Chapter 4

LALR(k) Parser and Error Recovery

4.1 Overview

4.1.1 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 state_stack, and an input buffer containing a fixed number of input symbols. These symbols include the current token or *lookahead*, denoted *curtok* and the token immediately preceding the current token (last token processed), denoted *prevtok*. The remaining tokens in the buffer are 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 in 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 corresponding to a viable prefix. 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 the current token, etc. The symbol t_0 which is not shown above represents the class of previous token.

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.

4.1.2 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:

- *Simple recovery.* A simple recovery (also called *first level recovery* [21]) is 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 [21][37] have used a more restricted form of simple recovery involving only terminal symbols as repair candidates.

- *Phrase-level recovery.* In phrase-level recovery, the error procedure tries to recover by either deleting as small a sequence of tokens as possible in the vicinity of the error token or replacing such a sequence with a nonterminal symbol. This technique can be viewed as an *automatic* generalization of the *error productions* method described in [9].

- *Scope recovery.* A scope is a syntactically nested structure such as a parenthesized expression, 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 incomplete scopes.

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. Repairs that are attempted in the primary phase include the different kinds of simple recoveries, scope recoveries that require the deletion of no or one input symbol and phrase-level recoveries that do not require any deletion of input symbols. Figure 4.1 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 remaining input (right context) are attempted. Figure 4.2 shows some examples of secondary phase recoveries.

4.1.3 Error Detection

A canonical LR(k) parser has the capability of detecting an error at the earliest possible point. However, as mentioned earlier, canonical LR(k) parsers are seldom used because of their size. Instead, variants such as LALR(k) and SLR(k) (usually $k = 1$) 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 earliest possible point. This problem is usually compounded by the use of the space-saving technique known as *default reduction* described in the previous chapter. Another undesirable side effect of default reductions is that when they are used, it is no longer possible to compute, from the parsing table, the set of terminal symbols on which valid

```
     1. program TEST(INPUT, OUTPUT);
     2.     var X,Y: array[] of integer;
                           ^
***Error: index_type_list expected after this token
     3. 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 phrase started at line 5, column 21

     8. end.
```

Figure 4.1: Primary phase recoveries

```
     1. program P(INPUT,OUTPUT);
     2.     procedure FACTORIAL(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 4.2: Secondary phase recoveries

---

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.

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 [37], 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 negligible 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 and optimization will be discussed in chapter 5.

The remainder of this chapter is organized as follows:

- detailed description of the new driver
- presentation of various recovery techniques
- discussion of how to apply these recovery techniques and how to issue accurate diagnostics.

4.2 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 consists of a single final item. Therefore, such a state contains only reduce actions by the rule from which the final item in question is derived. If a representation of the parsing tables with default action is used,

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*, which comprises a read transition followed by a reduction. If q is an LR(0) reduce state consisting of a single final item $[A \rightarrow \alpha X \cdot]$, in order to remove q from the automaton, the parsing tables can be modified as follows. For all states p such that $\mathrm{GOTO}_0(p, X) = q$, change the action in p on X to a read-reduce action by the rule $A \rightarrow \alpha X$. State q is now inaccessible and can be discarded. 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.

When the parser encounters a shift-reduce action by a rule $A \rightarrow \alpha t_1$ in a configuration $q_1, q_2, \ldots, q_m \mid t_1, t_2, \ldots, t_n$, it pops $|\alpha|$ elements from the state sequence, reads the input symbol t_1 and executes the appropriate nonterminal action on the state $q_{m-|\alpha|}$ and A. A goto-reduce action is processed in a similar fashion except that no input symbol is read in. From now on, it is assumed that all LR(0) reduce states have been removed from a given parsing table and replaced with read-reduce actions.

As there is only one kind of action that can be executed in an LR(0) reduce state, the removal of these states from an LR automaton does not cause *premature* reductions. Moreover, observe that when the parser computes a read-reduce action, that action is followed by a sequence of zero or more goto-reduce actions, and finally, by a goto action. All these actions may also be executed without deferral, by the same argument.

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 a final 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, copy 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 computed on the lookahead symbol. If the first non-reduce action computed on the lookahead is valid, the temporary 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 4.3.

Starting with a given configuration, the *lookahead_action* function, in essence, checks whether or not it is possible to advance the parse past the current token, and if so, it returns the first non-reduce action induced by the current token. As a side effect, it also computes the index position (pos) of the topmost element in the state stack sequence that is still useful after all the actions induced by the current token are executed. All new

```
-- Assume 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.

 1. function lookahead_action(stack, tok, pos);
 2.     temp_stack := [ ];
 3.     pos := #stack;
 4.     top := pos - 1;
 5.     act := ACTION(stack(pos), tok);
 6.     while act is a reduce action loop
 7.         until act is not a goto-reduce action loop
 8.             top := top - RHS(act) + 1;
 9.             if top > pos then
10.                 s := temp_stack(top);
11.             else s := stack(top);
12.             end if;
13.             act := GOTO(s, LHS(act));
14.         end loop;
15.         temp_stack(top+1) := act;
16.         act := ACTION(act, tok);
17.         pos := min(pos, top);
18.     end loop
19.     return act;
20. end lookahead_action;
```

Figure 4.3: *lookahead_action* function

```
act := q0;
state_stack := [ ];
location_stack := [ ];
while act is not the accept action loop
    state_stack := state_stack + [act];
    location_stack(#state_stack) := curtok.location;
    act := lookahead_action(stack, t1, pos);        - - recall that t1 = curtok.class
    if act ≠ error action then
        state_stack(pos+1..) := temp_stack(pos+1..);
        location_stack(pos+1..) := [curtok.location : i in [pos+1..#state.stack]];
        if act is a shift-reduce action then
            top := #state_stack;
            while act is a goto-reduce action loop
                top := top - RHS(act) + 1;
                act := GOTO(state_stack(top), LHS(act));
            end loop;
            stack(top+1..) := [ ];
        end if;
        get next token;
    else
        error_recovery();
    end if;
end loop;
```

Figure 4.4: Driver with one deferred token

---

stack information above that index position is stored in the temporary stack in its relative position.

Lemma 4.2.1 *Let* stack_top=top+1 *prior to entering the inner loop of lookaheadaction, when the inner loop is exited,* top $\leq$ stack_top.

Proof: The first iteration of the loop processes an initial reduce action. At the end of the first iteration, the variable top is either increased by 1 (in the case of an empty production, RHS(act) = 0), left unchanged (in the case of a unit production), or decreased. Each subsequent iteration of the loop is induced by a goto-reduce action processing a non-empty rule which either leaves the value of top unchanged or decreases it.

When the inner loop is exited, the transition state associated with the goto action that caused the exit is stored in the temporary stack at position top+1, the new upper bound of the stack. If top is a new upper bound for the initial state stack, pos is updated accordingly. When the outer loop is exited, the action that caused the exit is returned.

Assume that the starting state of an LALR(k) parser is denoted by q_0 and that a temporary stack denoted temp_stack is globally available. The algorithm of Figure 4.4 depicts the body of a driver with actions deferred on one token symbol. Initially, the parser is in configuration $q_0 \mid \omega$ where q_0 is the start state, and ω is the whole input string. Starting with the initial configuration, the idea is to advance through the input stream one token at a time,

```
state_stack := [q_0];
while true loop
    ppos := 0;    previous_stack := [ ];
    npos := 0;    next_stack := [ ];
    location_stack(#state_stack) := curtok.location;
    temp_stack := state_stack;
    act := lookahead_action(temp_stack, t_1, pos);
    while act ≠ error and act ≠ accept loop
        next_stack(npos+1..) := temp_stack(npos+1..);
        location_stack(pos+1..) :=
            [curtok.location : i in [pos+1..#next_stack]];
        if act is a shift-reduce action then
            top := #next_stack;
            until act is not a goto-reduce action loop
                top := top - RHS(act) + 1;
                act := GOTO(next_stack(top), LHS(act));
            end loop;
            next_stack(top+1..) := [act];
            pos := min(pos, top);
        end if
        act := lookahead_action(next_stack, t_2, npos);
        if act ≠ error then
            get next token;
            previous_stack(ppos+1..) := state_stack(ppos+1..);
            ppos := pos;
            state_stack(pos+1..) := next_stack(pos+1..);
            pos := npos;
        end if;
    end loop;
    if act =accept then
        return;
    end if
    error_recovery();
end loop;
```

Figure 4.5: Driver with 3 deferred tokens

executing all actions induced by the current input token at each step. Thus, the function *lookahead_action* is invoked at each step to check whether or not it is possible to advance. If so, the state stack is updated by replacing all of its topmost elements from pos+1 to #temp_stack by the corresponding temp_stack elements. Usually, pos+1=#temp_stack unless a non-empty sequence of empty reductions, each followed immediately by a goto action were executed prior to exiting the outer loop of the *lookahead_action* function. In such a case, #temp_stack exceeds pos+1 by the number of such empty reductions that were executed. If the valid action returned by the *lookahead_action* function is a shift-reduce action, then it and all its associated goto-reduce actions are processed. When the parser can advance successfully, the next token is read in and the process is repeated on the new configuration. If, on the other hand, the error action was returned by the *lookahead_action* function, the state stack is not updated and the error recovery routine is invoked instead.

It is not hard to see how this driver routine can be adapted 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: previous_stack which captures the configuration of the parser prior to processing any action induced by *prevtok*, state_stack which captures the configuration prior to processing actions induced by *curtok*, and next_stack 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 4.5 shows the body of a driver routine with actions deferred on three input symbols. This deferral usually increases the time requirement of the parser by 25% or less.

4.3 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 [37].

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. The same approach taken in implementing the *lookahead_action* function can be extended to implement the *parse_check* function; i.e., a temporary stack can be used to keep track of all state information related to transitions induced by the lookahead tokens in the buffer, thus, avoiding copying the state stack or destroying it.

In the remainder of this section, the implementation of the three recovery strategies mentioned earlier are described in detail.

4.3.1 Simple 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 simple recovery finds the best possible *simple repair* (if any) for that configuration.

The selection of a best simple 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 simple 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.

Simple 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 5.2.1.

As the trials are performed, the simple 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 $t_symbols(q_e) \subseteq t_symbols(q_m)$.) This ordering is not crucial but its benefits 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 14. in Figure 4.3:

```
error_state := act;
```

4.3.1.1 Misspelling 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. This ensures that if a merge trial yields a successful recovery, any subsequent recovery that is applicable in the given context will not be chosen over the merge recovery unless it yields a longer *parse_check* distance. Consider, the following example:

```
if x >  = y then
    x :  = z;
```

In this case, it is most likely that the programmer inadvertently inserted a space character between "`>`" and "`=`" in the first line, and between "`:`" and "`=`" in the second line. Both errors can be successfully repaired with merging. However, from a syntactic point of view, the first error is just as easily repaired by deleting the "`>`" symbol (or the "`=`"); but, since the merging trial is perfomed first and, in addition, it yields a higher misspelling index, it is chosen over the deletion.

As mentioned earlier, a misspelling function is invoked to calculate the misspelling index for a simple substitution. For all other recoveries, the misspelling index is set to 0.0 unless the candidate in question was identified as a special *end-of-line* terminal and the correction is to be made at the end of a line. In that case, the misspelling index is set to 1.0. (A boolean attribute is associated with each input token indicating whether or not it is located at the end of a line in the input). In Pascal, the end-of-line terminal is ";". Consider the following example:

```
x := y
p(x);
```

The error token in this incorrect fragment is `p`. Syntactically, any arithmetic operator, say "`+`", inserted after the symbol `y` would repair this error. However, it is clear from the context that since `y` is at the end of the line, "`;`" is a better candidate. Setting the misspelling index to 1.0 when inserting the end-of-line terminal at the end of a line gives it precedence over other insertion candidates.

4.3.1.2 Misspelling Function

In designing a misspelling function for syntactic error recovery, special attention should be paid to the kinds of errors that a programmer is likely to make. For example, the use of abbreviations, such as `int` instead of `integer` or `proc` instead of `procedure`, is

common. Often, words that start with a similar prefix are substituted for each other: e.g., `procedure` for `program`. The algorithm used should be sensitive enough to identify such a pair of words as a misspelling error (though with a low probability); but, it should also be able to reject a pair of words such as `return` and `turner` that are permutations of the same set of characters but unlikely to be incorrect misspellings of each other.

The misspelling algorithm used in this method is an adaptation of an algorithm proposed by Jüergen Uhl [43]. In that approach, three kinds of misspelling errors are identified: transposition of two adjacent characters, mismatch of corresponding characters and omission (insertion) of a character. The idea is to scan the two strings simultaneously and compute the following information as they are being traversed.

- the length of the initial prefix of the two strings that matches (prefix_length)
- the total number of characters that match (match_count)
- the number of errors found (error_count)

With this information, the probability that a string s_1 is a misspelling of a string s_2 is calculated as follows. Let the error threshold be the length of the shortest string divided by six plus one; i.e., at least one error is allowed plus an additional error for each six characters in the shortest string. If the number of errors detected by the pattern match (error_count) is less than or equal to the error threshold then match_count represents the result of the pattern match (pattern_count); otherwise, only the number of initial characters that matched (prefix_length) are considered. The final result is obtained by dividing pattern_count by the length of the longest string plus error_count. Figure 4.6 shows a complete implementation of this algorithm.

Certain static characters are also likely to be substituted for others. For example,

$$
\begin{array}{ccc}
; & \leftrightarrow & , \\
; & \leftrightarrow & : \\
. & \leftrightarrow & , \\
' & \leftrightarrow & " \\
\end{array}
$$

In this method, the characters of each of the above pairs are considered to be 0.33% likely to be a misspelling of each other.

4.3.2 Phrase-Level Recovery

Phrase-level recovery 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:

$$q_1, \ldots, q_m \quad | \quad t_1, \ldots, t_n \tag{4.1}$$

is an error configuration, then a substring

$$q_{i+1}, \ldots, q_m \quad | \quad t_1, \ldots, t_{j-1} \tag{4.2}$$

```
--  The function misspell takes two arguments: s1 and
--  s2 which are character strings of arbitrary length.

function misspell(s1, s2);
    i := 0;
    j := 0;
    match_count := 0;
    prefix_length := 0;
    error_count := 0;
    while ((i < #s1) and (j < #s2)) loop
        if s1(i) = s2(j) then    -- matched characters?
            match_count := match_count + 1;
            i := i + 1;
            j := j + 1;
            if error_count = 0 then    -- prefix character?
                prefix_length := prefix_length + 1;
            end if;
        elseif s1(i+1) = s2(j) and s1(i) = s2(j+1) then    -- transposition?
            match_count := match_count + 2;
            i := i + 2;
            j := j + 2;
            error_count := error_count + 1;
        elseif s1[i+1] = s2[j+1] then    -- mismatch?
            i := i + 1;
            j := j + 1;
            error_count := error_count + 1;
        else -- definitely a deletion!
            if (#s1-i) > (#s2-j) then    -- suffix of s1 longer?
                i := i + 1;
            elseif (#s2-j) > (#s1-i) then    -- suffix of s2 longer?
                j := j + 1;
            else
                i := i + 1;
                j := j + 1;
            end if;
            error_count := error_count + 1;
        end if;
    end loop;
    if (i < #s1) or (j < #s2) then
        error_count := error_count + 1;
    end if;
    if error_count <= (min(#s1, #s2) / 6 + 1) then    -- check error threshold
         pattern_count := prefix_length;
    else pattern_count := match_count;
    end if;
    return float(pattern_count) / (max(#s1, #s2) + error_count);
end misspell;
```

Figure 4.6: Misspelling function

$1 \leq i \leq m$, $1 \leq j \leq n$, 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.

In [27], the authors present a detailed discussion of different strategies that are used for selecting error phrases, and the advantages and disadvantages of each strategy. In general, the search for the error phrase begins with the shortest possible error phrase; that is, with the one consisting only of the vertical bar, and proceeds with larger and larger segments of the configuration. A search order may be used that consumes state stacks faster than input symbols or consumes input symbols faster than state stacks or a more balanced scheme may be used.

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 (4.2) above. The case of the empty error phrase is considered during simple recovery as a nonterminal insertion. Similarly, the case of an error phrase $\varepsilon|t_1$ being deleted or replaced by a nonterminal candidate is processed by a simple deletion or substitution. Next, priority is given to a successful phrase-level 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 $\beta \neq \varepsilon$. This kind of error is called a *misplacement error*, and β is called a *misplaced phrase*. The following Pascal example 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 (4.1). An initial attempt is made to recover by parse checking the resulting configuration. This action, called *phrase 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

```
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 4.7: BNF rule for Ada **if** statement

phrase substitution. This process continues until a successful recovery is found or all the possibilities are exhausted.

In phrase-level 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 phrase-level 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 phrase 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.

4.3.3 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 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, simple recovery can successfully insert that symbol. However, if two or more right parenthesis are missing, neither simple nor phrase-level recovery can successfully repair such an error. Similarly, consider the BNF rule for an Ada *if-statement* in Figure 4.7 [28]: 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 [37]. 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, because the method is based on a pattern match with complete rules rather than just the insertion of closing sequences of terminal symbols, 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.

4.3.3.1 Scope Information

Definition 4.3.1 *A rule $A \rightarrow \alpha B \beta$ is a* **scoped rule** *if and only if the following conditions are satisfied:*

1. $B \Rightarrow^* \gamma A \delta$, *for some arbitrary strings* γ *and* δ
2. $\beta \not\Rightarrow^* \varepsilon$
3. $\alpha \not\Rightarrow \varepsilon$ *or* $B \not\Rightarrow^*_{lm} A\psi$, *for some string* ψ
4. $\alpha \neq \epsilon$ *or* ($\exists$ *a viable prefix* $\phi A \mid$ *either* ϕB *is not a viable prefix or* $\nexists C \in N$ *such that* $\phi C \Rightarrow^*_{rm} \phi A$ *and* $\phi C \Rightarrow^+_{rm} \phi B$).

Condition 1 simply states that for a rule to be a *scoped rule* it must be recursive. Condition 2 guarantees that the closing sequence following the recursive symbol in the right-hand side of the rule is not nullable. These two conditions are consistent with the intuitive notion of a scope presented earlier. By contrast, the purpose of conditions 3 and 4 is to exclude some unnecessary cases. For example, consider the following left-recursive rule:

$$\mathsf{List} \rightarrow \mathsf{List}\ ,\ \mathsf{atom}$$

Condition 3 eliminates such a rule from consideration since the initial prefix preceding the recursive nonterminal List in the right-hand side is empty and $\mathsf{List} \Rightarrow^*_{lm} \mathsf{List}$. The rules affected by condition 4 are more complex. Consider the following grammar:

$$\begin{array}{lcl} \mathsf{List} & \rightarrow & \mathsf{Sublist} \\ \mathsf{List} & \rightarrow & \mathsf{Sublist\ List} \\ \mathsf{Sublist} & \rightarrow & \mathsf{atom} \\ \mathsf{Sublist} & \rightarrow & \mathsf{(\ List\ )} \end{array}$$

Since $\mathsf{Sublist} \Rightarrow \mathsf{(\ List\ )}$, the right-recursive rule $\mathsf{List} \rightarrow \mathsf{Sublist\ List}$ satisfies conditions 1, 2 and 3. However, it does not satisfy condition 4, because $\mathsf{List} \Rightarrow^+_{rm} \mathsf{Sublist}$. In essence, condition 4 eliminates from consideration all rules of the form $A \rightarrow B\beta$ such that whenever A is introduced through closure, it is introduced by some nonterminal C (where C may be the symbol A itself, as in the above example) which can produce both A and B by a sequence of right-most derivations.

In the above example, the rule $\mathsf{Sublist} \rightarrow \mathsf{(\ List\ )}$ is a scoped rule. In the example of Figure 5.5, the rule $P \rightarrow (E)$ is a scoped rule since P can be derived from E. The

if_stmt rule of Figure 4.7 is also a scoped rule since each of the symbols: st_list, elsif_list and opt_else 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), $k > 0$) 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 $\beta \not\Rightarrow^* \varepsilon$, there exists a string $\psi X \phi$ such that $\beta \Rightarrow^* \psi X \phi$, $\psi \Rightarrow^* \varepsilon$, and $X \Rightarrow^*_{rm} a\omega$, for some string ω. Let $\alpha B \psi X \phi$ be the suitable string mentioned above, then a valid *scope* for the rule $A \rightarrow \alpha B \beta$ 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 4.7 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 the desired form $\psi X \phi$. Thus, assuming the set of transition states Q is available, the scope induced by st_list for the if_stmt rule is:

(**if** cond **then** st_list elsif_list opt_else, **end if**;, **end**, if_stmt, Q)

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

4.3.3.2 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, assuming, once again, that m denotes the upper bound subscript of the updated state sequence, 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: $q_{m-|\pi_i|+1}..q_m$. 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 succeeds. 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 4.8 depicts a complete implementation of the scope error detection algorithm. The algorithm mirrors the preceding discussion in a straightforward manner except for the tests in line 12 and the code segment in lines 2 through 5. In line 12, the test for top $>$ 0 ensures that the stack is longer than the scope prefix with which it is being matched and the test for top $<$ min(#stack, pos+1) prevents the algorithm from considering a scope whose prefix would match a null string. This latter test also guarantees that each time a scope is tested and applied it does not extend the length of the stack. The code segment in lines 2 through 5 prevents the algorithm from visiting a given configuration more than once. Since the application of a scope never extends the state sequence it started with, one only needs to keep track of the states that have been entered at each index position of the initial state sequence.

The emphasis in writing the code of Figure 4.8 was on the clarity of the exposition rather than efficiency. In particular, note that stack should not be copied into scope_stack(as in line 7) for each application of a scope. Instead, the attempt to match a scope prefix with the viable prefix should be made using the relevant segment of the state sequence that is in stack and the relevant segment that is in temp_stack. Since the application of a scope never extends the stack, one can simply substitute stack for scope_stack in the recursive call on line 19.

[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.

```
-- Let scope_sequence and state_seen be global variables. scope_trial is invoked with the sequence
-- of states: stack=q_1,...,q_m in the error configuration. The input sequence t_1,...,t_n is assumed
-- to be global. Initially, scope_sequence=[ ] and state_seen=[∅ : i ∈ [1..m]].

 1. proc scope_trial(stack);
 2.     if (q_1 := stack(#stack)) ∈ state_seen(#stack) then
 3.         return;
 4.     end if;
 5.     state_seen(#stack) := state_seen(#stack) ∪ {q_1};
 6.     for each scope (π_i, σ_i, a_i, A_i, Q_i) loop
 7.         scope_stack := stack;
 8.         act := lookahead_action(scope_stack,a_i,pos)
 9.         if act ≠ error then
10.             scope_stack(pos+1..) := temp_stack(pos+1..);
11.             top := #scope_stack - |π_i|;
12.             if top > 0 and top < min(#stack, pos+1) then
13.                 pref := [in_sym(scope_stack(j))  :  j in top+1..#scope_stack];
14.                 if pref = π_i and scope_stack(top) ∈ Q_i then
15.                     until act is a not goto-reduce action loop
16.                         top := top - RHS(act) + 1;
17.                         act := GOTO(scope_stack[top], LHS(act));
18.                     end loop;
19.                     if parse_check(scope_stack(1..top)+[act], t_1,...,t_n) > min_distance then
20.                         scope_sequence := [i];
21.                         return;
22.                     else
23.                         scope_trial(scope_stack);
24.                         if scope_sequence ≠ [ ] then
25.                             scope_sequence := scope_sequence + [i];
26.                         end if;
27.                         return;
28.                     end if;
29.                 end if;
30.             end if;
31.         end if;
32.     end loop;
33. end scope_trial;
```

Figure 4.8: *scope_trial* procedure

4.3.3.3 User-Supplied Scopes

This scope recovery method can be extended to accommodate any kind of multiple insertion of symbols by allowing the user to specify his own "scopes". Scopes are explicitly specified with productions that use the special error symbol, *%error*, as a right-hand side marker to separate the scope prefix from its suffix. These productions are called *user-supplied scoped rules* and the scopes derived from them are called *user-supplied scopes*.

The parser generator upon scanning a rule with *%error* in its right-hand side identifies such a rule as a scope whose prefix consists of the symbols preceding *%error* and whose suffix consists of the symbols following it. The symbol *%error* is used as the lookahead for the scope but it is neither included in the prefix nor the suffix. Therefore, a single rule should never be used to specify more than one user-supplied scope because that will cause the suffix of some scopes to contain the special *%error* symbol.

When specifying scopes, special attention must be paid to the kinds of symbols that are likely to be omitted by a programmer. For example, consider an Ada record_type_definition and the context in which it is used:

```
full_type_declaration  →  type identifier [discriminant_part] is type_definition ;
type_definition        →  ... | record_type_definition | ...
record_type_definition →  record
                              component_list
                          end record
```

Even though the above rule for record_type_definition is terminated by an "**end record**" sequence, it is not a scope, because in Ada, no string containing record_type_definition can be derived from component_list. However, simply duplicating the record_type_definition rule to identify "**end record**" as a scope closer is unlikely to be effective since a programmer who omits this sequence would, most likely, also omit the terminating semicolon. In order to identify the complete closing sequence one must respecify the whole full_type_declaration for a record_type_definition as follows:

```
full_type_declaration  →  type identifier [discriminant_part] is
                          record
                              component_list
                              %error
                          end record ;
```

During normal parsing, user-supplied scoped rules are never reduced except when used for scope recovery because *%error* is only used internally the error recovery procedure and is never recognized as an input symbol by the lexical analyzer. When detecting scope errors and applying scope recovery at run time, user-supplied scopes are treated in the same manner as the automatic ones.

4.4 Recovery Phases

This section describes how the different error recovery strategies discussed in the previous sections are incorporated into the unified two-phase scheme of this method. Error diagnosis, error repair and a suitable data structure for implementing and managing the input stream are also discussed.

It is assumed that the driver with 3 deferred tokens of Figure 4.5 is used. This driver can detect an error either on *curtok* or on the successor of *curtok* which will be denoted *succtok*. When the parsing starts (or restarts after a recovery), if *curtok* is in error, the parser stops immediately and invokes the error recovery routine with a single configuration, namely, the configuration prior to the execution of any action on *curtok*. The state sequence of this error configuration is contained in state_stack. If the parser can successfully process *curtok* but fails on *succtok*, the error recovery routine has access to both the *curtok* configuration and the configuration prior to the execution of any action on *succtok*. The state sequence of this configuration is contained in *next_stack*. If the parser is able to parse at least 2 tokens successfully before an error is detected, the error is detected on *succtok*, and in addition to the *curtok* and *succtok* configurations, the configuration of the parser prior to the execution of any action on *prevtok* is also available. The state sequence of this configuration is contained in *previous_stack*.

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 phrase-level recovery is obtained or the end of the input stream is reached.

When the error recovery procedure finds a successful recovery, it issues a diagnosis for it and repairs the configuration before returning control to the parser.

4.4.1 Primary Phase

In the primary phase, error recovery is applied on each available configuration, starting with next_stack, proceeding with state_stack and finally processing previous_stack.

For each configuration, scope recovery is attempted first followed by simple recovery. The same criteria used in choosing a simple 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 simple recovery trial that yields the same parse_check distance.

If a successful recovery is obtained from the primary phase and its stack configuration is next_stack or state_stack, the recovery trial is evaluated against certain phrase-level recovery trials on the stack configuration in question before being accepted. The phrase-level recovery trials in question are the ones whose repair actions would have as little impact on the recovery configuration as a simple recovery. They are misplacement recovery trials and scope recovery trials that require the deletion of a single input token. The idea is to

ensure that none of these borderline recoveries can be more effective than the best primary phase recovery.

4.4.2 Secondary Phase

In the secondary phase, phrase-level recovery is applied first on the next_stack configuration if it is available and then on the state_stack configuration. Phrase-Level recovery is never attempted on the previous_stack configuration, because it is not always possible to issue an accurate diagnosis for a recovery found in that configuration (see Section 4.4.3) or to properly repair that configuration (see Section 4.4.5).

If a successful phrase-level 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 programmer 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 programmer used two "`]`" instead of "`)`" to close a parenthesized expression and the error is detected on the first "`]`". Nothing short of a phrase deletion of the sequence "`[listdata[sub] := 0`" in the first instance and a phrase 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 phrase-level 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 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 4.2).

4.4.3 Error Diagnosis

In order to accurately diagnose an error, one must identify the location of the tokens that are in error. This is straightforward for a simple recovery since such a recovery involves the modification of one or two input tokens and the location of each token is available. Recall that each state in the state stack is associated with the location of the first token on which an action was executed in that state. Thus, given an error configuration $q_1, \ldots, q_m \mid t_1, \ldots, t_n$, if a successful phrase-level recovery based on an error phrase $q_{i+1}, \ldots, q_m \mid t_1, \ldots, t_{j-1}$ of that configuration is found, the location of the first token of this error phrase is the location associated with the recovery state q_i. If the state sequence associated with the error configuration is state_stack then the symbol t_1 is *curtok*. In that case, if $j = 1$, t_0 is *prevtok*. Similarly, if the state sequence is next_stack and $j = 1$ then

$t_1 = succtok$ and $t_0 = curtok$. Finally, if the state sequence is previous_stack and $j = 1$ then $t_1 = prevtok$ and t_0 is undefined. Therefore, as mentioned earlier, if phrase-level recovery is attempted on the previous_stack configuration, one cannot identify the last symbol of an error phrase that does not contain any input symbol unless the predecessor of the previous token is also kept. When a scope recovery associated with a scope (π, σ, a, A, Q) is applicable on an error configuration like the one above, it can be viewed as a secondary substitution of the nonterminal A for the error phrase $q_{i+1}, \ldots, q_m \mid \varepsilon$, where $|\pi| = m$. That is, the location of the first token of this error phrase is the location associated with q_i and the location of the last token is the location of t_0 as described above. (Note that this implies that the location of the last symbol in a scope error phrase of the previous_stack configuration cannot be accurately determined.)

Once the location of the error token or error phrase has been identified, an error diagnosis message is issued describing the repair. The diagnosis of a simple recovery repair is straightforward except that for an insertion or substitution some preprocessing of the repair token is required before the message is emitted. This will be discussed later. To diagnose a phrase deletion, the user is advised to delete the symbols in the error phrase in question. For a phrase substitution, if the relevant reduction goal is a nullable nonterminal, the diagnosis is treated like a phrase deletion. Otherwise, the reduction goal is suggested as a replacement for the error phrase. To diagnose a scope recovery, the user is advised to insert the symbols of the scope suffix in question after t_0, if the location of t_0 is defined, or before t_1, otherwise. In addition, it is very useful to identify the starting location of the scope which can be computed from the recovery state.

The main goal of an error message is to inform the programmer as to how the input source was modified. However, it is also desirable to issue error messages that give the user an accurate diagnosis of the error. In particular, whenever a symbol X is inserted into the text or it is substituted for an error token (by simple recovery) or for an error phrase (by phrase-level recovery), the symbol reported in the message is the *highest-level symbol* Y that can subsume X. That is, a symbol Y is used instead of X, if assuming ϕ is the viable prefix corresponding to the state sequence of the error configuration and w is the remaining input, $\phi Y w \Rightarrow^*_{rm} \phi X w$ and $\forall B \in N$ such that $B \neq Y$, $\phi B w \not\Rightarrow^+_{rm} \phi Y w$. Consider the following erroneous Pascal statement:

```
if n <= then POWER := else POWER := 2;
```

This statement is missing a subexpression after "<=" and an expression after the first ":=". Since simple recovery is attempted first on terminal symbols and any of the symbols: identifier, NIL, string_literal, integer_literal or real_literal can be reduced to a subexpression, one such symbol will be inserted into the text to repair each of these two errors. Depending on the semantic context, the arbitrary insertion of a symbol can be misleading even though it is syntactically correct. For example, it is clear from the example above that POWER is not a pointer variable. Nonetheless, NIL is a valid candidate that may be inserted after the first ":=". In reporting such an error, if the highest-level symbol associated with the repair candidate is used, simple_expression will be suggested as an insertion candidate after "<=" and expression will be suggested as an insertion candidate after the first ":=". (See Pascal BNF definition in [11].)

The computation of the highest-level symbol is straightforward. Starting in the recovery

```
function highest_symbol(stack, X, t);
    hs := X;
    q := stack(#stack);
    if X ∈ N then
        act := GOTO(q, X);
    else if (act := ACTION(q, X)) is not a shift or shift-reduce action then
        act := lookahead_action(stack, X, pos);
        q := temp_stack(#temp_stack);
    end if;
    temp_stack := [q];
    if act is a shift or goto action then
        temp_stack := temp_stack + [act];
        act := ACTION(act, t);
    end if;
    top := 1;
    while act is a reduce action loop
        until act is not a goto-reduce action loop
            top := top - RHS(act) + 1;
            if top < 1 then
                return hs;
            elseif top = 1 then
                hs := LHS(act);
            end if;
            act := GOTO(temp_stack(top), LHS(act));
        end loop;
        temp_stack(top+1) := act;
        act := ACTION(act, t);
    end loop
    return hs;
end highest_symbol;
```

Figure 4.9: *highest_symbol* function

---

state one simply simulates the steps of the parser until a state q is entered where the candidate X can be shifted. Next, starting with state q as the only state in a stack and using the next input symbol t as lookahead, X is shifted and all reduce actions induced by t and their associated goto actions are applied until the parser wants to shift t or reduce below q. At that point, the last symbol on which a transition was made in q is the highest-level symbol that subsumes X. Figure 4.9 is an implementation of this algorithm.

4.4.4 Error Repair

A repair is applied by resetting the components of the main configuration: the input buffer and state_stack.

The resetting of the input buffer 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.

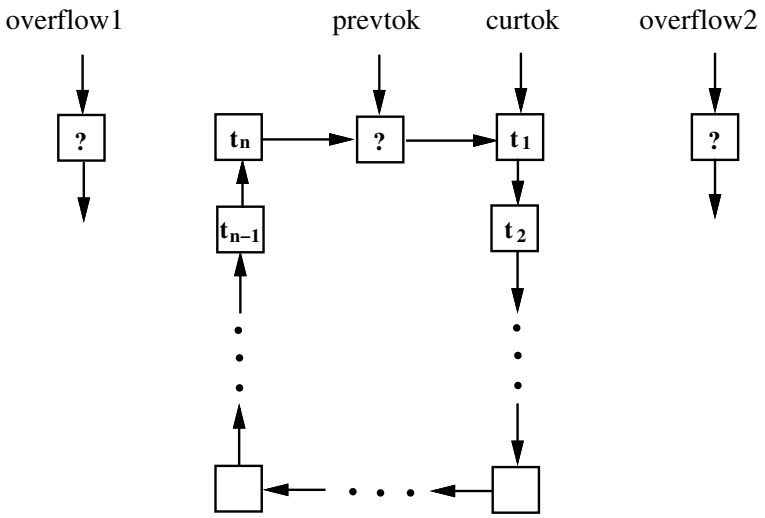

Figure 4.10: Initial buffer configuration

A good scheme for managing the input buffer is discussed in the next section.

To reset state_stack, the first step is to initialize it with the proper state sequence if the successful recovery was found on previous_stack or next_stack. For a simple recovery, nothing more is required. For a phrase-level recovery, all states following the recovery state are removed from the stack. Similarly, for a scope recovery, the sequence of states on top of the stack that corresponds to the prefix of the scope in question is removed and the repair proceeds as if the error was a simple insertion of the left-hand side of the scope.

4.4.5 The Input Buffer

The input buffer can be implemented as a fixed-size circular queue containing the previous token and the next n tokens to be processed, for some fixed integer n. *prevtok* is a pointer variable that identifies the previous token element in the circular queue. Its successor in the queue is identified by *curtok* which is also a pointer variable. Initially, the *prevtok* element is undefined and the next n tokens from the input are read into the remaining elements. See figure 4.10.

If the input is correct, the buffer is processed as follows. Each time the parser executes a shift (or shift-reduce action), the next input token is read into the *prevtok* element, *prevtok* is set to *curtok* and *curtok* is updated to point to its successor. This process continues until the parse terminates. (When the end of the input source is reached, the lexical analyser is expected to keep returning $\perp$.)

Note how this scheme easily accomodates the general case of an LALR(k) parser with variable lookahead-strings described in the previous chapter. If the parser needs to perform more lookahead, after having consulted a given element of the buffer, it obtains the next lookahead symbol by simply moving to the next element in the queue. Of course, this implies that the size of the input buffer queue must be greater than or equal to k.

When an error is encountered, the reparation of the configuration usually involves

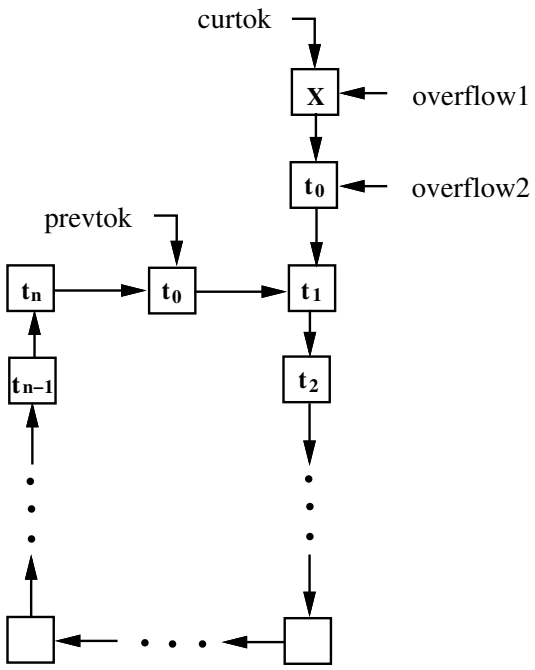

Figure 4.11: Buffer configuration after insertion of t in front of t_0

---

some modification of the input buffer. In particular, it may involve the insertion of some symbols. If a repair calls for the insertion of a symbol in front of *curtok* or *prevtok*, the buffer *overflows*. To accomodate such a repair, two extra *overflow* elements are needed. They are identified by two pointers: *overflow1* and *overflow2*. Let's consider the worst case, where the buffer contains a sequence: $t_0, t_1, \ldots, t_n$, where t_0 is the previous token, and the repair modification is to insert a symbol X in front of t_0. In that case, the previous token t_0 is copied into the *overflow2* element whose successor is set to be *curtok*; the new symbol X is placed in the *overflow1* element whose successor is set to be *overflow2* and *curtok* is set to *overflow1*. Figure 4.11 illustrates this case.

When executing a shift action on an overflow element, the parser simply updates *curtok* to point to its successor and no new tokens are read in. When the parser reenters the queue, the processing continues as before, and the next shift action updates the previous pointer. Note that this approach assumes that no error will occur on an overflow element. For normal recoveries, this can be guaranteed by *min_distance* being set to a value greater than or equal to 2. However, this restrinction explains the main reason why secondary phase recovery is not attempted on the previous_stack configuration. Recall from section 4.4.2 that in some cases, instead of applying a phrase-level recovery, a scope recovery is applied and control is returned to the parser with a configuration that will declare an error on the next input symbol. In such a case, if previous_stack is the state sequence that is used, then after the repair is applied, the input buffer is placed in a configuration similar to the configuration of Figure 4.11, where the symbol X is the left-hand side of the scope. Thus, after processing X, the parser will detect an error on an overflow element.

There are other repair cases where one or both overflow elements are needed in the

resetting of the input buffer. The detail of these cases is left to the reader.

4.5 Remarks

This error recovery method has been successfully implemented. Parsers were built for Ada and Pascal and tested on the Ada examples of [37] and the Pascal examples of [20]. (Appendix E shows some of these results.) Penello and DeRemer [17] proposed that the quality of a repair be rated "excellent" if it repaired the test as a human reader would have, "good" if it 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 [20] 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.