Robust error correction in a generalized LR parser

STEPHEN LEAKE, retired

Error correction in a parser is important when the parser is used in an interactive development environment (IDE); the source code is often not syntactically correct. Several extensions to work by McKenzie, Yeatman, and De Vere [15] are described, including 'Minimal_Complete', which quickly provides the missing end of any grammar production. The algorithm has been in production use in Emacs ada-mode for over a year; metrics show that the algorithm is useful.[1]

CCS Concepts: • **Software and its engineering → Parsers**; **Translator writing systems and compiler generators**; • **Human-centered computing → User interface programming**; • **Computing methodologies** → *Shared memory algorithms.*

ACM Reference Format:
Stephen Leake. 2020. Robust error correction in a generalized LR parser. 1, 1 (July 2020), 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Emacs Ada mode has used an LR parser to support indentation, syntax highlighting, and navigation since 2013 [13]. However, the parser did not provide error correction, so indentation was often confusing when the syntax was incorrect, as it usually is in an interactive editing environment. This motivated the search for error correction algorithms.

Grune and Jacobs [11] provides a thorough overview of error correction algorithms in LR and LL parsers. Of those, the work by McKenzie, Yeatman, and De Vere [15] provides the foundation for the current work.

The McKenzie algorithm works by exploring the parse table (or Deterministic Parsing Automata (DPA) as Mckenzie et al. [15] calls it) at the error point, finding tokens to insert. It also tries deleting tokens following the error point. Each possible solution, together with the parse stack at the error point, forms a *configuration*. Each configuration also has a cost, determined by what tokens are inserted and deleted. At each step in the algorithm, new configurations are generated from the current error point. Then the minimum cost configuration is checked to see if it succeeds; if not, more configurations are generated.

There are several situations where the McKenzie algorithm can take a long time, or is inefficient. For example, consider this Ada code:

```
procedure Example_01
is begin
   Msg : constant String;
begin
   Put_Line (Msg);
end;
```

[1]This work was supported in part by Eurocontrol

Author's address: Stephen Leake, stephen_leake@stephe-leake.orgretired.

There is an extra '**begin**' immediately after '**is**' (a common occurance while editing code). However, the error is not detected until ':' after 'Msg', which can only occur in declarations, not statements (in Ada, declarations occur between '**in**' and '**begin**'; statements between '**begin**' and '**end**').

To fix this, the McKenzie algorithm must insert
'; **end**; **begin** IDENTIFIER', or delete '**constant** String ; **begin**' and then insert ';'. A much better solution would be to "push back" '**begin** Msg', and then delete '**begin**'.

A harder problem is when there are several missing "end"s, because the user is typing a nested statement:

```
begin
   if A then
      B;
      if C then
         loop
            Do_Something;
end;
```

Fig. 1. missing many 'end's

Here we are missing several tokens: '**end loop**; **end if** ; **end if** ;'. McKenzie will theoretically find the solution that inserts all of these, but along the way it will try inserting every possible statement as well, wasting a lot of time, and in practice hitting a time-out limit. This article introduces the 'Minimal_Complete' algorithm, which quickly finds the minimal number of tokens to insert to complete the trailing statement in situations like this.

This article also introduces several other new operations for the core McKenzie algorithm to try at the error point, and adapts the algorithm to work with a generalized parser.

Unless quoting from another source, we use the notation described by DeRemer and Pennello [6] or Aho, Sethi, and Ullman [3].

2 THE PARSING CONTEXT

2.1 Generalized parsing

We use a generalized LR parser Tomita [16], to tolerate conflicts in the parse table. This allows using an Ada grammar that is close to the one given in the ISO Ada language standard appendix P [9], which is not LR(1). Using that grammar as faithfully as possible ensures we are implementing the correct language, and simplifies updating the parser to a new language version.

This means the main parser may have several parallel parsers executing when an error is encountered. The McKenzie error correction algorithm is enhanced to maintain state information for each parser. When more than one solution is found with the same cost, additional parsers are created to use them.

The parser builds a syntax tree; to handle parallel parsers, we use a branched tree similar to Tomita's sub-tree sharing. The syntax tree will contain "virtual tokens" that are inserted by error correction.

The entire input token sequence is kept in memory, to allow arbitrary push back.

2.2 Partial parse

In order to handle very large files, the parser supports "partial parse"; parsing only part of a file.

Note that this is not the same as "incremental parse", where an existing syntax tree from a previous parse is modified based on a text edit. Integrating this error correction algorithm with incremental parse is the subject of future work.

In order to minimize the amount of text passed from the editor to the parser, we modify the grammar to allow smaller chunks of code to be accepted as a complete parse. In Ada, this means adding ' declaration ' and 'statement' to 'compilation_unit'.

When the file length is greater than a threshold, Emacs invokes a partial parse whenever a parse is needed. It first uses a regular expression search to find a reasonable start point, then finds a possible matching end that includes the requested parse position, and passes that region to the parser.

In Ada, the search for a start point finds a block begin, or the point after a block end. If a block begin was found, the matching end is looked for; otherwise, the requested parse point is the end point; 'Minimal_Complete' will provide the missing tokens.

3 EXTENSIONS TO MCKENZIE

We define the following operations that are tried in each McKenzie step:

- 'push_back'
- 'undo_reduce'
- 'Try_Insert_Quote'
- 'Minimal_Complete'
- 'Language_Matching_Begin_Tokens'
- 'Language_Fixes'

3.1 push_back

'push_back' pops the top parse stack item, and moves the input stream pointer back to the first terminal contained by that item. We call the point in the input stream at which insert and delete is done the "edit point"; it may not be an error point. 'push_back' moves the edit point.

3.2 undo_reduce

'undo_reduce' undoes the reduce that produced the top stack item (which must be a nonterminal), replacing the top stack item by the sequence of stack items just before the reduction, without moving the edit point. This requires a syntax tree that records the shifts and reductions done during the parse (a "concrete" syntax tree). This operation serves two purposes;

(1) It allows a subsequenct 'push_back' to push back fewer tokens.
(2) It allows token insertions that would otherwise be forbidden by the grammar.

To illustrate the second point, consider:

```
procedure Example_2
is
   I : Integer;
begin
   procedure Put_Top_10
   is begin
   ...
   end Put_Top_10;
begin
end Example_2;
```

Fig. 2

There is an extra '**begin**' after 'I : Integer'. The error is detected at '**procedure**'; at that point, the parse stack looks like (top is to the left):

```
245 : BEGIN, 208 : declarative_part, 159 : IS, 36 : subprogram_specification, 0 :
```

Here the numbers label the states, terminals are in uppercase, nonterminals in lowercase.

The grammar productions relevant to this example are:

```
declarative_part <= declarations | ;
declarations <= declarations declaration | declaration ;
declaration <= subprogram_declaration | ... ;
```

Fixing the parse error starts by 'push_back **BEGIN**, delete **BEGIN**', leaving ' declarative_part ' on top of the stack. '**procedure**' is the next token, which is illegal in state 208; it starts a 'subprogram_declaration', but we've already "closed" the declaration section by reducing to ' declarative_part '. We could do 'push_back declarative_part ', but that moves the edit point to before the object declaration for 'I', where there is no error and nothing helpful to insert or delete. Instead, 'undo_reduce' leaves the stack as:

```
137 : declarations, 159 : IS, 36 : subprogram_specification, 0 :
```

and now inserting '**procedure**' is legal.

We do not maintain a syntax tree for the parsing done during error correction; doing that proved to be much too slow. Therefore the 'undo_reduce' operation can only be applied to configurations where the top stack item was produced by the main parse, so it has a valid syntax tree entry. An exception is when the nonterminal is empty; that is easy to undo.

3.3 Try_Insert_Quote

Missing string quotes cause problems for the McKenzie algorithm. Consider the code:

```
A : String := Now is the time for all good men";
```

There is a missing quote before 'Now'. In Ada, strings cannot cross newline, and the lexer handles the error by inserting a virtual quote just before the existing one. Then the parser sees a list of identifiers followed by an empty string literal. The McKenzie algorithm would have to delete all the identifiers one by one, with a cost for each.

The 'Try_Insert_Quote' operation attempts to find a better place to insert the string quote, depending on the relative placement of the unbalanced quote and the parse error.

- If the parse error is at the unbalanced quote, assume the unbalanced quote is the intended closing quote, and insert the opening quote one non-empty token before it. Example:

  ```
  A := "for all" & good ";
  ```

  We are in the process of splitting a string across lines; we just added '"␣&', but are missing the '"' before 'good'. This solution inserts that missing quote.
- If the parse error is after the unbalanced quote, assume the unbalanced quote is the intended opening quote, and insert the closing quote at the line end. Example:

  ```
  A := "for all" & "good
  ```

  The missing '"' should be after 'good'. This solution inserts that missing quote.
- If the parse error is before the unbalanced quote, assume the unbalanced quote is the intended closing quote, and insert the opening quote in several places (generate one new configuration for each):
  - before the error token. Example:

    ```
    A := for all good";
    ```

    The missing '"' should be before '**for**'. The parse error is at '**all**'; this solution inserts the missing quote before '**all**', which is almost right.
  - One non-empty token before the unbalanced quote. Same example, but this inserts the '"' before '**all**', which is correct.
  - If there is a string literal on the parse stack, assume the closing quote of that string literal is new (or extra), push back thru the token containing that string literal, and extend the string literal to the unbalanced quote. Example:

    ```
    A := "for all" good";
    ```

    The '"' after '**all**' is extra. The parse error is at 'good'; this solution in effect deletes the extra quote.
    Note that the search for a string literal on the parse stack must take into account nonterminals that may contain a string literal. The set of nonterminals that may contain a string literal is provided as a function call written by the grammar author; it is not computed from the grammar because it should not include higher level nonterminals that are not likely to be contained in a string; for Ada, it stops at expression.

Since the lexer recognizes string literals, we cannot actually insert an unbalanced string quote; we actually delete all tokens between the inserted quote and the unbalanced quote, which matches what the lexer would have returned. This is still much better than the original McKenzie algorithm, because we do all the deletions in one step, with one low cost.

3.4 Minimal_Complete

The 'Minimal_Complete' strategy is to insert the minimum number of tokens to finish the current grammar production. This is supported by precomputing a set of 'Minimal_Complete_Action' for each parser state.

Section 4 gives the algorithm for computing the 'Minimal_Complete_Actions'; here we describe how they are used in error correction.

Each 'Minimal_Complete_Action' is either "insert this terminal token" or "reduce to this nonterminal token".

Consider the code in figure 1. When parsing this code, an error is detected at the final ';'; the parser is expecting '**end loop**;'. The kernel of that state has one production:

```
loop_statement <= LOOP sequence_of_statements END ^ LOOP
   identifier_opt SEMICOLON
```

where the caret (^) shows the parse point (also known as "dot" in an LR(1) item). In this case, it is easy to see that 'Minimal_Complete_Action' must be "insert '**loop**' ". Similarly, after parsing '**loop**', 'Minimal_Complete_Action' is "reduce to ' identifier_opt ' ", and then "insert ';' ". After that, we will be completing the inner '**if then**' statement, and then the outer '**if then**' statement. At that point, the existing '**end** ;' is legal.

To handle cases where only part of a nonterminal production needs to be inserted, the check to see if the original error token is now legal must be performed after each token is inserted; this means that 'Minimal_Complete' inserts at most one token for each cycle of the underlying McKenzie algorithm.

In order to prefer the 'Minimal_Complete' solution over others, we give it a negative cost. In figure 1, the grammar is a small subset of Ada, the default insert and delete cost is 4, delete '**begin**' is cost 1, delete '**end**', ';' are cost 2. With 'Minimal_Complete' cost 0, -1, or -2, no solution for figure 1 is found, with an enqueue limit of 120,000. With 'Minimal_Complete' cost -3, the desired solution is found with cost 9, after enqueueing 6804 configurations and checking 1051.

As usual, there is a trade off here; sometimes other solutions would be better than 'Minimal_Complete'. For example, consider:

```
for I in 1 to Result_Length loop
end loop;
```

Here 'to' should be '..' (this is an actual error the author typed late one night). The error is detected at 'to', so the desired solution is
'delete IDENTIFIER, insert ..'. With the 'Minimal_Complete' cost set to 0, this solution is found, along with 19 other solutions with the same cost; a few of these change the code to:

```
in to.Result_Length loop
in 1 .. to*Result_Length loop
in 1 .. to/Result_Length loop
```

Finding these takes a relatively long time; 1273 configurations were enqueued and 218 checked.

Setting 'Minimal_Complete' cost to -1 finds similar solutions, but more quickly (341 enqueued, 56 checked); ' insert .. ' is done by 'Minimal_Complete', so it is cheaper, and more expensive solutions are not checked.

Setting 'Minimal_Complete' cost to -3 (as required for figure 1) finds one cost 3 solution very quickly (67 enqueued, 10 checked); it changes the code to:

```
for I in 1 .. to loop Result_Length; loop
end loop;
```

which causes another syntax error later in the code (missing '**end loop**;'). Here three tokens were inserted by 'Minimal_Complete'; '.. **loop** ;'.

In the production Ada parser, -3 proves to be a good compromise for 'Minimal_Complete' cost.

In some cases, the action required for 'Minimal_Complete' is reduce, not shift. For example:

```
case Current_Token is
= +Right_Paren_ID then
   Matching_Begin_Token := +Left_Paren_ID;
else
   Matching_Begin_Token := Invalid_Token_ID;
end if;
```

Here the user is in the middle of converting an '**if**' statement to a '**case**' statement. The relevant grammar productions are:

```
if_statement <=
    IF expression_opt THEN sequence_of_statements
    elsif_statement_list ELSE sequence_of_statements
    END IF SEMICOLON
case_statement <=
    CASE expression_opt IS case_statement_alternative_list
    END CASE SEMICOLON
```

An error is detected at '='; 'Minimal_Complete' inserts '**when** NUMERIC_LITERAL =>', then the original McKenzie algorithm inserts '**if** NUMERIC_LITERAL', which makes the remaining code legal, terminating one error correction session. Then parsing proceeds to then end of the input, where another error is enountered; missing '**end case**;'. At that point, the parse state kernel has one production:

```
if_statement <= IF expression_opt THEN sequence_of_statements
   ELSE sequence_of_statements END IF SEMICOLON ^;

Minimal_Complete_Action => if_statement
```

Since the 'Minimal_Complete_Action' token is a nonterminal, the action is reduce, not shift. That leads to several more states where the 'Minimal_Complete_Action' is reduce:

```
compound_statement <= if_statement ^
Minimal_Complete_Action => compound_statement

statement <= compound_statement ^
```

```
Minimal_Complete_Action => statement

  ...
```

and finally arrives at the state:

```
case_statement <= CASE expression_opt IS
   case_statement_alternative_list ^ END CASE SEMICOLON
case_statement_alternative_list <=
   case_statement_alternative_list ^ case_statement_alternative

Minimal_Complete_Action => END
```

which inserts '**end**'. 'Minimal_Complete' does all required reductions, and one insertion, in one McKenzie step (similar to McKenzie insert).

If there is more than one production in the kernel for a parse state that gives the minimum length for that state, there is more than one 'Minimal_Complete_Action' for that state. In that case, we look at the length after the parse point for each production in the kernel for the state that the shift or reduce goes to; if one of the actions results in a minimum length, that action is chosen. If more than one action gives the minimum length, all are kept. Therefore 'Minimal_Complete' maintains a queue of configurations with an action to check. Each may produce a new configuration for the next McKenzie step.

Sometimes the minimal length production cannot be computed at compile time. Consider the Java code fragment:

```
{ B = UpdateText (A }
```

Fig. 3

This is missing '); ' after 'A'. The error is detected at '}'. At that point, the kernel is:

```
LambdaExpression <= Identifier ^ MINUS_GREATER Identifier
LeftHandSide <= Identifier ^
ClassType <= Identifier ^
MethodInvocation <= Identifier ^ LEFT_PAREN ArgumentList RIGHT_PAREN

Minimal_Complete_Action => (MINUS_GREATER, LeftHandSide, ClassType)
```

Here two actions are reduce, and therefore there are no tokens in these productions after the parse point, so we have to look at the next state to find the minimal length production. Reducing to 'LeftHandSide' goes to the state:

```
Assignment <= LeftHandSide ^ EQUAL Expression
```

where there are two tokens needed to complete the production.

Reducing to 'ClassType' goes to the state:

```
PostfixExpression <= ClassType ^
ClassType <= ClassType ^ DOT Identifier
```

Once again, we reduce 'ClassType' to the next state, and the next, until we find a shift. This ends in the state:

```
MethodInvocation <= Identifier LEFT_PAREN ArgumentList ^ RIGHT_PAREN
ArgumentList <= ArgumentList ^ COMMA Expression
```

which has 1 token after the parse point; this is the minimal length. All of these reductions, and the final shift, are done in one McKenzie step. The function that computes the length after parse point is recursive, and explores all paths that might be minimal.

Sometimes 'Minimal_Complete' results in an ambiguous parse. Consider the following Java code:

```
{ UpdateText (A) }
```

In the subset of Java we are using for this example, this is not a complete statement. The relevant productions are:

```
StatementExpression     <= PostIncrementExpression | PostDecrementExpression ;
PostIncrementExpression <= PostfixExpression '++' ;
PostDecrementExpression <= PostfixExpression '--' ;
PostfixExpression       <=
   ClassType | MethodInvocation | PostIncrementExpression
 | PostDecrementExpression ;
ClassType               <= Identifier | ClassType '.' Identifier ;
```

To complete the code fragment, we have to insert either '++' or '−−'. The error is detected at the closing brace; the state is:

```
PostIncrementExpression <= PostfixExpression ^ PLUS_PLUS
PostDecrementExpression <= PostfixExpression ^ MINUS_MINUS
Minimal_Complete_Action => (PLUS_PLUS, MINUS_MINUS)
```

There are two actions, neither reduce, so we enqueue both. The next McKenzie step inserts ';' in each, completing the error recovery session with two solutions. In the main parser, both solutions parse to end of input. In the absence of error correction multiple parallel parsers getting to end of input is an error (ambiguous parse), but since error correction often produces multiple solutions, we don't report an error; the parser with an error solution that has minimum cost and minimum recover ops length is chosen to be the final parse.

In general, we can rely on cost to exclude cycles in error recovery, but this is not true for 'Minimal_Complete', since it has negative cost. We will see in section 4 that minimal actions that might lead to a cycle are dropped at compile time, so there is no need for detecting cycles at runtime.

3.5 Matching_Begin

Consider the Ada code fragment:

```
procedure ... end;


   end if;
end Foo;
```

This is a result of cut and paste. The code is missing
'**procedure** Foo **is begin if** expression **then**' before '**end if**'.

The error is detected at the '**end**' in '**end if**'. 'Minimal_Complete' is no help here; the parse stack looks like:

```
34 : subprogram_body, 0 :
```

'Minimal_Action' in state 34 is reduce to ' compilation_unit ', which is then reduced to ' compilation_unit_list ', which is the grammar start symbol, which has no minimal complete action. So 'Minimal_Complete' has nothing useful to insert.

To the grammar author, the solution is obvious. We capture that knowledge by having the grammar author provide a function
'Language_Matching_Begin_Tokens', which takes the current parse stack and the next three input tokens as input, and returns a list of tokens to try inserting at the edit point.

The 'Language_Matching_Begin_Tokens' function is similar to the table
$E(A, a)$ given by Fischer, Milton, and Quiring in [7] for an LL parser:

$$E(A, a) \equiv x \mid A \Rightarrow^* xay, \texttt{Cost(x) is minimized} \tag{1}$$

However, we are using an LR parser, so that table is not directly applicable. We could try to compute a table that gives the minimal sequence of tokens to insert starting in any state s , and allowing any token a as the next token:

$$M(s, a) \equiv y \mid A \Rightarrow^* xyaz, A \in Kernel(s), \texttt{Cost(x) is minimized} \tag{2}$$

where x is the prefix of the production A in state s. That is the table that the McKenzie algorithm computes on the fly; it is not worth precomputing. It is worth providing 'Language_Matching_Begin_Tokens' to shortcut some common situations.

In Ada, three tokens are required to determine the proper match for '**end**'; consider:

```
case is end case;
loop end loop;
block_1 : begin end Block_1;
package Parent_1.Child_1 is begin end Parent_1.Child_1;
```

In the case and loop statements, the three tokens starting with '**end**' are '**end loop** ;' and '**end case** ;'; here we only need two tokens to determine that the matching begin is '**case**' or '**loop**'. To distinguish between the named block statement and the package declaration, we need three tokens; in a named block statement the name must be a simple identifier, with no dots, so the third token must be ';'.

In Ada, 'Language_Matching_Begin_Tokens' returns the proper statement start token for '**end** ... ', and for '**then**, **else**, **elsif**, **excepti**
For '**when**', it returns '**case** IDENTIFIER **is**', which assumes a partial case statement is more common than a partial exception handler. For any other error token, it returns an empty token list; no guess is better than a bad guess.

'Language_Matching_Begin_Tokens' also returns a flag
'Forbid_Minimal_Complete', which is True when 'Minimal_Complete' would be harmful. In Ada, this is set True when the error point is after '**end**' in one of the '**end**' sequences above; it is better to push back '**end**'.

3.6 Language_Fixes

To take advantage of the redundant block name information in Ada, we provide a general hook 'Language_Fixes'; it takes as input a configuration, the parse table, and the syntax tree and token stream for one parser. It enqueues new configurations to test. Consider:

```
procedure Proc_1
is begin
    Block_1:
    begin
end Proc_1;
```

Here '**end** Block_1;' is missing. The grammar rules for Ada do not require the start and end block names to match; that is checked later in the compilation process. To take advantage of it for error correction, we add that check as a parse-time action in the grammar declaration:

```
block_statement <=
    block_label_opt BEGIN handled_sequence_of_statements END
    identifier_opt SEMICOLON
    %()%
    %(return Match_Names
        (Lexer, Descriptor, Tokens, 1, 5, End_Name_Optional);)%
```

Here the first '%()%' gives the post-parse action, which is run after the parse is complete and the syntax tree is available; Emacs uses this action to compute indent, navigation, and highlight. The second '%()%' gives the in-parse action, which is run when the production is reduced, both in the main parse and during error correction.

Here 'Match_Names' will return a status of 'Match_Names_Error' if the names do not match, with the error point after the final ';', and the production not reduced (so it is possible to edit the token sequence without an 'undo_reduce'). 'Language_Fixes' then tries to determine the best fix based on the name information.

In this example, the error is reported after '**end** Proc_1;'.
'Language_Fixes' finds the matching '**procedure** Proc_1' on the parse stack, and inserts '**end** Block_1;' before '**end** Proc_1;'.

Consider:

```
package Parent_1.Child_1
is begin
    begin
end Parent_1.Child_1;
```

Here we might expect 'Match_Names' will fail with 'Extra_Name_Error', but instead we get a parse error on '.' in 'Parent_1.Child_1'; block names must be simple identifiers. Since this is similar to 'Match_Names_Error', 'Language_Fixes' handles it, finding the matching name; the fix is to insert '**end** ;' before
'**end** Parent_1.Child_1;'.

Consider:

```
Block_1 :
begin
    if A then
        null;
end Block_1;
```

Here we get a syntax error on the final 'Block_1'; '**if**' is expected. Again, 'Language_Fixes' handles it, finding the matching name.

More complex patterns can be recognized in 'Language_Fixes'; see the production code for the Ada parser.

4 COMPUTING MINIMAL_COMPLETE

4.1 Grammar recursions

We first compute the recursions in the grammar, so we can exclude cycles from 'Minimal_Complete'. A '*recursion*' is the result of a cycle in the grammar productions; for example:

```
association_list <=
    association_list COMMA association_opt
  | association_opt
```

The first right hand side (RHS) is direct left recursive; the second is not recursive.

Recursion can also be indirect; consider:

```
name <=
    IDENTIFIER
  | selected_component

selected_component <= name DOT IDENTIFIER
```

Together, 'name, selected_component' are indirect recursive.

We can form a graph representing the grammar by taking the nonterminals as the graph vertices, and if there is a production $A \Rightarrow xBy$ then there is a directed edge from the A to B. Then recursion is represented by a cycle in the graph.

In a useful grammar, every recursion must have a non-recursive RHS in one of the nonterminals, to terminate the recursion.

We use Johnson's algorithm [12] to find the cycles in the graph. However, that algorithm does not apply to '*multi-graphs*', which have more than one edge connecting any two nodes. Real grammars can be multigraphs, so we first filter out all such edges, and add them back after we compute the cycles.

Some languages have a lot of recursion, so it can be prohibitive to compute the exact set of cycles in the graph. For example, Java SE 12 (using the grammar given in chapter 19 of the language reference manual [8]) takes too long to compute. In that case, we only compute the strongly connected components (SCCs) (which is one of the steps in Johnson [12]), and use that as the recursion. This gives more recursion than necessary, and thus reduces the number of minimal actions computed, but still gives good performance in error correction. For Ada 2012, computing the full recursion is fast, but for Ada 2020 draft 25 [10], it is prohibitively slow.

Once we have the cycles or SCCs, we add a Boolean 'recursive' flag to the items in the state kernels; true if the production is in a cycle or SCC and the recursive token is the first token (ie the production is left recursive, direct or indirect), false otherwise.

4.2 Other preliminaries

We also need the minimal terminal token sequence for each production. This is the same as $S(A)$ from Fischer et al. [7]:

$$S(A) \equiv x \in V_t^* \mid A \Rightarrow^* x, \texttt{Cost(x) is minimized} \tag{3}$$

where the cost of each token is 1. Including individual token costs at this point would make it very difficult to assign useful costs; we only use token costs in the run-time portion of the algorithm.

Next we need 'Minimal_Terminal_First(A)', which is the first token in $S(A)$, or the invalid token ξ if $S(A)$ is empty.

Finally we need $Nullable(A, \omega)$, which gives the production that reduces A to ϵ:

$$Nullable(A, \omega) \equiv (B, \psi) \in P \mid (A \Rightarrow B\gamma, B \Rightarrow^* \epsilon) \; else \; \xi \tag{4}$$

We say a nonterminal token 'A' is "nullable" if $Nullable(A, \omega)$ returns ξ for some ω.

4.3 Minimal Complete Action

For each state, we consider each item in the kernel. There are several possible cases, listed in priority order:

(0) There is only one item in the kernel. Recursion is ignored, because any other McKenzie operation also has no choice here. The minimal action is given by 'Compute_Action (Dot)' (see figure 4), where 'Dot' is the token after dot in the kernel item.

(1) Dot is at the end of the production, or all tokens following dot are nullable; the actual production length must be computed at runtime. If Dot is at the end of the production, recursion is ignored again because any other McKenzie operation also has no choice here; the minimal action is reduce to the item LHS. If Dot is not at the end of the production, we cannot ignore recursion; the null token might be in a recursion cycle.

(2) The item is left recursive; there is no minimal action.

(3) There is no recursion, and dot is not at the end of the production. If the number of tokens after dot is minimal within the kernel, include 'Compute_Action (Dot)' in the minimal actions.

```
function Compute_Action (Token)
   if Token in Terminals then
      return Action (State, Token);
   else
      if Minimal_Terminal_First (Token) = Invalid_Token then
         return (Reduce 0 tokens to Token);
      else
         return Action (State, Minimal_Terminal_First (Token));
      end if;
   end if;
end Compute_Action;
```

'Action (State, Token)' returns the parse action for the token in the state.

Fig. 4

5 IMPLEMENTATION

The algorithm presented here is implemented in the WisiToken parser [14], and used in Emacs ada-mode.

McKenzie uses a bit map to prune redundant (but presumably higher cost) configurations from the queue; instead, we use a Fibonacci min heap (Cormen, Leiserson, and Rivest Cormen et al. [5] chapter 19), so finding the minimal cost configuration is asymptotically free.

Checking each configuration to see if it is a solution, and generating new configurations from it, is independent of all other configurations, so the underlying McKenzie algorithm is easily parallelized. We use one Ada protected object to store the min heap of configurations to check, and one Ada task per processor to check and generate configurations. However, that results in only 40% speedup with 8 processors; more work is needed in this area.

The configure data structure is optimized for speed; it has a bounded parse stack of 70 items, and a bounded vector of 80 recover operations; creating a new configuration on the min heap requires only one fixed length memory allocation. Hitting either of those limits is not at all likely in a low cost solution, so we simply drop any configurations that do so.

We use an LR1 parse table, not LALR. The extra information retained by the LR1 parse table is helpful in error recovery. Using the same stress-test case as the parallel task speed test, with the LALR parser there are 4 parsers active when recovery is entered, they enqueue a total of 1_094_252 configurations (two hit the enqueue limit of 500_000), and check a total of 48_035 configurations. With the LR1 parser, there are two parsers active when recovery is entered, they enqueue 558_102 configurations (one hit the enqueue limit; a gain of almost 50%) and check 47_530. On the other hand, due in part to using multiple tasks, the total time spent in recovery is about the same between the LALR and LR1 parsers; 1.10 seconds in this case.

6 RESULTS

This error correction algorithm has been in production use in Emacs ada-mode since November 2018. The parser can be configured to output information about each error recovery; that is summarized in figures 5 and 6 for one month of the author's use. The enqueue limit is set at 58_000; there is a noticeable delay when that limit is hit. The maximimum enqueue value in the table is higher because multiple tasks are used in error recovery; each task is allowed to finish its current operation before aborting due to the limit.

This shows that the minimal complete operation is used in a large majority of cases, along with insert.

Fig. 5. Error recovery statistics for full file parsing

	count	percent
fail enqueue limit	1_126	3%
ignore_error	2_130	1%
language_fix	11_138	6%
minimal_complete	151_977	78%
matching_begin	2_848	1%
push_back	13_694	7%
undo_reduce	7_491	4%
insert	141_629	72%
delete	18_557	9%
string_quote	2_615	1%
	mean	max
enqueue	796.3	60_147
check	61.8	5496

Fig. 6. Error recovery statistics for partial file parsing

	count	percent
fail enqueue limit	16	0%
ignore_error	850	5%
language_fix	356	2%
minimal_complete	14_579	77%
matching_begin	2_355	13%
push_back	1_019	5%
undo_reduce	544	3%
insert	6_076	32%
delete	1_266	7%
string_quote	11	0%
	mean	max
enqueue	922.6	49_286
check	78.5	4201

To compare our error correction to other parsers, we use a set of 59 Ada source files with known errors - this is the set of files used to test Emacs indentation etc. We write code to use the parsers to output the corrected string of tokens found for each file. Then we difference that string from the nominal correct string, using the ‘ diff ’ program. The length of the diff output is then a measure of error correction quality (shorter is better).

libadalang is a parser provided by AdaCore [2], used in their GNAT Studio IDE (although not yet for indentation; see AdaCore [1]). The results of comparing wisitoken with libadalang are given in figure 7. Here "perfect files" is the count of files where the corrected token stream is the same as the nominally correct token stream; "better files" means the diff is shorter than the other algorithm. Overall, WisiToken does a better job, although there are 7 files where libadalang does a better job.

Fig. 7. error comparison metrics with full Ada grammar

	total diff size	perfect files	better files
wisitoken	13_097	23	49
libadalang	27_935	4	7

Tree-sitter [4] is an incremental parser, designed for use in IDEs. However, it cannot cope with the full Ada grammar, so we used the Ada subset grammar to compare error correction. The test files were edited to conform to the subset grammar; three became meaningless after that and were dropped. The results are in figure 8

Fig. 8. error comparison metrics with subset Ada grammar

	total diff size	perfect files	better files
wisitoken	20_450	12	51
tree-sitter	47_047	1	4

REFERENCES

[1] AdaCore. Adacore gnat studio, 2020. URL https://www.adacore.com/gnatpro/toolsuite/gnatstudio.
[2] AdaCore. Adacore libadalang, 2020. URL https://github.com/AdaCore/libadalang.
[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1985.
[4] Max Brunsfeld. tree-sitter home page, 2020. URL https://github.com/tree-sitter/tree-sitter.
[5] Thomas H. Cormen, Charles Eric. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.
[6] Frank Deremer and Thomas Pennello. Efficient computation of lalr(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(4):615–649, 1982. doi: 10.1145/69622.357187.
[7] C. N. Fischer, D. R. Milton, and S. B. Quiring. Efficient ll(1) error correction and recovery using only insertions. *Acta Informatica*, 13(2):141–154, 1980. doi: 10.1007/bf00263990.
[8] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. Java language grammar, 2019. URL https://docs.oracle.com/javase/specs/jls/se12/html/index.html.
[9] Ada Working Group. Ada 2012 language standard, 2012. URL http://ada-auth.org/standards/ada12.html.
[10] Ada Working Group. Ada 2020 draft 25, 2020. URL http://www.ada-auth.org/standards/ada2x.html.
[11] Dick Grune and Ceriel J.H Jacobs. *Parsing Techniques; A Practical Guide*. Springer Science, second edition, 2008.
[12] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975. doi: 10.1137/0204007.
[13] Stephen Leake. Ada mode homepage, 2020. URL http://www.nongnu.org/ada-mode/NEWS-ada-mode.text.
[14] Stephen Leake. Wisitoken home page, 2020. URL https://stephe-leake.org/ada/wisitoken.html.
[15] Bruce J. Mckenzie, Corey Yeatman, and Lorraine De Vere. Error repair in shift-reduce parsers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(4):672–689, 1995. doi: 10.1145/210184.210193.
[16] Masaru Tomita. *Efficient parsing for natural language*. Kluwer Academic Publishers, 1986.