AN LR(k) ERROR DIAGNOSIS AND RECOVERY METHOD

Philippe Charles
IBM T.J. Watson Research Center
P.O. box 704,
Yorktown Heights, N.Y 10598

Abstract

In this paper, a new practical, efficient and language-independent syntactic error recovery method for LR(k) parsers is presented. This method is similar to and builds upon the three-level approach of Burke-Fisher [11]. However, it is more time- and space-efficient and fully automatic.

1 Introduction and Overview

1.1 The Parsing Framework

An LR *parsing configuration* has two components: a state stack and the remaining input tokens. This method assumes a framework in which the parser maintains a state stack, denoted *stack*, and a fixed number of input symbols. These symbols include the current token or *lookahead*, denoted *curtok*, the token immediately preceding the current token, denoted *prevtok*, and an input buffer, denoted *buffer*, containing a predetermined number of the input tokens following *curtok*. A number of attributes are associated with each input symbol such as its class, its location within the input source, its character string representation, etc ...An input symbol together with all its attributes is referred to as a *token element*. Each state q in the state stack is also associated with certain attributes including the grammar symbol that caused the transition into q (called the *in_symbol* of q), and the location of the first input token on which an action was executed on q.

An LR parsing configuration may be represented by a string of the form:

$$q_1, q_2, \ldots, q_m \quad | \quad t_1, t_2, \ldots, t_n.$$

The sequence to the left of the vertical bar is the content of the state stack, with q_m at the top; $q_1 \ldots q_m$ is a valid sequence of states in the LR parsing machine. The sequence to the right of the vertical bar is the unexpended input. Each element t_i represents the class of a corresponding input symbol. The symbol t_1 represents the class of the current token, t_2 represents the class of the successor of *curtok*, etc. The symbol t_0 which is not shown above represents the class of *prevtok*.

For simplicity, it will be assumed that the grammar used to construct the parser is LR(1), but this method is applicable to all forms of LR(k) parsers.

1.2 Error Recovery

A parsing configuration in which no legal action is possible is called an *error configuration*. When an error configuration is reached, the error recovery procedure is invoked. Its role is to adjust the configuration so as to allow the parser to advance a minimum predetermined distance in the input stream, usually two or three tokens past the repair point. The token on which the error is detected is referred to as the *error token* and the state in which the error is detected is called the *error state*.

Three kinds of recovery strategies are used. They are:

- *Primary recovery.* A single symbol modification of the source text; i.e., the insertion of a single symbol into the input stream, the deletion of an input token, the substitution of a grammar symbol for an input token or the merging of two adjacent tokens to form a single one. Previous authors [7][11] have used a more restricted form of primary recovery involving only terminal symbols as repair candidates.
- *Secondary recovery.* Deletion of as small a sequence of tokens as possible in the vicinity of the error token or replacement of such a sequence with a nonterminal symbol. This approach can be viewed as an **automatic** generalization of the *error productions* method described in [3].
- *Scope recovery.* A scope is a syntactically nested structure such as a parenthesized ex-

```
 1. program TEST(INPUT, OUTPUT);
 2.     var X,Y: array[] of integer;
                      ^
*Error: index_list expected after ...
 3. begn
    ^
*Error: misspelling of BEGIN
 4. 1:    x := y,
                ^
*Error: ; expected instead of this token
 5.      if x == b then begin
                ^
*Error: Unexpected symbol ignored
 6.         go to 1;
            <--->
*Error: Symbols merged to form GOTO
 7.      a := ((b + c)
                     ^
                     ^
*Error: ")" inserted to complete phrase
*Error: "END" inserted to complete ...
 8. end.
```

Figure 1: Primary phase recoveries

pression, a block or a procedure. In scope recovery, the strategy is to recover by inserting relevant symbols into the text to complete the construction of scopes that are incompletely specified.

This error recovery scheme consists of two phases called *Primary phase* and *Secondary phase.* In the *Primary phase*, an attempt is made to recover with minimal modification of the remaining input stream. Figure 1 shows some examples of primary phase recoveries. In the *Secondary phase*, more radical approaches involving removal of some left context (state stack) information as well as multiple deletion of tokens from the input stream (right context) are attempted. Figure 2 shows some examples of secondary phase recoveries.

1.3 Error Detection

A canonical LR(k) parser has the capability of detecting an error at the earliest possible point. However, because of their size, canonical LR(k) parsers are seldom used. Instead, variants such as LALR(k) and SLR(k) (usually $k = 1$), invented by DeRemer [1][2] are used. These LR variants, in part, solve the space problem by always using the underlying LR(0) automaton. However, certain states in these parsers usually contain reduce actions that may be illegal, depending on the actual context. Illegal reduce actions do not cause the resulting parser to accept illegal inputs, but they prevent it from always detecting errors at the earlier possible point. This problem is usually compounded by a space-saving technique known as *default reductions* which is often used in compressing parsing tables. To apply the default reductions technique, the most common rule by which the parser can reduce in each state is chosen as a default action for that state and all the reduce actions by that rule are removed from the parsing table. Another undesirable side effect of using default reductions is that it is no longer possible to compute, from the parsing table, the set of terminal symbols on which valid actions are defined in a given state. The inability to detect errors as soon as possible and to obtain a set of viable terminal candidates for a given state is very problematic for error recovery.

```
 1. program P(INPUT,OUTPUT);
 2.     procedure P(X:INTEGER):integer;
                              <------>
*Error: Unexpected input discarded
 3.     begin
 4.     end;
 5. begin
 6.    if count[listdata[sub] := 0 then
                            ^
                              ^
*Error: "]" inserted to complete phrase
*Error: invalid relational_operator
 7.        a := ((b + c ]];
                      ^
                      ^
                        <>
*Error: ")" inserted to complete phrase
*Error: ")" inserted to complete phrase
*Error: Unexpected input discarded
 8. end.
```

Figure 2: Secondary phase recoveries

Furthermore, even with a canonical LR(k) parser, the ability to detect an error at the *earliest possible point* only guarantees that the prefix parsed up to that point is correct. Therefore, it is possible that the token on which an error is detected is not the one that is actually in error. Consider the following Pascal declaration:

```
FUNCTION F(X:TINY, Y:BIG, Z:REAL);
```

In this example, it is very difficult to deduce the actual intention of the programmer, but a simple substitution of the keyword "PROCEDURE" for the keyword "FUNCTION" would solve the problem. However, the error is not detected until the semicolon (;) is encountered or 15 tokens later.

In [11], Burke and Fisher introduced a *deferred parsing* technique where two parsers are run concurrently: one that parses normally and another that is kept at a fixed distance (measured in terminal symbols) back. When an error is encountered, error recovery is attempted at all points between the two parsers. This approach avoids the premature reductions problem and solves, in part, the problem of late detection of errors. However, the overhead of the two parsers penalizes correct programs.

In this method, a new LR driver routine called *deferred driver* is introduced. This new driver can effectively detect an error at the earliest possible point even if the parser contains default reductions. It can also be adapted to defer parsing actions on a fixed number of tokens with very little slow-down on correct programs. To achieve this goal, an additional state stack is required for each deferred symbol. Thus, in practice, one must restrict the number of symbols on which actions are deferred.

The method also relies on having two mappings: *t_symbols* and *nt_symbols*, statically constructed, which yield for each state, a subset of the terminal and nonterminal symbols, respectively, on which an action is defined in the state in question. These subsets are the smallest subsets of viable error recovery *candidates* for each state. Their computation will be discussed later.

The remainder of this paper is organized as follows:

- detailed description of the new driver
- presentation of various recovery techniques
- discussion of how to apply these recovery techniques
- concluding remarks

2 The Driver

An important improvement that can be made to an LR(k) automaton is the removal of *LR(0) reduce states*. An LR(0) reduce state is a state that contains only reduce actions by a particular rule. If a representation of the parsing tables with default action is used, then the parser will never consult the lookahead symbol when it is in one of these states. Thus, such states may be completely removed from the parser by introducing a new parsing action: *read-reduce*. The read-reduce action comprises a read transition followed by a reduction. A read-reduce action is referred to as a *shift-reduce* when the symbol X in question is a terminal symbol and as a *goto-reduce* action when X is a nonterminal.

```
# let #x denote the number of elements in a
# sequence x. rhs and lhs are maps that yield the
# size of the right-hand side and left-hand side
# symbol of a given rule, respectively. ACTION
# and GOTO are the terminal and nonterminal
# parsing functions, respectively.
function lookahead_action(stk, tok, pos);
{   pos := #stk.state;
    top := pos - 1;
    act := ACTION(stk.state[pos], tok);
    while act is a reduce action do
    {   do
        {   top := top - rhs[act] + 1;
            if top > pos then
                s := tstk[top];
            else s := stk.state[top];
            act := GOTO(s, lhs[act]);
        } while act is a goto-reduce action;
        tstk[top+1] := act;
        act := ACTION(act, tok);
        pos := min(pos, top);
    }
    return act;
}
```

Figure 3: *lookahead_action* function

The removal of LR(0) reduce states from an LR automaton does not cause premature reductions. Moreover, the execution of a read-reduce action is always followed by a sequence of zero or more goto-reduce actions, and finally, by a goto action. All of these actions may also be executed without deferral.

When the parser executes a reduce action in a non-LR(0) reduce state, that action is also followed by goto-reduce actions and a final goto action. If the reduce action in question is an illegal action, executed by default, then all the associated goto-reduce and goto actions following it are also illegal moves. To complicate matters, the goto action may be followed by a sequence of reduce actions on empty rules, each followed by its associated goto-reduces and goto action. In such a case, all actions induced by the lookahead symbol must be invalidated and the original configuration of the parser (prior to the initial reduction) must be restored.

One way to achieve this goal is as follows. When a reduce action is encountered, make a copy of the state stack into a temporary stack and simulate the parser using the temporary stack until either a shift, shift-reduce or error action is com-

```
stk.state := [start_state];
loop do
{   ppos := 0;   pstk := [ ];
    npos := 0;   nstk := [ ];
    stk.loc[#stk.state] := curtok.loc;
    tstk := stk;
    act := lookahead_action(tstk, $t_1$, pos);
    while act $\neq$ error and act $\neq$ accept do
    {   nstk[npos+1..] := tstk[npos+1..];
        stk.loc[pos+1..] :=
            [curtok.loc : i in [pos+1..#nstk]];
        if act is a shift-reduce action then
        {   top := #nstk;
            do
            {   top := top - rhs[act] + 1;
                act := GOTO(nstk[top], lhs[act]);
            } while act is a goto-reduce action;
            nstk[top+1..] := [act];
            pos := min(pos, top);
        }
        act := lookahead_action(nstk, $t_2$, npos);
        if act $\neq$ error then
        {   get next token;
            pstk[ppos+1..] := stk.state[ppos+1..];
            ppos := pos;
            stk.state[pos+1..] := nstk[pos+1..];
            pos := npos;
        }
    }
    if act =accept then
        return;
    error_recovery();
}
```

Figure 4: *Driver with 3 deferred tokens*

puted on the lookahead symbol. If the first non-reduce action computed on the lookahead is valid, the temporary state stack is copied into the state stack and the parsing can continue. Otherwise, the error recovery routine is invoked with the unadulterated state stack. This idea captures the essence of what needs to be done, but it is too costly for practical use.

Instead of copying the information, the temporary stack is used to hold the values of the contiguous elements of the state stack that have been added or rewritten. If the moves turn out to be valid, then only the added or rewritten elements are copied to the state stack. Otherwise, the original configuration is passed to the error recovery routine. This idea is illustrated in the *lookahead_action* function of Figure 3, written in pseudo-code.

The *lookahead_action* function always returns the first non-reduce action computed on the lookahead symbol. If that action is valid, the state sequence of the new configuration consists of the elements $1..pos$ of $stk.state$ and the elements $pos + 1..top + 1$ of $tstk$.

A parser with actions deferred on one token can be constructed as follows. Starting with the initial configuration, the parser advances through the input stream one token at a time after verifying that the token in question is a valid input by invoking the *lookahead_action* function. When the *lookahead_action* function is invoked with a valid lookahead it returns either a shift or a shift-reduce action which is processed immediately. As mentioned earlier, shift-reduce actions and all their associated goto-reduce and final goto actions may be processed without deferral. After successfully processing a token, the next token is read in and the process is repeated on the new configuration. If, on the other hand, the *lookahead_action* function returned the error action, the state stack is not updated and the error recovery routine is invoked instead.

A driver routine can be constructed, using the *lookahead_action* function, to defer parsing actions on n tokens given n state stacks. In experiments with this method, parsing has been deferred for three tokens. The three stacks that are used are: pstk which captures the configuration of the parser prior to processing any action induced by *prevtok*, stk which captures the configuration prior to processing actions induced by *curtok*, and nstk which captures the configuration prior to processing actions induced by the successor of *curtok*. Associated with each of these stacks are three integer variables: ppos, pos and npos which are used to mark the position of the top element in the corresponding stack that is still valid after the actions induced by the relevant lookahead symbol are applied. Figure 4 shows the body of a driver routine with actions deferred on three input symbols.

3 Recovery Strategies

Each recovery attempt is called a *trial*. The effectiveness of a recovery is evaluated using a validation function: *parse_check*, which indicates how many tokens in the input buffer can be successfully parsed after the repair in question is applied: *parse_check distance*. A recovery trial is not considered successful unless the parse_check distance is greater than or equal to a certain value, called *min_distance*. Experiments have shown that a good choice for *min_distance* is 2 [11].

The *parse_check* function is essentially an LR driver that simulates the parse until it has either shifted all the tokens in the buffer, completed the parse successfully, or reached a token in error.

In the following subsections, algorithms for optimizing the necessary error recovery information and implementing the three different recovery strategies are presented.

3.1 Primary Recovery

Given a configuration: $q_1, q_2, \ldots, q_m \mid t_1, t_2, \ldots, t_n$, where t_1 is assumed to be the error token, the primary recovery finds the best possible *primary repair* (if any) for that configuration. The selection of a best primary repair is based on three criteria:

- the parse_check distance
- the *misspelling index*
- the order in which the trials are performed.

The misspelling index is a real value between 0.0 and 1.0 that is associated with each primary recovery trial. When a new token is substituted for the error token - a *simple substitution*, a misspelling function is invoked to determine the misspelling index; i.e., the relative proximity of the two tokens in question expressed as a probabilistic value. For other kinds of recoveries, the misspelling index is set to a constant value depending on the recovery in question and other conditions. This will be discussed later.

Primary recoveries are attempted in the following order: merging of the error token (t_1) with its successor (t_2); deletion of t_1; insertion of each terminal candidate in $t\_symbols(q_m)$ before t_1; substitution of each legal terminal candidate in $t\_symbols(q_m)$ for t_1; insertion of each nonterminal candidate in $nt\_symbols(q_m)$ before t_1; and, finally, substitution of each nonterminal candidate $nt\_symbols(q_m)$ for t_1; For now, one can assume that for a state q, $t\_symbols(q)$ and $nt\_symbols(q)$ yield the sets of all terminal and nonterminal symbols, respectively, on which actions are defined in q. Optimization of these sets is discussed in section 3.3.

As the trials are performed, the primary recovery routine keeps track of the most succesful trial. Initially, the merge recovery is chosen since it is attempted first. If a subsequent recovery yields a larger parse_check distance than the previously chosen recovery or it yields the same parse_check distance but with a greater misspelling index, then it is chosen instead as the best recovery candidate.

For the merge trial, the character string representation of t_2, is concatenated to the character string representation of t_1 to obtain a merged string s. A test is then performed to determine if s is the character string representation of some $t \in t\_symbols(q_m)$. If such an element t, called a *merge candidate*, is found, a new configuration is obtained by temporarily replacing t_1 and t_2 with t in the input sequence and the parse_check distance is computed for this new configuration.

As described in the previous section, the deferred driver insures that the state q_m on top of the stack of the error configuration is the state entered prior to the execution of any action on t_1. In that configuration, it may be possible to execute a sequence of reduce, goto-reduce and goto actions before the illegality of t_1 is detected in another state q_e. In such a case, the elements in $t\_symbols(q_m)$ that are also in $t\_symbols(q_e)$ are given priority in applying the insertion and substitution trials. (It is not hard to show that $t\_symbols(q_e) \subseteq \mathrm{t\_symbols(q_m)}$.) The benefits of this ordering can be seen in the following example:

```
write(1*5+6;2*3,4/2)
```

In this erroneous Pascal statement, a semicolon is used instead of a comma after the first parameter. Assume state q_m is the first state that encounters the semicolon. At that point, the parser has just shifted an expression operand and the set of valid lookahead symbols includes not only the comma but all the arithmetic operators. However, if the parser is allowed to interpret the operand as a complete expression, it will enter an error state q_e where the comma is the only candidate.

In order to give priority to the candidates in an error state q_e, it is necessary to identify when the parser has entered such a state. State q_e can be computed in the *lookahead_action* function by inserting the following statement after lines 3. and 13. in Figure 3:

```
error_state := act;
```

3.1.1 The Mispelling Index

For a successful merge trial, the misspelling index is set to 1.0 since the merged string must perfectly match the character string representation of the merge candidate.

As mentioned earlier, a misspelling function is invoked to calculate the misspelling index for a simple substitution. The misspelling function used in this method was proposed by Uhl [14]. The distance between two words is measured by the number of letter inversions, insertions and substitutions. The smaller the distance between two words, the more likely it is that one is a misspelling of the other.

For all other recoveries, the misspelling index is set to 0.0.

3.2 Secondary Recovery

Secondary recovery (also called *Phrase-level recovery* [8] [12]) is based on the identification of an *error phrase* which is then deleted from the input or replaced by a suitable nonterminal symbol or *reduction goal.* If the string:

$$q_1, \ldots, q_m \quad | \quad t_1, \ldots, t_n \tag{1}$$

is an error configuration, then a substring

$$q_{i+1}, \ldots, q_m \quad | \quad t_1, \ldots, t_{j-1} \tag{2}$$

$1 \leq i \leq m$, $1 \leq j \leq n$, of that configuration is an error phrase (of the configuration) if removing that substring allows the parser to advance at least *min_distance* tokens into the forward context, or if there is a nonterminal A such that a valid action is defined in state q_i on A, and after processing A, the parser can advance at least *min_distance* into the forward context. Here, q_i, A and t_j are the *recovery state*, *reduction goal* and *recovery symbol*, respectively.

The scheme used in this method to select error phrases reflects a fundamental distinction that is made among three different kinds of errors. Consider the error configuration (2) above. The case of the empty error phrase is considered during primary recovery as a nonterminal insertion. Similarly, the case where an error phrase $\epsilon|t_1$ is deleted or replaced by a nonterminal candidate is processed by a primary recovery deletion or nonterminal substitution. Next, priority is given to a successful secondary recovery that consumes no input symbol and requires no insertion of a reduction goal; i.e., a recovery based on the removal of an error phrase of the form $\beta|\varepsilon$ where $\beta \neq \epsilon$. This kind of error is called a *misplacement error*, and β is called a *misplaced phrase.* The following Pascal program illustrates this case:

```
 1. program P(INPUT,OUTPUT);
 2.    var I:real;
       <--------->
*Error: Misplaced construct(s)
 3.    type ORDER=array[1..MAX] of real;
 4.    var Q:integer;
 5. begin
 6. end.
```

Finally, the case in which one or more input symbols and/or states must be deleted or replaced with a nonterminal candidate is considered. In that case, input symbols are consumed faster than states. In other words, the error phrases are selected as indicated by the row-major order of the table below:

$$\begin{array}{ccc}
\varepsilon|\varepsilon & \cdots & \varepsilon|t_1, \ldots, t_n \\
q_m|\varepsilon & \cdots & q_m|t_1, \ldots, t_n \\
\vdots & & \vdots \\
q_2, \ldots, q_m|\varepsilon & \cdots & q_2, \ldots, q_m|t_1, \ldots, t_n
\end{array}$$

In this final case, each error phrase selected is removed from the base configuration (1). An initial attempt is made to recover by parse checking the resulting configuration. This action, called *secondary deletion*, can be viewed as a multiple deletion of the symbols that make up the error phrase. Next, each element in the set of nonterminal candidates for the newly exposed state on top of the state stack is substituted, in turn, for the error phrase and the *parse_check* function is invoked to determine its viability. This action is called a *secondary substitution.* This process continues until a successful recovery is found or all the possibilities are exhausted.

In secondary recovery, the aim is to find a repair that least alters the original configuration. For this reason, misplacement trials are performed separately from the other secondary trials and given higher priority, since such a repair does not delete any symbol from the forward context and tends to remove whole structures from the left context that have been previously analysed. The *parse_check* distance is used as the criterion to select the best misplacement repair. After the misplacement trials, a secondary deletion and substitution trial is performed on successive error phrases. The selection of a best deletion or substitution repair is based on the length of the relevant error phrase and the *parse_check* distance, with deletion having priority over substitution in case of a tie. The length of an error phrase $\beta|x$ is obtained by adding the length of the string x to the number of non-null symbols in β.

Given the best misplacement repair and the best deletion or substitution repair, if the misplacement repair is based on a shorter error phrase or it yields a longer *parse_check* distance, then it is chosen. Otherwise, the deletion or substitution is chosen.

3.3 Optimization of Candidates

Consider the case of a secondary substitution in which a recovery goal A must be inserted into the input stream. In such a case, every nonterminal

$$\begin{array}{ll} E \rightarrow \cdot E + T & E \rightarrow \cdot T \\ T \rightarrow \cdot T * F & T \rightarrow \cdot F \\ F \rightarrow \cdot F \uparrow P & F \rightarrow \cdot P \\ P \rightarrow \cdot id & P \rightarrow \cdot (E) \end{array}$$

Figure 5: Items in a state q_i

candidate in state q_i is a potential reduction goal. However, an implementation that checks all potential candidates for each error phrase would be prohibitively slow.

Two optimizations are applied to the set of nonterminal candidates in a given state to obtain, in most cases, a substantially reduced subset of *relevant reduction goals.*

In [8], the following concept is presented: a reduction goal A of error phrase $\beta|x$ in error configuration $\alpha\beta|xy$ is *important* if $\beta|x$ has no reduction goal B such that $B \rightarrow^{+} A$. In this method a more restricted concept of an important symbol is used. The new concept takes into consideration the full context of the error phrase. A nonterminal A on which a transition is defined in a state q_i is said to be *important* if A does not appear in a single item of the form $B \rightarrow \cdot A$ in q_i. For example, assume a recovery state q_i contains the set of items shown in Figure 5. By the definition of [8], the only important reduction goal in such a state is E, since T, F and P can be derived from E via a chain of unit productions. By the more restricted definition of this method, T and F would also be considered important symbols since they appear immediately to the right of the dot in more than one item. To understand the importance of T and F, assume that the rules from which the items of Figure 5 are derived are all the productions of a grammar and consider the following erroneous input strings:

```
( ) ) ( * id + id
( ) ) ( ↑ id + id
```

If E is the only important symbol considered, then the best secondary repair that is achievable is the replacement of "`( ) ) ( * id`" by E in the first sentence and "`( ) ) ( ↑ id`" by E in the second sentence. However, it is clear from the grammar that replacing "`( ) ) (`" by T in the first sentence and by F in the second sentence would be preferable.

One further notices that using F as a reduction goal in the first sentence would have worked just as well, since after a transition on F, with the symbol "*" as lookahead, a reduction by the rule "$T \rightarrow F$" would be applied. Similarly, P could have been used as a suitable reduction goal in both sentences. This leads to the following concept, on which the second optimization is based: a nonterminal element C of a set of nonterminal candidates S in an LR state q is said to be *relevant* with respect to S if there does not exist a nonterminal D, such that $D \in S$, $D \neq C$, and D can be successfully substituted for C as a reduction goal for any error phrase with q as the recovery state.

Given a set S of nonterminal candidates for a given state, the objective is to find the largest subset $S' \subseteq S$ such that S' contains only relevant reduction goals. Let $S = \{B_1, \ldots, B_k\}$ for $1 \leq i \leq k$, $B_i \in S$ is relevant iff $\nexists B_j$, $j \neq i$, such that $B_i \Rightarrow^{+}_{rm} B_j$. The proof of this assertion follows directly from the definition of an LR parser. If a nonterminal B can be substituted for an error phrase, then the recovery symbol t in question must be a valid lookahead symbol for any rule derivable from B. In particular, if $B_i \Rightarrow^{+}_{rm} B_j$ and B_j is substituted for an error phase where B_i is known to be a valid reduction goal, the recovery symbol will cause $B_j$to be reduced to B_i.

For each state q in an LR automaton, the set $nt\_symbols(q)$ is obtained as follows. Starting with the set of nonterminal symbols on which an action is defined in q, remove all unimportant symbols from that set, and reduce the resulting set further by removing all irrelevant reduction goals from it. For example, consider the state q_i of Figure 5. State q_i contains nonterminal transitions on the symbols E, T, F and P. The only unimportant symbol in that set is P. After P is removed, the irrelevant symbols E and T are removed from the subset $\{E, T, F\}$ leaving F as the only relevant reduction goal in q_i.

The notion of an important symbol can also be extended to terminal candidates in the *t_symbols* sets. Once again, consider the state q_i of Figure 5. This state contains a single terminal action on the symbol *id*, but, since *id* appears only in the item $P \rightarrow \cdot id$, it is not an important candidate in q_i. The removal of unimportant terminals improves the time performance of the primary recovery and saves space. However, it may suppress some opportunities for merging and misspelling corrections.

In [13], an algorithm is presented that can be used to further reduce the space used by *t_symbols* and *nt_symbols*.

3.4 Scope Recovery

One of the most common errors committed by programmers is the omission of block closers such as an *end* statement or a right parenthesis. Such

```
if_stmt → IF cond THEN
                   st_list  elsif_list  opt_else
               END IF ;
st_list → stmt | st_list stmt
elsif_list → ε | elsif_list ELSIF cond THEN st_list
opt_else → ε | ELSE st_list
stmt → ... | if_stmt | ...
```

Figure 6: BNF rule for Ada **if** statement

an error is referred to as a *scope error*. Scope errors are common because the structures requiring block closers are usually recursive structures that, in practice, are specified in a nested fashion. In such a case, a matching block closer must accompany each structure in the nest. For example, if a user specifies an expression that is missing a single right parenthesis, primary recovery can successfully insert that symbol. However, if two or more right parenthesis are missing, neither primary nor secondary recovery can successfully repair such an error. Similarly, consider the BNF rule for an Ada *if statement* in Figure 6 [9]: If an Ada *if statement* is specified without the "END IF ;" closer, neither of the two recovery techniques mentioned so far can effectively repair this error. The repair that is necessary for this kind of error is the insertion of a sequence of symbols, called *multiple symbol insertion.*

Scope recovery was first introduced by Burke and Fisher [11]. Their technique requires that each closing sequence be supplied by the user as a list of terminal symbols. Scope recovery is attempted by checking whether or not the insertion of a combination of these closing sequences can allow the parser to recover.

By contrast, the scope recovery technique used in this method is based on the identification of one or more recursively defined rules that are incompletely specified, and insertion of the appropriate closing symbols to complete these phrases. All necessary scope information required by this method is precomputed automatically from the input grammar. In addition, the method is based on a pattern match with complete rules rather than just the insertion of closing sequences of terminal symbols. As a result, the diagnosis of scope errors is more accurate in that it identifies whole structures that are incompletely specified instead of just the missing sequence of closing terminals.

3.4.1 Scope Information

Definition 3.1 *A rule* $A \rightarrow \alpha B\beta$ *is a* **scoped rule** *if* $\alpha \neq \varepsilon$, $B \Rightarrow^* \gamma A\delta$, *for some arbitrary string* γ *and* δ, *and* $\beta \not\Rightarrow^* \epsilon$.

In the example of Figure 5, the rule $P \rightarrow (E)$ is a scoped rule since P can be derived from E. The if_stmt rule of Figure 6 is also a scoped rule since each of the bold symbols following THEN in that rule can recursively derive a string containing the symbol if_stmt. A *scope* can be derived from a scoped rule for each recursive symbol in the right-hand side of the scoped rule.

A scope is a quintuple (π, σ, a, A, Q) where π and σ are strings of symbols called *scope prefix* and *scope suffix*, respectively, a is a terminal symbol called the *scope lookahead*, A is a nonterminal symbol called the *left-hand side* and Q is a set of states. The scope prefix is the prefix of a *suitable string* derivable from the scoped rule in question. It is used to determine whether or not a recovery by the associated scope is applicable; i.e., at run time, a repair by a given scope is considered only if this initial substring of the suitable string can be successfully derived before the error token causes an error action. The scope suffix is the suffix (of the suitable string) that follows the scope prefix. When diagnosing a scope error, the user is advised to insert the symbols of the scope suffix into the input stream to complete the specification of the scoped rule. The scope lookahead symbol (string, if the grammar is LR(k)) is a terminal symbol (string) that may immediately follow the prefix in a legal input. The left-hand side of the scope is the nonterminal on the left of the scoped rule. The set Q contains the states of the LR(k) automaton in which the left-hand side can be introduced through closure.

Given a scoped rule $A \rightarrow \alpha B\beta$, the scope information related to B is computed as follows. Since $\beta \not\Rightarrow^* \epsilon$, there exists a string $\psi X\phi$ such that $\beta \Rightarrow^* \psi X\phi$, $\psi \Rightarrow^* \epsilon$, and $X \Rightarrow^*_{rm} a\omega$. Let $\alpha B\psi X\phi$ be the suitable string mentioned above, then a valid *scope* for the above rule is $(\alpha B\psi, X\phi, a, A, Q)$, where Q is the set of states in the LR automaton containing a transition on A.

As an example, consider the if_stmt rule of Figure 6 and the scope induced by the nonterminal **st_list** in its right-hand side. To put it in the form $A \rightarrow \alpha B\beta$, let B be the symbol "**st_list**". It follows that α is the string "IF cond THEN", and β is the string "**elsif_list opt_else** END IF ;". Let ψ be the string "**elsif_list opt_else**" and let X be the symbol "END". One observes that β is exactly in

```
# Let scope_seq be a global output variable.
# Initially, scope_seq= [ ] and scope_trial is
# invoked with the sequence $q_1, \ldots, q_m$. The input
# sequence $t_1, \ldots, t_n$ is assumed to be global.
proc scope_trial(stack);
{ for each scope $(\pi_i, \sigma_i, a_i, A_i, Q_i)$ do
  { sstk := stack;
    act := lookahead_action(sstk, $a_i$, pos)
    if act $\neq$ error then
    { sstk[pos+1..] := tstk[pos+1..];
      top := #sstk - $|\pi_i|$;
      if top > 0 then
      { pref := [in_sym[sstk[j]] : j in top+1..#sstk];
        if pref = $\pi_i$ and sstk[top] $\in Q_i$ then
        { do
          { top := top - rhs[act] + 1;
            act := GOTO(sstk[top], lhs[act]);
          } while act is a goto-reduce action
          sstk[top+1 ..] := [act];
          if prschck(sstk, $t_1, \ldots, t_n$) > min_dist then
          { scope_seq := [i];
            return;
          }
          else
          { scope_trial(sstk);
            if scope_seq $\neq$ [ ] then
              scope_seq := scope_seq + [i];
            return;
          }
        }
      }
    }
  }
}
```

Figure 7: *scope_trial* procedure

the desired form $\psi X \phi$. Thus, assuming the set of transition states Q is available, the scope induced by **st_list** for the rule if_stmt is:

(IF cond THEN **st_list elsif_list opt_else**, END IF ;, END, if_stmt, Q)

The other recursive symbols in if_stmt: **elsif_list** and **opt_else** induce exactly the same scope as **st_list**, since they are both *nullable*.

3.4.2 Scope Error Detection

Given an error configuration:

$$q_1, \ldots, q_m \quad | \quad t_1, \ldots, t_n$$

and a set of scopes:

$$\{(\pi_1, \sigma_1, a_1, A_1, Q_1), \; \ldots, \; (\pi_l, \sigma_l, a_l, A_l, Q_l)\},$$

the applicability of scope recovery to this configuration is determined as follows. For each scope $(\pi_i, \sigma_i, a_i, A_i, Q_i)$, a three-step test is performed:

step 1: The *lookahead_action* function is invoked with a_i as the current token to check if a_i is a valid lookahead symbol for the viable prefix. As a side-effect, this function updates the state stack configuration (using a temporary stack) to reflect all reduce actions, including empty reductions, induced by a_i. If the action returned by *lookaheadaction* is the error action then the whole test fails. Otherwise, step 2 is executed.

step 2: A pattern match is made between the prefix π_i and the topmost $|\pi_i|$ symbols of the viable prefix, i.e., the string obtained from the concatenation of the in_symbols of the states: $q_{m-|\pi_i|+1} \cdots q_m$. Again, if this test fails, the whole test fails. Otherwise the final step is executed.

step 3: If $q_{m-|\pi_i|} \in Q_i$ then the test is successful. Otherwise, the test fails.

If the three-step test is successful, then a parse check is performed on the configuration: $q_1, \ldots, q_{m-|\pi_i|}, q_A \mid t_1, \ldots t_n$, where q_A is the successor state of q_m and A[1]. If the *parse_check* function can parse at least *min_distance* symbols, the scope recovery is successful. Otherwise, it is invoked recursively with the new configuration above and the process is repeated until scope recovery either succeeds, or there are no more possibilities to try.

When scope recovery is successful, the sequence of scopes that resulted in the successful recovery must be saved for the issuance of an accurate diagnostic.

Figure 7 shows a complete implementation of the scope error detection algorithm. The algorithm mirrors the preceding discussion in a straightforward manner. The emphasis in writing the code was on the clarity of the exposition rather than efficiency.

[1] If the action in q_m on A is a goto-reduce, the parser is simulated through the whole sequence of goto-reduce actions that follow, until a goto action is encountered. This final goto is executed and the resulting state sequence is used instead. Note that these actions do not consume any input symbol.

4 Recovery Phases

This section describes how the different repair strategies discussed in the previous sections are incorporated into the unified two-phase scheme of this method. At the global level, the effectiveness of a recovery trial is measured based on two criteria:

- the number of symbols that must be deleted if the repair in question is applied
- the *parse_check* distance of the recovery

The primary phase recovery which includes all recovery trials that are based on at most a single input token modification is attempted first. If a successful primary phase recovery is found that cannot be beaten by any other recovery in terms of the criteria above, it is accepted. If such a primary phase recovery is not found, secondary phase recovery is attempted. If a successful secondary phase recovery is found, then it is accepted. Otherwise, the error recovery gets into a form of panic mode, where the current input buffer is flushed, new input tokens are read in and secondary phase recovery is attempted again. This process is repeated until either a successful secondary recovery is obtained or the end of the input stream is reached.

When a recovery is accepted, the following actions are taken: a diagnosis is issued, the repair is applied and the error recovery procedure returns successfully.

The diagnosis of a primary recovery is straightforward. To diagnose a secondary deletion, the user is advised to delete the symbols in the error phrase in question. Similarly, for a secondary substitution, the relevant reduction goal is suggested as a replacement for the error phrase. The location of an error phrase starts from the location associated with the recovery state to the location of the last token in the error phrase. To diagnose a scope recovery, the location of *prevtok* is used to indicate where the symbols of the scope suffix in question should be inserted.

A repair is applied by resetting the components of the main configuration (*buffer* and *stk*). The resetting of the input buffer simply involves the insertion of some symbols into the buffer, the reading of new input tokens into the buffer, or the replacement of some buffer elements. The resetting of the stack is more complicated. For a primary recovery, one only needs to choose the stack on which the recovery was successful. For a secondary recovery, all states following the recovery state are removed from the stack. For a scope recovery, the sequence of states on top of the stack that corresponds to the prefix of the scope is removed and the repair proceeds as if the error was a simple insertion of the left-hand side of the scope.

4.0.3 Primary Phase

In the primary phase, error recovery is applied on each available configuration, starting with *nstk*, proceeding with *stk* and finally processing *pstk*. For each configuration, scope recovery is attempted first followed by primary recovery. The same criteria used in choosing a primary recovery is used in the primary phase. The misspelling index of a scope recovery trial is set to 1.0. Thus, for a given configuration, a successful scope recovery always has priority over a primary recovery trial that yields the same parse_check distance.

If a successful recovery is obtained from the primary phase and its stack configuration is *nstk* or *stk*, the recovery trial is evaluated against certain secondary recovery trials on the stack configuration in question before being accepted. These recovery trials are the ones whose repair actions would have as little impact on the recovery configuration as a primary recovery. They are misplacement recovery trials and scope recovery trials that require the deletion of one input token. The idea is to ensure that none of these borderline recoveries can be more effective than the best primary phase recovery.

4.0.4 Secondary Phase

In the secondary phase, secondary error recovery is applied first on *nstk* if it is available and then on *stk*. If a successful secondary recovery is obtained, a check is made to see if the error can be better repaired by the closing of some scopes followed by less radical surgery. Consider the following Pascal example:

```
if count[listdata[sub] := 0 then
    x := (( 3 ]];
```

In the first line, the user is missing a closing "]" and the assignment operator ":=" is used instead of a relational operator. This error is detected on the symbol ":=". In the second line, the user used the wrong closing symbols in an expression and the error is detected on the first "]". Nothing short of a secondary deletion of the sequence "[listdata[sub] := 0" in the first instance and a secondary substitution of "expression" for the sequence "((3]]" would successfully repair these errors. However, it is not difficult to see that they can be repaired more accurately, using scope recovery by proceeding as follows.

Before accepting a secondary recovery based on an error phrase $\beta|x$, a scope recovery check is performed on the recovery configuration, followed by the deletion of up to $|x|$ tokens in the right context. If the scope recovery is successful, then its associated repair actions are applied without the subsequent deletion and the secondary phase returns successfully. The parser fails right away and once again invokes the error recovery procedure. On this next round, primary and secondary phase recovery are attempted again. This subsequent attempt will at best fix the remaining input or at worst delete a string up to the length x from the input. In the example above, the missing "]" is inserted and "relational_operator" is substituted for ":=" in the first line. In the second line, two closing ")" are inserted, followed by a deletion of the pair "]]" (See figure 2).

5 Implementation

The error recovery method described in this paper has been successfully implemented. An LALR(k) parser generator was modified to produce the extra tables required: *t_symbols*, *nt_symbols* and the scopes. The method can be used with any LR(k) application. However, programming languages were used in our examples because such applications are the best illustrations of the problems one is likely to encounter. Parsers were built for Ada and Pascal and tested on the Ada examples of [11] and the Pascal examples of [6].

Penello and DeRemer [4] proposed that the quality of a repair be rated "excellent" if it repaired the test as a human reader would have, "good" if not but it still resulted in a reasonable program and no spurious errors, and "poor" if it resulted in one or more spurious errors. Based on these categories, the performance of this method on the test set of [6] was 85.9% excellent, 14.1% good and 0.0% poor. In fact, most of the "good" recoveries resulted from errors whose repair required some kind of semantic judgement.

The time performance of this method is excellent, usually requiring less than 50 milliseconds per error on a 16 MHz PS/2 model 80.

6 Conclusion

This paper described a new practical LR(k) error diagnosis and recovery method which improves upon the current state-of-the-art in some significant ways. Specifically,

- a new deferred driver is introduced which always detects an error at the earliest possible point;
- the primary recovery is generalized to process both terminal and nonterminal symbols;
- the secondary recovery is an efficient (and completely automatic) generalization of the error production method;
- techniques are presented for optimizing error recovery candidates;
- a new automatic method for scope recovery is presented.

Moreover, this method is completely language- and machine-independent and more efficient than other known methods.

7 Acknowledgements

The author wishes to thank the following people for many helpful suggestions and their encouragement throughout the development of this work: Michael Burke, Ron Cytron, Gerald Fisher, Laurent Pautet, Matthew Smosna. The author is especially thankful to Fran Allen and Ed Schonberg for their advice and support.

References

[1] F. L. DeRemer
Practical Translators for LR(k) Languages.
Ph.D. dissertation, MIT, Cambridge, Mass., 1969

[2] F. L. DeRemer: Simple LR(k) Grammars.
Comm. ACM 14, 7, 453-460 July 1971

[3] Alfred V. Aho, Jeffrey D. Ullman
The Theory of Parsing, Translation, and Compiling
Volume I & II, Prentice Hall, Inc 1972

[4] Penello, T. J., and DeRemer, F. L.
A forward move algorithm for LR error recovery.
ACM Symposium on Principles of Programming Languages (Jan. 23-25, 1978, Tuscon), pp. 241-254

[5] Ripley, G. D., and Druseikis, F.C.
A statistical analysis of syntax errors
Journal of Computer Languages 3,4 (1978) (227-240)

[6] Ripley, D. J.: Pascal Syntax Errors Data Base
RCA Laboratories, Princeton, N.J., Apr 1979

[7] S. L. Graham, C. B. Haley, W. N. Joy
Practical LR Error Recovery
SIGPLAN 79 Symposium on Compiler Construction
(August 6-10, 1979, Denver) ACM, NY, pp 168-175.

[8] Seppo Sippu, Eljas Soisalon-Soininen
A Syntax-Error-Handling Technique and Its Experimental Analysis
ACM Transactions on Programming Languages and Systems, Vol. 5, No. 4, October 1983, Pages 656-679

[9] Ref. Manual for the ADA Programming Language
ANSI/Mil-STD-1815A-1983, U.S. Dept. of Defense.

[10] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman
Compilers: Principles, Techniques and Tools
Addison Wesley Publishing Company, 1986

[11] Michael Burke, Gerald A. Fisher
A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery
ACM Transactions on Programming Languages and Systems, Vol. 9, No. 2, April 1987, Pages 164-197

[12] Nigel P. Chapman: LR Parsing: Theory and Practice
Cambridge University Press, 1987

[13] Philippe Charles, Laurent Pautet
Efficient Representation of LR Error Recovery tables
Unpublished paper, 1989

[14] Jüergen Uhl: Private communications