Repairing syntax errors in LR-based parsers

Carl Cerecke
Department of Computer Science
University of Canterbury
Private Bag 4800, Christchurch, New Zealand
Email: cdc@cosc.canterbury.ac.nz

Abstract
When a compiler encounters a syntax error, it usually attempts to restart parsing to check the remainder of the input for any further errors. One common method of recovering from syntax errors is to repair the incorrect input string, allowing parsing to continue. This research presents a language independent method for repairing the input string to an LALR(1) parser. The method results in much faster repairs in general than an existing method, enabling some errors to be repaired that were previously too costly. Results are based on repairing syntax errors in Java programs from first year computer science students.

Keywords: error repair, syntax errors, parsing, LALR(1), bison, yacc

1 Introduction
A common method of error recovery in parsing (Vilares, Darriba & Ribadas 2000, McKenzie, Yeatman & de Vere 1995, Dain 1989, Fischer, Milton & Quiring 1980) is to modify the remaining input to the parser, by inserting and deleting input symbols, in order for parsing to continue. Such a technique is often referred to as error repair.

An error repair algorithm will transform the incorrect input string into a valid input string (or valid prefix of an input string). Parsing is then continued using the transformed input string.

Repair algorithms generally provide a more accurate diagnosis of the repair, and are more likely to continue parsing without spurious errors that result from restarting the parser in a sub-optimal state (Fischer & Manney 1992). Often the repair transformation can be used by the compiler-writer as the basis for a meaningful error message. The focus of this paper is to obtain a good repair transformation.

In a small percentage of errors—those requiring substantial transformation of the input—a repair algorithm will not be able to find a repair in a reasonable length of time. As the number of symbols to be inserted and deleted for a repair increases, the number of possible repairs increases exponentially (true for all but trivial grammars).

This paper presents a method to reduce the exponential explosion in the possible number of repairs by eliminating searches that will never find a 'good' repair.

1.1 Parsing overview
A Context Free Grammar (CFG) is a method of describing the syntax of a language. A CFG $G$ is defined as a four-tuple:

$$G = (N, T, P, S)$$

- $N$ is the finite set of non-terminals. A non-terminal represents a set of sequences of terminals.
- $T$ is the finite set of terminals. Terminals are the basic building blocks of a language. For example in Java, terminals would include class, while, and ;
- $P$ is the finite set of productions. Productions give rules by which non-terminals can be expanded. The left-hand side of the production is a single non-terminal, and the right-hand side is a sequence of terminals and non-terminals. For example, a while statement in a language would typically take this form in a CFG:

$$\text{statement} \rightarrow \text{while expression do statement}$$

where statement and expression are non-terminals and while and do are terminal symbols.
- $S$ is a non-terminal indicating the start symbol. Every valid sentence in the language is represented by this non-terminal.

Parsing is the process of determining whether a string of tokens can be generated by a grammar (Aho, Sethi & Ullman 1986). In this paper we will only be dealing with shift-reduce parsing. Common parser generators (Johnson 1979) (including bison (Donnelly & Stallman 1991), on which this research is based) generate an LALR(1) based shift-reduce parser. LALR(1) parsers are not capable of parsing all constructs expressible with a CFG, but are expressive enough to describe nearly all constructs used in modern computer languages. They are also efficient, able to parse input in linear time.

The parser produced by bison is based on a deterministic parsing automata (DPA). The following Context Free Grammar is a small excerpt of the Assignment construct from a Java grammar in an introductory programming textbook (Lewis & Loftus 1998).

Given this grammar, bison produced the DPA represented in Figure 1.

$$S \rightarrow 'e' A 'e'$$
$$A \rightarrow +=$$
$$A \rightarrow ==$$
$$A \rightarrow =$$
The DPA is a function, $\delta(state, symbol)$, the possible values of which are:

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

The DPA shown in Figure 1 is slightly different than the function shown above, due to the addition of default reductions by *bison* to compress table size and increase parsing speed. The algorithm for parsing (at a basic level) uses the input symbols to make 'shift' or 'reduce' moves based on the DPA until either the string is accepted or an error occurs. The process of parsing a string can be shown using a \(<\text{stack}, \text{remaining input}>\) pair. When a shift is performed, the input symbol is consumed and the new state is pushed onto the stack. When a reduction is performed, the stack is reduced by the number of symbols on the right-hand side of the reduction, and a "goto" move is made to a new state, on the symbol on the left side of the reduction. For example, parsing the input $e = e$ using the DPA in Figure 1 would yield these successive configurations (the left column contains the reductions as they are performed; the right column contains the stack and remaining input. V indicates the bottom of the stack, and $\$\$ the end-of-input symbol.):  

$$
\begin{align*}
&\langle \nabla 0, e = e \$ \rangle \\
&\langle \nabla 0 1, e = e \$ \rangle \\
&\langle \nabla 0 1 4, e \$ \rangle \\
&\langle \nabla 0 1 5, e \$ \rangle \\
&\langle \nabla 0 1 5 6, \$ \rangle \\
&\langle \nabla 0 7, \$ \rangle \\
A &\rightarrow = \\
S &\rightarrow e A e
\end{align*}
$$

The parser would then shift to state 8 on the end-of-input symbol and the string would be accepted.

2 Current Error Repair Techniques

The error handler in a parser has three main goals (Aho et al. 1986):

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs.

Some common methods of error recovery include:

- **panic mode.**
- **error productions.**
- **transformation of input.**

### 2.1 Panic Mode

Panic mode is a simple method of error recovery to implement (Aho et al. 1986). When an error occurs, the parser will skip input symbols until one of a special set of synchronising tokens is found. Synchronising tokens are usually delimiters such as a semicolon, which have a clear meaning in a program. Panic mode may often have to skip over parts of the input, preventing any detection of errors in those parts.

### 2.2 Error productions

There are two forms of error productions. One is to have a special terminal symbol provided by the parser generator called error. The parser will generate this error token whenever a syntax error appears. If there is a rule which recognises this token in the context in which it appears, then the appropriate action can be taken, and the parse resumed if required. The *bison* parser generator supports this.

The other form of error production is when the grammar for a language is altered to recognise common errors, such as an extra semicolon. Fischer and Mauney (Fischer & Mauney 1980) have used this approach on a grammar for Pascal with some success. This approach requires a good knowledge of common syntax errors and context-free grammars.

### 2.3 Correction of input string (repairing)

Correcting the input string attempts to transform (correct) the input into a syntactically correct string by inserting and/or deleting tokens. Methods using this technique of repair can be divided into 'local' and 'global' error repair.

A local repair only involves those tokens after and including the error token, while a global repair will take into account those tokens which appeared before the error token. Global correction is hard to implement in practice, as previous decisions, such as syntax tree constructions, may have to be revoked if changes are required before the occurrence of the error symbol. It is also expensive in terms of resources, taking large amounts of time and space.

#### 2.3.1 Locally least-cost repairs

Fischer and Mauney et al. (Fischer et al. 1980, Fischer & Mauney 1992) used symbol insert/delete costs to precalculate two tables, $S(A)$ and $E(A, a)$ for each non-terminal $A$ and terminal $a$ in a CFG. The former table represented the least-cost string of terminals that can be generated from the terminal $A$, and the latter the least-cost prefix string (if one exists) that can be generated from the non-terminal $A$ that contains the terminal $a$. These tables are used in determining the least-cost recovery.

The size of the tables can be quite large, especially for LALR(1) grammars. Another problem is that there is no validation of the repair. Although the
repair may accept the next input symbol, another error may occur on the following symbol, possibly causing an avalanche of errors.

Validating a correction can dramatically reduce avalanche errors. This is known as regionally (or locally) least-cost repair (Mauney 1982). It requires a repaired string to be correct for at least \( l \) tokens after the error, where \( l \) is the validation length.

### 2.3.2 Work by McKenzie et al.

McKenzie et al. (McKenzie et al. 1995) describe an algorithm that produces repairs identical to those of the Fischer algorithm, but allows the repair to be validated with an arbitrary region size, and requires no tables. Instead of using tables, the method searches the DFA to find a suitable correction which will allow parsing to continue. When an error occurs, the repair algorithm (Figure 2) is started with a configuration containing the stack at the point of error. The configuration is placed in a priority queue ordered by the cost of the insertions and deletions made to the remaining input. A loop is entered that removes the lowest cost configuration from the priority queue and checks to see whether a valid repair is possible using that configuration. If there is no possible repair, new configurations are generated from the current configuration by following the shifts and reductions from the state on top of the current configurations stack.

Each configuration then, is a 4-tuple \( (S, I, D, C) \) where \( S \) is the stack of states, \( I \) is a list of symbols inserted at the point of error, \( D \) is a list of symbols deleted from the remaining input at the point of error, and \( C \) is the cost of the insertions and deletions. While this algorithm can theoretically repair any error it encounters, in practice only errors requiring a change of a few symbols can be performed in reasonably time. Although the majority of errors are 'easy' to repair, and can be repaired by this method, there are some 'difficult' errors that require a change of more than a few symbols to the input to be successfully repaired.

```plaintext
config c
priority queue Q

Initialise new config c with parse stack at point of error Q.insert(c)
while length(Q) > 0: [queue should never be empty]
  c = Q.head()
  if repair-possible(c) == true
    break loop.
  else
    foreach shift s from c:
      create new config c' by shifting s from c
      insert c' into Q
    foreach reduction r from c:
      create new config c' by reducing r from c
      insert c' into Q
Restart parsing with repair-config c
```

Figure 2: Pseudo-code outline for the algorithm described by McKenzie et al.

### 3 A more efficient error repair algorithm

This section describes a new algorithm for error repair that results in much faster repairs for difficult errors than the algorithm shown in Figure 2, at the expense of a slight slowdown for simple errors. The new algorithm (Figure 3) initially does extra work by following non-terminals (gotos) from a configuration in order to eliminate most reductions. For longer repairs, many more reductions may be eliminated than non-terminals followed.

To see the motivation behind following non-terminals, let us first look at a repair using the algorithm from Figure 2. Using the parser in Figure 1 as an example, let us assign the following insert costs:

\[
C_i('+'+) = C_i('-=') = 2 \\
C_i('=') = 1 \\
C_i('e') = 3
\]

For simplicity, we will assign the delete cost of a symbol to be the same as its insert cost. If the erroneous input \( e\$$ was then given to the parser (\$ symbol denotes end of input), an error would occur in state 1 because there is no shift on \( $ \) from that state. From state 1, three possible symbols \( +, -, = \) are considered for insertion. The priority queue of configurations might then look like this:

\[
(V014, =, e, 1) \\
(V012, +=, e, 2) \\
(V013, -=, e, 2)
\]

The searching algorithm then removes the configuration at the head of the priority queue, and checks whether parsing can be restarted using that configuration. In this case, there is only a reduction from state 4. Performing the reduction results in a configuration that has the same cost (no symbols were inserted/deleted) and is placed into the priority queue:

\[
(V015, =, e, 1) \\
(V012, +=, e, 2) \\
(V013, -=, e, 2)
\]

The new configuration is at the head of the priority queue, and is now removed and checked to see whether parsing can be restarted using it. Parsing cannot be restarted, but the \( e \) symbol can be inserted, generating a new configuration \( V0156, =, e, 4 \) that is put onto the priority queue.

The priority queue now looks like this:

\[
(V012, +=, e, 2) \\
(V013, -=, e, 2) \\
(V0156, =, e, 4)
\]

Neither of the two configurations at the head of the priority queue is able to restart the parse, but both can do a reduction, and create a new configuration with a stack of \( V015 \). The resulting configurations can never produce a least-cost repair because a lower cost configuration has already had a stack of \( V015 \) (the stack alone determines the possible strings of tokens that may be parsed). The two higher-cost configurations can be safely pruned.

An alternative (but simpler) way of achieving the same result is to insert non-terminals but not do any reductions. As well as the three shifts from state 1, we would also create a configuration for the goto: \( V015, A, e, 1 \). The cost of inserting a non-terminal is the least-cost insert string that can be generated by that non-terminal (the string = for \( A \)). Now, although the other three configurations will still be tested to see whether parsing can be restarted, the configurations resulting from the reductions can be safely pruned. This is the method the new algorithm uses.

Not all reductions are pruned. The exception is when a reduction would take the stack back to a point
config c
priority queue Q

Initialise new config c with parse stack at point of error Q.insert(c)
while length(Q) > 0: [queue should never be empty]
    c = Q.head()
    if repair-possible(c) == true
        break loop.
else
    foreach shift s from c:
        create new config c' by shifting s from c Q.insert(c')
    foreach goto g from c:
        create new config c' by goto-ing on g from c Q.insert(c')
    foreach reduction r from c:
        n = length(c.stack)−length(c.start-stack)
        if |RHS(r)| > n
            create new config c' by reducing r from c c'.start-stack = c.stack Q.insert(c')

Restart parsing with repair-config c

Figure 3: Pseudo-code outline for the new error repair algorithm.

that no previous configuration has yet searched. For
the example above, a configuration with a stack of
\textbackslash{\textit{V0156}} would still do the reduction back to state 0
and forward to state 7 because state 0 had not yet
been searched (searching for a repair began at state
1, the error state).

Figure 3 shows the pseudo code for the new algo-

rithm.

4 Experimental Results

While there have been many algorithms in the area
of error recovery, there is rarely any experimental
evidence provided using real programs. The no-
table exception is the now dated Pascal suite of pro-
grams analysed by Ripley and Druseikis (Ripley &
Druseikis 1978). This suite comprises 126 Pascal pro-
grams written by graduate students, with a total of
414 errors. Of these errors, 88% were single-token
errors (requiring a single insert, delete, or replace).

This paper presents results using 59,643 syntacti-
cally incorrect Java programs. These were extracted
from 196,860 programs submitted to the Java com-
piler by first year computer science students 1998–
2000 at the University of Canterbury. The 59,643
programs were chosen by first removing those pro-
grams with semantic errors only, accounting for 36%
of the incorrect programs. Furthermore, two differ-
ent Java grammars\footnote{1} had to agree on the same first
error token for a program to make the final set. This
reduced the remaining total by about half.

Only the first error in each program was considered
because its position can be guaranteed. The position
of a subsequent error might depend on the result of
parsing decisions made while repairing the first error.
Although the two authors claim that their gram-
mars are for \textit{Java}, both grammars recognise a (dif-
ferent) superset of the Java language. The results in
this paper are based on the grammar by Bäckerud.

config grammars\footnote{1}\footnote{1}.

A comparison of the effectiveness of each grammar in

the area of syntax error repair is outside the scope of
this paper.

Duplicates were left in the set of programs, as were
different programs with identical errors, and files that
were submitted to the compiler but were clearly not
Java programs.

The repair algorithm of McKenzie et al. from Sec-
tion 2.3.2 and the new algorithm presented in Figure 3
were added to \textit{bison} (Donnelly & Stallman 1991), a
widely used existing parser generator. To be success-
ful, a repair had to be able to parse the next three
tokens in the input (validation length of 3).

A repair was deemed unsuccessful (somewhat arbi-
trarily) if 1,000,000 configurations were added to the
priority queue with no repair found. An unsuccessful
repair takes approximately two seconds on an Athlon
800Mhz computer running Linux (The number of con-
figurations queued is approximately proportional to
the time taken). Requiring more than two seconds for
a repair would be unacceptable in most programming
environments. The software implementing the repairs
was written primarily with correctness and flexibility
for pruning mechanisms in mind, rather than being
optimised for speed. It is estimated that a production
version could easily increase the speed by a factor of
three or more.

Repairs were only attempted on the first syntax
error that occurred in each Java program.

Figure 4 shows the number of configurations
queued for each syntax error using both methods.
The vertical line of points along the right-hand side of
the graph show repairs that were unsuccessful using
the default search, but were able to be repaired when
non-terminals were followed.

The graph shows two major trends: the higher
trend is where following non-terminals has reduced
the number of configurations taken to find a repair
by a factor of about 2.5; the lower trend is where
following non-terminals has reduced the number of
configurations by a factor of about 5–7.

Although there is a clear distinction between the
trends, no obvious distinction was found in the errors
that make up each of these trends.

Interestingly, of the 59,643 error repairs, 53.8%
(32,080) of the repairs took longer when configura-
tions were pruned by following non-terminals, and
22.9% (13,660) of the repairs showed no change in
the number of configurations queued. Only 23.3%
(13,904) of the repairs showed an improvement when
following non-terminals. In nearly all of the repairs
where there was no improvement, the repairs were
extremely quick, with fewer than 5,000 configurations
queued. Even the most extreme of the slower re-
pairs took fewer than 50,000 configurations, and was
about 1.7 times slower than the default algorithm.
At approximately 0.1s, this is still a quick repair.

The reason some repairs take longer when following
non-terminals is that this involves some extra work
(queuing configurations by following non-terminals)
to avoid even more work (following unnecessary re-
ductions) at a later stage. For a number of shorter
repairs, the benefit of eliminating reductions is out-
weighed by the cost of following non-terminals.

Using the default searching algorithm, 1,213
(2.0%) errors were not able to be repaired within
the one million configuration limit. Enabling the fol-
lowing of non-terminals saw the unrepairable errors
drop to 964 (1.6%). Despite many of the longer re-
pairs showing a dramatic speed-up by following non-
terminals, there is still a core of difficult errors that
are resistant to a quick repair.

The errors unable to be repaired were usually
found to belong to one of the following categories:

- Truncated methods: Students tried to compile a
program where a partially finished method appears in the middle or at the end of the file.

- Non-Java errors: Students appear to have non-Java text in (or as) their program. For example, shell output is accidentally pasted into a program or the wrong language is used. In one program, a for loop appears as For Outer:=1 To 3 Do.

- Multiple errors: The program has more than one error in very close proximity, for example public void foo(int 1, int 2, int 3) has three errors in close proximity. This example would be much more easily repaired with a validation length of two.

- Incorrect delimiter use: Strings, comments, blocks, methods, or array initialisers have incorrect, missing, or mismatched delimiters.

The largest category is ‘incorrect delimiter use’. Although many simpler delimiter errors were able to be repaired, a number of the more difficult errors (for example, a long comment beginning with */ instead of /*) in this category could not be repaired in time. A possible solution that requires further investigation would be to discover the effect of reducing the insert and delete costs of tokens that are typically involved in delimiter errors.

5 Conclusion

Following non-terminals (and limiting reductions) when searching for a repair is effective in reducing the time taken to find a successful repair for many errors. Furthermore, there are a number of errors that take too long to repair unless non-terminals are followed in the repair search process. Nevertheless, there is a small number of errors that are still difficult to repair. Other improvements to the algorithm and refinements to the assignment of insert/delete costs may reduce the number of difficult repairs further. However, it is unlikely to ever transform a hybrid Java/Visual Basic ‘program’ into syntactically correct Java.

References


