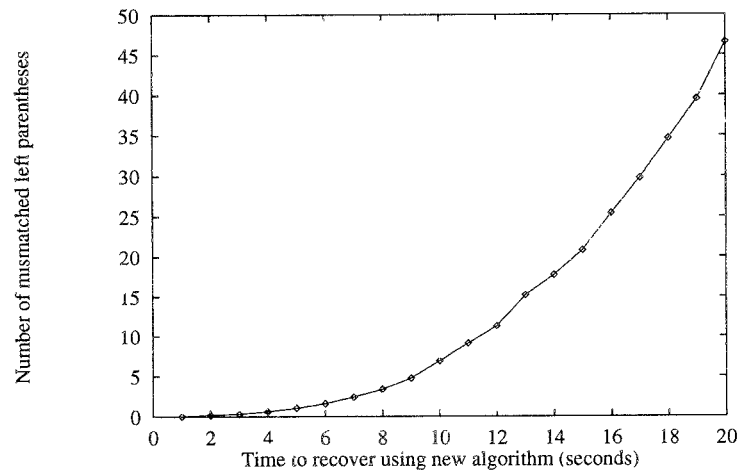


Fig. 6. Time to recover (in seconds) from error in C program fragment `main() { int x; x = (((... (0; } ... (0; }` as number of mismatched left parentheses increases.

complete the input. For example, the C program `main() {int x; x = (((... (0; }` with an arbitrary number of opening parentheses requires an arbitrarily large time to recover as shown by Figure 6.

Comparison with Fischer's ECP system is somewhat complicated by the validation of errors by lookahead. In order to be comparable we need to turn off such validation in our algorithm. When this is done the error repairs are either identical (given the same insert and delete costs), or an alternative recovery with the same cost is chosen. Figure 7 gives a comparison of the time taken for recovery when the same Modula-2 error is corrected by the two algorithms. This figure represents 207 errors in 87 student programs. The average recovery time is 0.027 ± 0.023 seconds for Fischer's ECP program and 0.019 ± 0.026 seconds for the new algorithm. Only when the recovery requires the insertion of a long string of symbols (and hence a long search of the parsing DPA) was the recovery time for Fischer's system signifi-

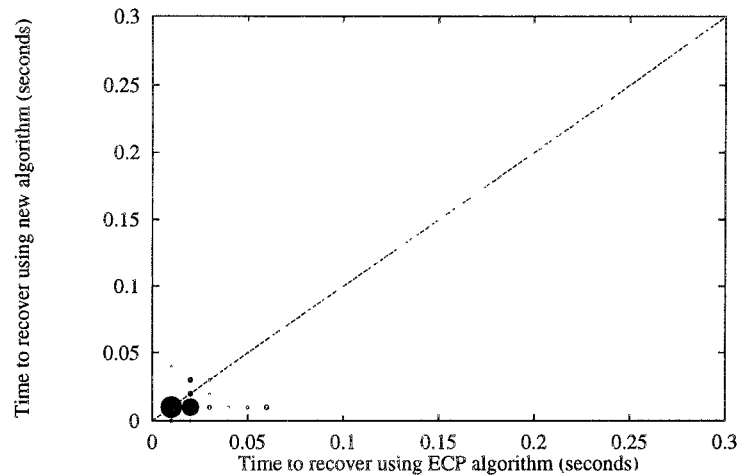


Fig. 7. Comparison of time to recover (in seconds) from the same 207 errors in 87 Modula-2 student programs using Fischer's ECP and the method proposed in this article. The radius of the circle is proportional to the number of programs it represents. The largest represents 50 programs, the next 40, down to the smallest which represent only a single program. Points below the line represent programs for which our new algorithm recovers more quickly than Fischer's algorithm.

cantly lower. The implementation used for ECP was a Pascal program distributed by Mauney and Fischer in 1983 while our *Bison*-based implementation is written in C, so these times should be treated with some care.

5. CONCLUSIONS

A new algorithm for recovering from syntax errors in shift-reduce parsers using the information from the parsing DPA itself to find a least-cost recovery has been presented. The algorithm has been tested on a number of realistic grammars including those for the programming languages C, Modula-2, and Pascal and was found to work well. As the time to find the recovery has no upper bound it is necessary to combine the algorithm with another recovery technique such as panic mode error production recovery in any practical implementation.

A version of a driver for the parser generator system *Bison* has been produced which implements the algorithm and is available via WWW.¹⁷ By default this uses the length of the token as both the insert and delete costs and so can be used as a replacement for the normal standard *Bison* driver program without any other change. Although these default costs result in quite reasonable recoveries it is straightforward for a more sophisticated user to override these defaults with their own costs.

¹⁷URL <http://www.canterbury.ac.nz/~bruce/ErrorRepair.html>

ACKNOWLEDGMENTS

The authors wish to thank Tim Bell and Jane McKenzie for their careful reading of drafts of this article and the anonymous referees for their helpful observations.

REFERENCES

- AHO, A. V. AND PETERSON, T. G. 1972. A minimal distance error correction parser for context free languages. *SIAM J. Comput.* 1, 305–312.
- AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass.
- ANDERSON, S. O. AND BACKHOUSE, R. C. 1981. Locally least-cost error recovery in Earley's algorithm. *ACM Trans. Program. Lang. Syst.* 3, 3, 318–347.
- ANDERSON, S. O., BACKHOUSE, R. C., BUGGE, E. H., AND STIRLING, C. P. 1983. An assessment of locally least-cost error recovery. *Comput. J.* 26, 1, 15–24.
- BACKHOUSE, R. C. 1981. Two global data flow analysis problems arising in locally least-cost error recovery. Tech. Rep. 14, Dept. of Computer Science, Heriot-Watt Univ., Edinburgh, Scotland.
- CORBETT, R. AND STALLMAN, R. 1991. Bison: Gnu parser generator. Texinfo documentation, Free Software Foundation, Cambridge, Mass.
- DION, B. A. 1978. Locally least-cost error correctors for context-free and context-sensitive parsers. Ph.D. thesis, Tech. Rep. 344, Univ. of Wisconsin-Madison, Madison, Wisc.
- FEYCOCK, S. AND LAZARUS, P. 1976. Syntax-directed correction of syntax errors. *Softw. Pract. Exper.* 6, 207–219.
- FISCHER, C. N. AND LEBLANC, R. J. JR. 1988. *Crafting a Compiler*. Benjamin/Cummings, Menlo Park, Calif., sect. 17.2.
- FISCHER, C. N. AND MAUNEY, J. 1980. On the role of error productions in syntactic error correction. *Comput. Lang.* 5, 131–139.
- FISCHER, C. N. AND MAUNEY, J. 1992. A simple, fast, and effective LL(1) error repair algorithm. *Acta Informatica* 29, 109–120.
- FISCHER, C. N., DION, B. A., AND MAUNEY, J. 1979a. A locally least-cost LR-error corrector. Tech. Rep. 363, Univ. of Wisconsin, Madison, Wisc.
- FISCHER, C. N., MILTON, D. R., AND MAUNEY, J. 1979b. A locally least-cost LL(1) error corrector. Tech. Rep. 371, Univ. of Wisconsin, Madison, Wisc.
- FISCHER, C. N., MILTON, D. R., AND QUIRING, S. B. 1980. Efficient LL(1) error correction and recovery using only insertions. *Acta Informatica* 13, 141–154.
- GROSCH, J. 1990. Efficient and comfortable error recovery in recursive descent parsers. *Struct. Program.* 11, 3, 129–140.
- IRONS, E. T. 1963. An error-correcting parse algorithm. *Commun. ACM* 6, 669–673.
- JAMES, L. R. 1972. A syntax directed error recovery method. M.S. thesis, Computer Systems Research Group Tech. Rep. CSRG-13, Univ. of Toronto, Toronto, Canada.
- JOHNSON, S. C. 1975. Yacc: Yet another compiler-compiler. Computer Science Tech. Rep. 32, Bell Laboratories, Murray Hill, N.J.
- KREUTZER, W. AND MCKENZIE, B. J. 1991. *Programming for Artificial Intelligence: Methods Tools and Applications*. International Computer Science Series. Addison-Wesley, Reading, Mass.
- LAFRANCE, J. E. 1971. Syntax directed error recovery for compilers. Ph.D. thesis, Illiac IV Doc. 249, Univ. of Illinois, Urbana-Champaign, Ill.
- LEINIUS, R. P. 1970. Error detection and recovery for syntax directed compiler systems. Ph.D. thesis, Univ. of Wisconsin-Madison, Madison, Wisc.
- LEVY, J. P. 1971. Automatic correction of syntax errors in programming errors in programming languages. Ph.D. thesis, Tech. Rep. TR 71-116, Cornell Univ., Ithaca, N.Y.
- LYON, G. 1974. Syntax-directed least-errors analysis for context-free languages: A practical approach. *Commun. ACM* 17, 3–14.

- MAUNEY, J. 1982. Least-cost error repair using extended right context. Tech. Rep. 495, Univ. of Wisconsin, Madison, Wisc.
- MAUNEY, J. AND FISCHER, C. N. 1980. An improvement to immediate error detection in strong LL(1) parsers. Tech. Rep. 392, Univ. of Wisconsin, Madison, Wisc.
- RIPLEY, G. D. AND DRUSEIKIS, F. C. 1978. A statistical analysis of syntax errors. *Comput. Lang.* 3, 227–240.
- RÖHRICH, J. 1980. Methods for the automatic construction for error correcting parsers. *Acta Informatica* 13, 115–139.
- SIPPU, S. AND SOISALON-SOININEN, E. 1980a. A scheme for LR(k) parsing with error recovery: Part I: LR(k) parsing. *Int. J. Comput. Math.* 8, 27–42, sect. A.
- SIPPU, S. AND SOISALON-SOININEN, E. 1980b. A scheme for LR(k) parsing with error recovery: Part II: Error recovery. *Int. J. Comput. Math.* 8, 107–119, sect. A.
- SIPPU, S. AND SOISALON-SOININEN, E. 1980c. A scheme for LR(k) parsing with error recovery: Part III: Error correction. *Int. J. Comput. Math.* 8, 189–206, sect. A.

Received August 1994; revised February 1995; accepted May 1995