University of Canterbury
Private Bag 4800, Christchurch, New Zealand
Email: cdc@cosc.canterbury.ac.nz

tract

n a compiler encounters a syntax error, it usually attempts start parsing to check the remainder of the input for any er errors. One common method of recovering from syntax s is to repair the incorrect input string, allowing parsing ntinue. This research presents a language independent od for repairing the input string to an LALR(1) parser. method results in much faster repairs in general than an ng method, enabling some errors to be repaired that were ously too costly. Results are based on repairing syntax s in Java programs from first year computer science stu- .

words: error repair, syntax errors, parsing, R(1), bison, yacc

Introduction

ommon method of error recovery in parsing ares, Darriba & Ribadas 2000, McKenzie, Yeat- & de Vere 1995, Dain 1989, Fischer, Milton & ing 1980) is to modify the remaining input to the er, by inserting and deleting input symbols, in or- or parsing to continue. Such a technique is often red to as *error repair*.

n error repair algorithm will transform the in- ect input string into a valid input string (or valid x of an input string). Parsing is then continued g the transformed input string.

epair algorithms generally provide a more ac- te diagnosis of the repair, and are more likely ontinue parsing without spurious errors that re- from restarting the parser in a sub-optimal state her & Mauney 1992). Often the repair trans- ation can be used by the compiler-writer as the for a meaningful error message. The focus of paper is to obtain a good repair transformation. a small percentage of errors— those requiring tantial transformation of the input— a repair al- hm will not be able to find a repair in a reason- length of time. As the number of symbols to be ted and deleted for a repair increases, the num- of possible repairs increases exponentially (true ll but trivial grammars).

his paper presents a method to reduce the ex- ential explosion in the possible number of repairs liminating searches that will never find a 'good' ir.

Parsing overview

ontext Free Grammar (CFG) is a method of de- ing 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 no terminal represents a set of sequences of term nals.
- T is the finite set of *terminals*. Terminals are th basic building blocks of a language. For examp in Java, terminals would include `class`, `whil` and `;`.
- P is the finite set of productions. Productio give rules by which non-terminals can be e panded. The left-hand side of the production a single non-terminal, and the right-hand side a sequence of terminals and non-terminals. F example, a **while** statement in a language wou typically take this form in a CFG:

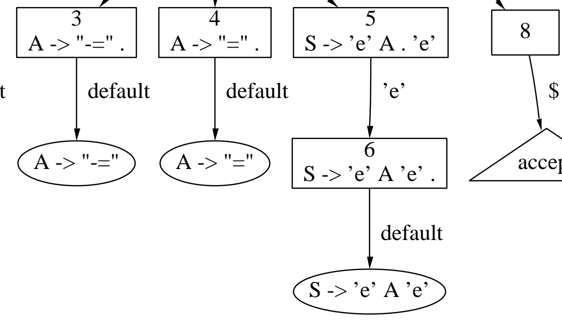
  statement → `while` *expression* `do` *statement*

  where *statement* and *expression* are no terminals and `while` and `do` are terminal sy bols.
- S is a non-terminal indicating the start symb Every valid sentence in the language is repr sented by this non-terminal.

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

The parser produced by *bison* is based on a dete ministic parsing automata (DPA). The following Co text Free Grammar is a small excerpt of the *Assig ment* construct from a Java grammar in an introdu tory programming textbook (Lewis & Loftus 199 Only three of the twelve assignment operators a shown, and the expressions on both sides of the a signment are condensed into the terminal '`e`'. Give this grammar, *bison* produced the DPA represent in Figure 1.

right ©2001, Australian Computer Society, Inc. This pa



re 1: A DPA produced by the *bison* parser gen-
or

he DPA is a function, $\delta(state,symbol)$, the possi-
alues 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.

he DPA shown in Figure 1 is slightly different
the function shown above, due to the addition
efault reductions by *bison* to compress table size
increase parsing speed. The algorithm for pars-
at a basic level) uses the input symbols to make
t' or 'reduce' moves based on the DPA until ei-
the string is accepted or an error occurs. The
ess of parsing a string can be shown using a
ck,remaining input> pair. When a shift is per-
ed, the input symbol is consumed and the new
e is pushed onto the stack. When a reduction
erformed, the stack is reduced by the number of
bols on the right-hand side of the reduction, and
oto" move is made to a new state, on the symbol
he left side of the reduction. For example, parsing
nput `e=e` using the DPA in Figure 1 would yield
e successive configurations (the left column con-
s the reductions as they are performed; the right
mn contains the stack and remaining input. ∇
cates the bottom of the stack, and `$` the end-of-
t symbol.):

	<∇0 , e=e$>
	<∇01 , =e$>
	<∇014 , e)$>
=	<∇015 , e$>
	<∇0156 , $>
eAe	<∇07 , $>

he parser would then shift to state 8 on the end-
put symbol and the string would be accepted.

Current Error Repair Techniques

- error productions.
- transformation of input.

2.1 Panic Mode

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

2.2 Error productions

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

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

2.3 Correction of input string (repairing)

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

A local repair only involves those tokens after an
including the error token, while a global repair wi
take into account those tokens which appeared befor
the error token. Global correction is hard to imple
ment in practice, as previous decisions, such as synta
tree constructions, may have to be revoked if change
are required before the occurrence of the error sym
bol. It is also expensive in terms of resources, takin
large amounts of time and space.

2.3.1 Locally least-cost repairs

Fischer and Mauney et al. (Fischer et al. 1980, Fi
cher & Mauney 1992) used symbol insert/delete cost
to precalculate two tables, $S(A)$ and $E(A,a)$ for eac
non-terminal A and terminal a in a CFG. The fo
mer table represented the least-cost string of term
nals that can be generated from the terminal A, an
the latter the least-cost prefix string (if one exist
that can be generated from the non-terminal A tha
contains the terminal a. These tables are used in d

2 Work by McKenzie et al.

[McK]enzie et al. (McKenzie et al. 1995) describe an [algo]rithm that produces repairs identical to those of [the] Fischer algorithm, but allows the repair to be val[idat]ed with an arbitrary region size, and requires no [tabl]es. Instead of using tables, the method searches [the] DPA to find a suitable correction which will al[low] parsing to continue. When an error occurs, the [repa]ir algorithm (Figure 2) is started with a configu[rati]on containing the stack at the point of error. The [conf]iguration is placed in a priority queue ordered by [the] cost of the insertions and deletions made to the [rema]ining input. A loop is entered that removes the [lea]st cost configuration from the priority queue and [chec]ks to see whether a valid repair is possible using [that] configuration. If there is no possible repair, new [conf]igurations are generated from the current config[urati]on by following the shifts and reductions from [the s]tate on top of the current configurations stack.

[E]ach *configuration* then, is a 4-tuple (S, I, D, C) [whe]re S is the stack of states, I is a list of symbols [inser]ted at the point of error, D is a list of symbols [delet]ed from the remaining input at the point of error, [and] C is the cost of the insertions and deletions.

[W]hile this algorithm can theoretically repair any [erro]r it encounters, in practice only errors requiring [a ch]ange of a few symbols can be performed in rea[sona]bly time. Although the majority of errors are [‘easy]’ to repair, and can be repaired by this method, [ther]e are some ‘difficult’ errors that require a change [of m]ore than a few symbols to the input to be suc[cessf]ully repaired.

```
[confi]g c
[prior]ity queue Q

[initia]lise new config c with parse stack at point of error
[Q.in]sert(c)
[whil]e 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
[rest]art parsing with repair-config c
```

[Figu]re 2: Pseudo-code outline for the algorithm de[scrib]ed by McKenzie et al.

[3] A more efficient error repair algorithm

[As] an example, let us assign the following insert cost[s]

$$C_i(\text{`+='}) = C_i(\text{`-='}) = 2$$
$$C_i(\text{`='}) = 1$$
$$C_i(\text{`e'}) = 3$$

For simplicity, we will assign the delete cost of [a] symbol to be the same as its insert cost. If the erro[r]neous input `e$` was then given to the parser (`$` symbo[l] denotes end of input), an error would occur in state [1] because there is no shift on `$` from that state. Fro[m] state 1, three possible symbols (`+=`, `-=`, and `=`) ar[e] considered for insertion. The priority queue of con[-]figurations might then look like this:

$(\nabla 014 , \texttt{=} , \epsilon , 1)$
$(\nabla 012 , \texttt{+=} , \epsilon , 2)$
$(\nabla 013 , \texttt{-=} , \epsilon , 2)$

The searching algorithm then removes the config[-]uration at the head of the priority queue, and check[s] whether parsing can be restarted using that config[-]uration. In this case, there is only a reduction fro[m] state 4. Performing the reduction results in a config[-]uration that has the same cost (no symbols were in[-]serted/deleted) and is placed into the priority queu[e]

$(\nabla 015 , \texttt{=} , \epsilon , 1)$
$(\nabla 012 , \texttt{+=} , \epsilon , 2)$
$(\nabla 013 , \texttt{-=} , \epsilon , 2)$

The new configuration is at the head of the pr[i]ority queue, and is now removed and checked to se[e] whether parsing can be restarted using it. Parsin[g] cannot be restarted, but the `e` symbol can be inserte[d] generating a new configuration $(\nabla 0156 , \texttt{= e} , \epsilon , 4)$ that is put onto the priority queue.

The priority queue now looks like this:

$(\nabla 012 , \texttt{+=} , \epsilon, 2)$
$(\nabla 013 , \texttt{-=} , \epsilon, 2)$
$(\nabla 0156 , \texttt{= e} , \epsilon, 4)$

Neither of the two configurations at the head o[f] the priority queue is able to restart the parse, bu[t] both can do a reduction, and create a new configu[-]ration with a stack of $\nabla 015$. The resulting configu[-]rations can never produce a least-cost repair becaus[e] a lower cost configuration has already had a stack o[f] $\nabla 015$ (the stack alone determines the possible string[s] of tokens that may be parsed). The two higher-co[st] configurations can be safely pruned.

An alternative (but simpler) way of achieving th[e] same result is to insert non-terminals but not do an[y] reductions. As well as the three shifts from state 1, w[e] would also create a configuration for the goto: $(\nabla 01[5]$ $, \texttt{A} , \epsilon , 1)$. The cost of inserting a non-terminal is th[e] least-cost insert string that can be generated by tha[t] non-terminal (the string `=` for `A`). Now, although th[e]

```
      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')
art parsing with repair-config c
```

re 3: Pseudo-code outline for the new error repair rithm.

no previous configuration has yet searched. For example above, a configuration with a stack of 56 would still do the reduction back to state 0 forward to state 7 because state 0 had not yet searched (searching for a repair began at state e error state).

igure 3 shows the pseudo code for the new algo- n.

Experimental Results

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

his paper presents results using 59,643 syntacti- incorrect Java programs. These were extracted 196,860 programs submitted to the Java com- by first year computer science students 1998– at the University of Canterbury. The 59,643 rams were chosen by first removing those pro- ns with semantic errors only, accounting for 36% he incorrect programs. Furthermore, two differ- Java grammars[1] had to agree on the same first token for a program to make the final set. This ced the remaining total by about half.

nly the first error in each program was considered use its position can be guaranteed. The position subsequent error might depend on the result of ing decisions made while repairing the first error. lthough the two authors claim that their gram- s are for 'Java', both grammars recognise a (dif- t) superset of the Java language. The results in

widely used existing parser generator. To be succes ful, a repair had to be able to parse the next thre tokens in the input (validation length of 3).

A repair was deemed unsuccessful (somewhat arb trarily) if 1,000,000 configurations were added to th priority queue with no repair found. An unsuccessf repair takes approximately two seconds on an Athlo 800Mhz computer running Linux (The number of co figurations queued is approximately proportional t the time taken). Requiring more than two seconds fo a repair would be unacceptable in most programmi environments. The software implementing the repai was written primarily with correctness and flexibilit for pruning mechanisms in mind, rather than bein optimised for speed. It is estimated that a productio version could easily increase the speed by a factor three or more.

Repairs were only attempted on the first synta error that occured in each Java program.

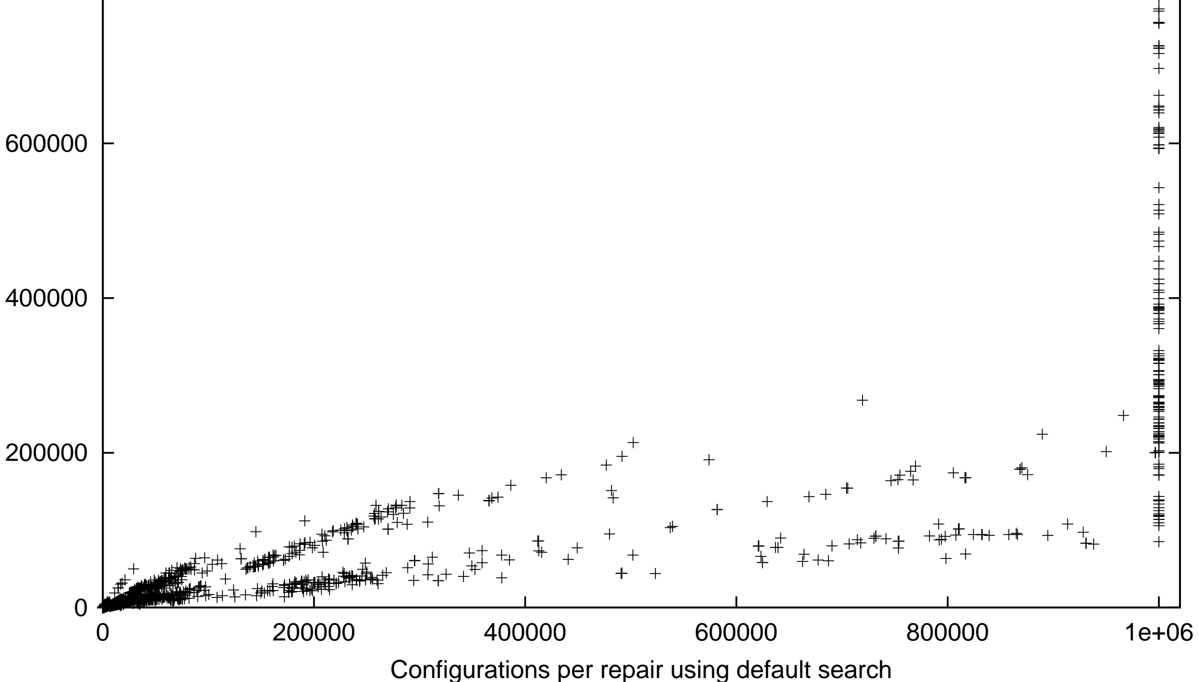
Figure 4 shows the number of configuratio queued for each syntax error using both method The vertical line of points along the right-hand side the graph show repairs that were unsuccessful usi the default search, but were able to be repaired whe non-terminals were followed.

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

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

Interestingly, of the 59,643 error repairs, 53.8 (32,080) of the repairs took *longer* when configur tions were pruned by following non-terminals, an 22.9% (13,660) of the repairs showed no change i the number of configurations queued. Only 23.3 (13,904) of the repairs showed an improvement whe following non-terminals. In nearly all of the repai where there was no improvement, the repairs were e tremely quick, with fewer than 5,000 configuratio queued. Even the most extreme of the slower r pairs took fewer than 50,000 configurations, and wa about 1.7 times slower than the default algorith At approximately 0.1s, this is still a quick repai The reason some repairs take longer when followi non-terminals is that this involves some extra wo (queueing configurations by following non-terminal to avoid even more work (following unnecessary r ductions) at a later stage. For a number of short repairs, the benefit of eliminating reductions is ou weighed by the cost of following non-terminals.

Using the default searching algorithm, 1,21 (2.0%) errors were not able to be repaired withi the one million configuration limit. Enabling the fo lowing of non-terminals saw the unrepairable erro drop to 964 (1.6)%. Despite many of the longer r pairs showing a dramatic speed-up by following no



re 4: A comparison of the difficulty of the repair, with and without following non-terminals, for eac
ax error.

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.

he largest category is 'incorrect delimiter use'. ough many simpler delimiter errors were able to epaired, a number of the more difficult errors (for nple, a long comment beginning with `*/` instead `*`) in this category could not be repaired in time. ossible solution that requires further investigation d be to discover the effect of reducing the insert delete costs of tokens that are typically involved elimiter errors.

too long to repair unless non-terminals are followe in the repair search process. Nevertheless, there is small number of errors that are still difficult to repai Other improvements to the algorithm and refinemen to the assignment of insert/delete costs may reduc the number of difficult repairs further. However, it unlikely to ever transform a hybrid Java/Visual Bas 'program' into syntactically correct Java.

References

Aho, A. V., Sethi, R. & Ullman, J. D. (1986), *Compi ers, Principles, Techniques, and Tools*, Addiso Wesley.

Dain, J. A. (1989), Automatic Error Recovery for L Parsers in Theory and Practice, PhD thesis, Un versity of Warwick.

Donnelly, C. & Stallman, R. M. (1991), *Biso Gnu parser generator*, Free Software Foundatio Cambridge, Mass.

Fischer, C. N. & Mauney, J. (1980), 'On the role error productions in syntactic error correctio *Computer Languages* **5**, 131–139.

Fischer, C. N. & Mauney, J. (1992), 'A simple, fas and effective LL(1) error repair algorithm', *Ac Informatica* pp. 109–120.

ney, J. (1982), Least-cost error repair using extended right context, Technical Report TR-495, University of Wisconsin.

lenzie, B. J., Yeatman, C. & de Vere, L. (1995), 'Error repair in shift-reduce parsers', *ACM TOPLAS* **17**(4), 672–689.

ey, G. D. & Druseikis, F. C. (1978), 'A statistical analysis of syntax errors', *Computer Languages* **3**, 227–240.

res, M., Darriba, V. & Ribadas, F. (2000), Regional least-cost error repair, *in* 'International Conference on Implementation and Application of Automata'.