

2

Programming Language Syntax

2.3.4 Syntax Errors

EXAMPLE 2.43

Syntax error in C (reprise)

The main text illustrated the problem of syntax error recovery with a simple example in C:

```
A = B : C + D;
```

The compiler will detect a syntax error immediately after the B, but it cannot give up at that point: it needs to keep looking for errors in the remainder of the program. To permit this, we must modify the input program, the state of the parser, or both, in a way that allows parsing to continue, hopefully without announcing a significant number of spurious cascading errors and without missing a significant number of real errors. The techniques discussed below allow the compiler to search for further syntax errors. In Chapter 4 we will consider additional techniques that allow it to search for additional static semantic errors as well. ■

Panic Mode

Perhaps the simplest form of syntax error recovery is a technique known as *panic mode*. It defines a small set of “safe symbols” that delimit clean points in the input. When an error occurs, a panic mode recovery algorithm deletes input tokens until it finds a safe symbol, then backs the parser out to a context in which that symbol might appear. In the earlier example, a recursive descent parser with panic mode recovery might delete input tokens until it finds the semicolon, return from all subroutines called from within `stmt`, and restart the body of `stmt` itself.

Unfortunately, panic mode tends to be a bit drastic. By limiting itself to a static set of “safe” symbols at which to resume parsing, it admits the possibility of deleting a significant amount of input while looking for such a symbol. Worse, if some of the deleted tokens are “starter” symbols that begin large-scale constructs in the language (e.g., `begin`, `procedure`, `while`), we shall almost surely see spurious cascading errors when we reach the end of the construct.

EXAMPLE 2.44

The problem with panic mode

Consider the following fragment of code in Modula-2:

```
IF a b THEN x;
ELSE y;
END;
```

When it discovers the error at *b* in the first line, a panic-mode recovery algorithm is likely to skip forward to the semicolon, thereby missing the *THEN*. When the parser finds the *ELSE* on line 2 it will produce a spurious error message. When it finds the *END* on line 3 it will think it has reached the end of the enclosing structure (e.g., the whole subroutine), and will probably generate additional cascading errors on subsequent lines. Panic mode tends to work acceptably only in relatively “unstructured” languages, such as Basic and (early) Fortran, which don’t have many “starter” symbols. ■

Phrase-Level Recovery

We can improve the quality of recovery by employing different sets of “safe” symbols in different contexts. Parsers that incorporate this improvement are said to implement *phrase-level recovery*. When it discovers an error in an expression, for example, a phrase-level recovery algorithm can delete input tokens until it reaches something that is likely to follow an expression. This more local recovery is better than always backing out to the end of the current statement, because it gives us the opportunity to examine the parts of the statement that follow the erroneous expression.

EXAMPLE 2.45

Phrase-level recovery in recursive descent

Niklaus Wirth, the inventor of Pascal, published an elegant implementation of phrase-level recovery for recursive descent parsers in 1976 [Wir76, Sec. 5.9]. The simplest version of his algorithm depends on the *FIRST* and *FOLLOW* sets defined at the end of Section 2.3.1. If the parsing routine for nonterminal *foo* discovers an error at the beginning of its code, it deletes incoming tokens until it finds a member of *FIRST(foo)*, in which case it proceeds, or a member of *FOLLOW(foo)*, in which case it returns:

```
procedure foo
  if (input_token ∉ FIRST(foo)) and (not EPS(foo))
    report_error          -- print message for the user
  repeat
    delete_token
  until input_token ∈ (FIRST(foo) ∪ FOLLOW(foo) ∪ { $$ })
  case input_token of
    ...: ...
    ...: ...                -- valid starting tokens
    ...: ...
  otherwise return        -- error or foo → ε
```

Note that the *report_error* routine does *not* terminate the parse; it simply prints a message and returns. To complete the algorithm, the *match* routine must be

altered so that it, too, will return after announcing an error, effectively inserting the expected token when something else appears:

```

procedure match(expected)
  if input_token = expected
    consume input_token
  else
    report_error

```

Finally, to simplify the code, the common prefix of the various nonterminal sub-routines can be moved into an error-checking subroutine:

```

procedure check_for_error(symbol)
  if (input_token  $\notin$  FIRST(symbol)) and (not EPS(symbol))
    report_error
  repeat
    delete_token
  until input_token  $\in$  (FIRST(symbol)  $\cup$  FOLLOW(symbol)  $\cup$  { $\$$ })

```

Context-Specific Look-Ahead

Though simple, the recovery algorithm just described has an unfortunate tendency, when $foo \rightarrow \epsilon$, to predict one or more epsilon productions when it should really announce an error right away. This weakness is known as the *immediate error detection* problem. It stems from the fact that FOLLOW(foo) is context-independent: it contains all tokens that may follow foo somewhere in some valid program, but not necessarily in the current context in the current program. (This is basically the same observation that underlies the distinction between SLR and LALR parsers, as mentioned in Section 2.3.3 [page 91]).

EXAMPLE 2.46

Cascading syntax errors

As an example, consider the following incorrect code in our calculator language:

```
Y := (A * X X*X) + (B * X*X) + (C * X) + D
```

To a human being, it is pretty clear that the programmer forgot a $*$ in the x^3 term of a polynomial. The recovery algorithm isn't so smart. In a recursive descent parser it will see an identifier (X) coming up on the input when it is inside the following routines:

```

program
stmt_list
stmt
expr
term
factor
expr
term
factor_tail
factor_tail

```

Since an *id* can follow a *factor_tail* in some programs (e.g., $A := B \quad C := D$), the innermost parsing routine will predict *factor_tail* $\rightarrow \epsilon$, and simply return. At that point both the outer *factor_tail* and the inner term will be at the end of their code, and they, too, will return. Next, the inner *expr* will call *term_tail*, which will also predict an epsilon production, since an *id* can follow a *term_tail* in certain programs. This will leave the inner *expr* at the end of its code, allowing it to return. Only then will we discover an error, when *factor* calls *match*, expecting to see a right parenthesis. Afterward there will be a host of cascading errors, as the input is transformed into

```
Y := (A * X)
X := X
B := X*X
C := X
```

EXAMPLE 2.47

Reducing cascading errors with context-specific look-ahead

To avoid inappropriate epsilon predictions, Wirth introduced the notion of context-specific FOLLOW sets, passed into each nonterminal subroutine as an explicit parameter. In our example, we would pass *id* as part of the FOLLOW set for the initial, outer *expr*, which is called as part of the production *stmt* $\rightarrow id := expr$, but *not* into the second, inner *expr*, which is called as part of the production *factor* $\rightarrow (expr)$. The nested calls to *term* and *factor_tail* will end up being called with a FOLLOW set whose only member is a right parenthesis. When the inner call to *factor_tail* discovers that *id* is not in $FIRST(factor_tail)$, it will delete tokens up to the right parenthesis before returning. The net result is a single error message, and a transformation of the input into

```
Y := (A * X*X) + (B * X*X) + (C * X) + D
```

That's still not the "right" interpretation, but it's a lot better than it was.

The final version of Wirth's phrase-level recovery employs one additional heuristic: to avoid cascading errors it refrains from deleting members of a statically defined set of "starter" symbols (e.g., *begin*, *procedure*, (etc.)). These are the symbols that tend to require matching tokens later in the program. If we see a starter symbol while deleting input, we give up on the attempt to delete the rest of the erroneous construct. We simply return, even though we know that the starter symbol will not be acceptable to the calling routine. With context-specific FOLLOW sets and starter symbols, phrase-level recovery looks like this:

EXAMPLE 2.48

Recursive descent with full phrase-level recovery

```
procedure check_for_error(symbol, follow_set)
  if (input_token  $\notin$  FIRST(symbol)) and (not EPS(symbol))
    report_error
    repeat
      delete_token
    until input_token  $\in$  FIRST(symbol)  $\cup$  follow_set  $\cup$  starter_set  $\cup$  { $$$$$ }
```

```

procedure expr(follow_set)
  check_for_error(expr, follow_set)
  case input_token of
    ...:
    ...:
    ...:
    ...:
    otherwise return

```

Exception-Based Recovery in Recursive Descent

An attractive alternative to Wirth's technique relies on the exception-handling mechanisms available in many modern languages (we will discuss these mechanisms in detail in Section 8.5). Rather than implement recovery for every nonterminal in the language (a somewhat tedious task), the exception-based approach identifies a small set of contexts to which we back out in the event of an error. In many languages, we could obtain simple, but probably serviceable error recovery by backing out to the nearest statement or declaration. In the limit, if we choose a single place to “back out to,” we have an implementation of panic-mode recovery.

The basic idea is to attach an exception handler (a special syntactic construct) to the blocks of code in which we want to implement recovery:

EXAMPLE 2.49

Exceptions in a recursive descent parser

```

procedure statement
  try
    ...
  except when syntax_error
    loop
      if next_token ∈ FIRST(statement)
        statement
        return
      elsif next_token ∈ FOLLOW(statement)
        return
      else get_next_token

```

Code for declaration would be similar. For better-quality repair, we might add handlers around the bodies of expression, aggregate, or other complex constructs. To guarantee that we can always recover from an error, we must ensure that all parts of the grammar lie inside at least one handler.

When we detect an error (possibly nested many procedure calls deep), we *raise* a syntax error exception (“*raise*” is a built-in command in languages with exceptions). The language implementation then unwinds the stack to the most recent context in which we have an exception handler, which it executes in place of the remainder of the block to which the handler is attached. For phrase-level (or panic mode) recovery, the handler can delete input tokens until it sees one with which it can recommence parsing.

As noted in Section 2.3.1, the ANTLR parser generator takes a CFG as input and builds a human-readable recursive descent parser. Compiler writers have the option of generating Java, C#, or C++, all of which have exception-handling mechanisms. When an ANTLR-generated parser encounters a syntax error, it throws a `MismatchedTokenException` or `NoViableAltException`. By default ANTLR includes a handler for these exceptions in every nonterminal subroutine. The handler prints an error message, deletes tokens until it finds something in the FOLLOW set of the nonterminal, and then returns. The compiler writer can define alternative handlers if desired on a production-by-production basis.

Error Productions

As a general rule, it is desirable for an error recovery technique to be as language-independent as possible. Even in a recursive descent parser, which is handwritten for a particular language, it is nice to be able to encapsulate error recovery in the `check_for_error` and `match` subroutines. Sometimes, however, one can obtain much better repairs by being highly language specific.

EXAMPLE 2.50

Error production for
“; else”

Most languages have a few unintuitive rules that programmers tend to violate in predictable ways. In Pascal, for example, semicolons are used to separate statements, but many programmers think of them as *terminating* statements instead. Most of the time the difference is unimportant, since a statement is allowed to be empty. In the following, for example,

```
begin
  x := (-b + sqrt(b*b -4*a*c)) / (2*a);
  writeln(x);
end;
```

the compiler parses the `begin...end` block as a sequence of three statements, the third of which is empty. In the following, however,

```
if d <> 0 then
  a := n/d;
else
  a := n;
end;
```

the compiler must complain, since the `then...then...else` construct must consist of a single statement in Pascal. A Pascal semicolon is never allowed immediately before an `else`, but programmers put them there all the time. Rather than try to tune a general recovery or repair algorithm to deal correctly with this problem, most Pascal compiler writers modify the grammar: they include an extra production that allows the semicolon, but causes the semantic analyzer to print a warning message, telling the user that the semicolon shouldn't be there. Similar error productions are used in C compilers to cope with “anachronisms” that have crept into the language as it evolved. Syntax that was valid only in early versions of C is still accepted by the parser, but evokes a warning message. ■

Error Recovery in Table-Driven LL Parsers

Given the similarity to recursive descent parsing, it is straightforward to implement phrase-level recovery in a table-driven top-down parser. Whenever we encounter an error entry in the parse table, we simply delete input tokens until we find a member of a statically defined set of starter symbols (including $\$ \$$), or a member of the FIRST or FOLLOW set of the nonterminal at the top of the parse stack.¹ If we find a member of the FIRST set, we continue the main loop of the driver. If we find a member of the FOLLOW set or the starter set, we pop the nonterminal off the parse stack first. If we encounter an error in match, rather than in the parse table, we simply pop the token off the parse stack.

But we can do better than this! Since we have the entire parse stack easily accessible (it was hidden in the control flow and procedure calling sequence of recursive descent), we can enumerate all possible combinations of insertions and deletions that would allow us to continue parsing. Given appropriate metrics, we can then evaluate the alternatives to pick the one that is in some sense “best.”

Because perfect error recovery (actually error *repair*) would require that we read the programmer’s mind, any practical technique to evaluate alternative “corrections” must rely on heuristics. For the sake of simplicity, most compilers limit themselves to heuristics that (1) require no semantic information, (2) do not require that we “back up” the parser or the input stream (i.e., to some state prior to the one in which the error was detected), and (3) do not change the spelling of tokens or the boundaries between them. A particularly elegant algorithm that conforms to these limits was published by Fischer, Milton, and Quiring in 1980 [FMQ80, FL88]. As originally described, the algorithm was limited to languages in which programs could always be corrected by inserting appropriate tokens into the input stream, without ever requiring deletions. It is relatively easy, however, to extend the algorithm to encompass deletions and substitutions. We consider the insert-only algorithm first; the version with deletions employs it as a subroutine. We do not consider substitutions here.²

The FMQ error-repair algorithm requires the compiler writer to assign an insertion cost $C(\tau)$ and a deletion cost $D(\tau)$ to every token τ . (Since we cannot change where the input ends, we have $C(\$ \$) = D(\$ \$) = \infty$.) In any given error situation, the algorithm chooses the least cost combination of insertions and

¹ This description uses global FOLLOW sets. If we want to use context-specific look-aheads instead, we can peek farther down in the stack. A token is an acceptable context-specific look-ahead if it is in the FIRST set of the second symbol A from the top in the stack or, if it would cause us to predict $A \rightarrow \epsilon$, the FIRST set of the third symbol B from the top or, if it would cause us to predict $B \rightarrow \epsilon$, the FIRST set of the fourth symbol from the top, and so on.

² A substitution can always be effected as a deletion/insertion pair, but we might want ideally to give it special consideration. For example, we probably want to be cautious about deleting a left square bracket or inserting a left parenthesis, since both of these symbols must be matched by something later in the input, at which point we are likely to see cascading errors. But substituting a left parenthesis for a left square bracket is in some sense more plausible, especially given the differences in array subscript syntax in different programming languages.

deletions that allows the parser to consume one more token of real input. The state of the parser is never changed; only the input is modified (rather than pop a stack symbol, the repair algorithm pushes its yield onto the input stream).

As in phrase-level recovery in a recursive descent parser, the FMQ algorithm needs to address the immediate error detection problem. There are several ways we could do this. One would be to use a “full LL” parser, which keeps track of local FOLLOW sets. Another would be to inspect the stack when predicting an epsilon production, to see if what lies underneath will allow us to accept the incoming token. The first option significantly increases the size and complexity of the parser. The second option leads to a nonlinear-time parsing algorithm. Fortunately, there is a third option. We can save all changes to the stack (and calls to the semantic analyzer’s action routines) in a temporary buffer until the match routine accepts another real token of input. If we discover an error before we accept a real token, we undo the stack changes and throw away the buffered calls to action routines. Then we can pretend we recognized the error when a full LL parser would have.

We now consider the task of repairing with only insertions. We begin by extending the notion of insertion costs to strings in the obvious way: if $w = a_1 a_2 \dots a_n$, we have $C(w) = \sum_{i=1}^n C(a_i)$. Using the cost function C , we then build a pair of tables S and E . The S table is one-dimensional, and is indexed by grammar symbol. For any symbol X , $S(X)$ is a least-cost string of terminals derivable from X . That is,

$$S(X) = w \iff X \Rightarrow^* w \text{ and } \forall x \text{ such that } X \Rightarrow^* x, C(w) \leq C(x)$$

Clearly $S(a) = a \forall$ tokens a .

The E table is two-dimensional, and is indexed by symbol/token pairs. For any symbol X and token a , $E(X, a)$ is the lowest-cost prefix of a in X ; that is, the lowest cost token string w such that $X \Rightarrow^* wax$. If X cannot yield a string containing a , then $E(X, a)$ is defined to be a special symbol $??$ whose insertion cost is ∞ . If $X = a$, or if $X \Rightarrow^* ax$, then $E(X, a) = \epsilon$, where $C(\epsilon) = 0$.

To find a least-cost insertion that will repair a given error, we execute the function `find_insertion`, shown in Figure 2.30. The function begins by considering the least-cost insertion that will allow it to derive the input token from the symbol at the top of the stack (there may be none). It then considers the possibility of “deleting” that top-of-stack symbol (by inserting its least-cost yield into the input stream) and deriving the input token from the second symbol on the stack. It continues in this fashion, considering ways to derive the input token from ever deeper symbols on the stack, until the cost of inserting the yields of the symbols above exceeds the cost of the cheapest repair found so far. If it reaches the bottom of the stack without finding a finite-cost repair, then the error cannot be repaired by insertions alone. ■

EXAMPLE 2.51

Insertion-only repair in FMQ

EXAMPLE 2.52

FMQ with deletions

To produce better-quality repairs, and to handle languages that cannot be repaired with insertions only, we need to consider deletions. As we did with the insert cost vector C , we extend the deletion cost vector D to strings of tokens in


```

function find_insertion(a : token) : string
  -- assume that the parse stack consists of symbols  $X_n, \dots, X_2, X_1$ ,
  -- with  $X_n$  at top-of-stack
  ins := ??
  prefix :=  $\epsilon$ 
  for i in n..1
    if  $C(\text{prefix}) \geq C(\text{ins})$ 
      -- no better insertion is possible
      return ins
    if  $C(\text{prefix} \cdot E(X_i, a)) < C(\text{ins})$ 
      -- better insertion found
      ins := prefix ·  $E(X_i, a)$ 
      prefix := prefix ·  $S(X_i)$ 
  return ins

```

Figure 2.30 Outline of a function to find a least-cost insertion that will allow the parser to accept the input token a . The dot character (.) is used here for string concatenation.

```

function find_repair : string, int
  -- assume that the parse stack consists of symbols  $X_n, \dots, X_2, X_1$ ,
  -- with  $X_n$  at top-of-stack,
  -- and that the input stream consists of tokens  $a_1, a_2, a_3, \dots$ 
  i := 0      -- number of tokens we're considering deleting
  best_ins := ??
  best_del := 0
  loop
    cur_ins := find_insertion( $a_{i+1}$ )
    if  $C(\text{cur\_ins}) + D(a_1 \dots a_i) < C(\text{best\_ins}) + D(a_1 \dots a_{\text{best\_del}})$ 
      -- better repair found
      best_ins := cur_ins
      best_del := i
    i += 1
    if  $D(a_1 \dots a_i) > C(\text{best\_ins}) + D(a_1 \dots a_{\text{best\_del}})$ 
      -- no better repair is possible
      return (best_ins, best_del)

```

Figure 2.31 Outline of a function to find a least-cost combination of insertions and deletions that will allow the parser to accept one more token of input.

the obvious way. We then embed calls to `find_insertion` in a second loop, shown in Figure 2.31. This loop repeatedly considers deleting more and more tokens, each time calling `find_insertion` on the remaining input, until the cost of deleting additional tokens exceeds the cost of the cheapest repair found so far. The search can never fail; it is always possible to find a combination of insertions and deletions that will allow the end-of-file token to be accepted. Since the algorithm may need to consider (and then reject) the option of deleting an arbitrary number of

tokens, the scanner must be prepared to peek an arbitrary distance ahead in the input stream and then back up again. ■

The FMQ algorithm has several desirable properties. It is simple and efficient (given that the grammar is bounded in size, we can prove that the time to choose a repair is bounded by a constant). It can repair an arbitrary input string. Its decisions are locally optimal, in the sense that no cheaper repair can allow the parser to make forward progress. It is table-driven and therefore fully automatic. Finally, it can be tuned to prefer “likely” repairs by modifying the insertion and deletion costs of tokens. Some obvious heuristics include:

- Deletion should usually be more expensive than insertion.
- Common operators (e.g., multiplication) should have lower cost than uncommon operators (e.g., modulo division) in the same place in the grammar.
- Starter symbols (e.g., `begin`, `if`, `(`) should have higher cost than their corresponding final symbols (`end`, `fi`, `)`).
- “Noise” symbols (comma, semicolon, `do`) should have very low cost.

Error Recovery in Bottom-Up Parsers

Locally least-cost repair is possible in bottom-up parsers, but it isn’t as easy as it is in top-down parsers. The advantage of a top-down parser is that the content of the parse stack unambiguously identifies the context of an error, and specifies the constructs expected in the future. The stack of a bottom-up parser, by contrast, describes a set of possible contexts, and says nothing explicit about the future.

In practice, most bottom-up parsers tend to rely on panic-mode or phrase-level recovery. The intuition is that when an error occurs, the top few states on the parse stack represent the shifted prefix of an erroneous construct. Recovery consists of popping these states off the stack, deleting the remainder of the construct from the incoming token stream, and then restarting the parser, possibly after shifting a fictitious nonterminal to represent the erroneous construct.

Unix’s `yacc/bison` provides a typical example of bottom-up phrase-level recovery. In addition to the usual tokens of the language, `yacc/bison` allows the compiler writer to include a special token, `error`, anywhere in the right-hand sides of grammar productions. When the parser built from the grammar detects a syntax error, it

1. Calls the function `yyerror`, which the compiler writer must provide. Normally, `yyerror` simply prints a message (e.g., “parse error”), which `yacc/bison` passes as an argument
2. Pops states off the parse stack until it finds a state in which it can shift the error token (if there is no such state, the parser terminates)
3. Inserts and then shifts the `error` token
4. Deletes tokens from the input stream until it finds a valid look-ahead for the new (post error) context

5. Temporarily disables reporting of further errors
6. Resumes parsing

If there are any semantic action routines associated with the production containing the error token, these are executed in the normal fashion. They can do such things as print additional error messages, modify the symbol table, patch up semantic processing, prompt the user for additional input in an interactive tool (yacc/bison can be used to build things other than batch-mode compilers), or disable code generation. The rationale for disabling further syntax errors is to make sure that we have really found an acceptable context in which to resume parsing before risking cascading errors. Yacc/bison automatically re-enables the reporting of errors after successfully shifting three real tokens of input. A semantic action routine can re-enable error messages sooner if desired by calling the built-in routine `yyerrorok`.

EXAMPLE 2.53

Panic mode in yacc/bison

For our example calculator language, we can imagine building a yacc/bison parser using the bottom-up grammar of Figure 2.24. For panic-mode recovery, we might want to back out to the nearest statement:

```
stmt → error
      {printf("parsing resumed at end of current statement\n");}
```

The semantic routine written in curly braces would be executed when the parser recognizes *stmt* → error.³ Parsing would resume at the next token that can follow a statement—in our calculator language, at the next `id`, `read`, `write`, or `$$`. ■

EXAMPLE 2.54

Panic mode with statement terminators

A weakness of the calculator language, from the point of view of error recovery, is that the current, erroneous statement may well contain additional `ids`. If we resume parsing at one of these, we are likely to see another error right away. We could avoid the error by disabling error messages until several real tokens have been shifted. In a language in which every statement ends with a semicolon, we could have more safely written:

```
stmt → error ;
      {printf("parsing resumed at end of current statement\n");} ■
```

EXAMPLE 2.55

Phrase-level recovery in yacc/bison

In both of these examples we have placed the `error` symbol at the beginning of a right-hand side, but there is no rule that says it must be so. We might decide, for example, that we will abandon the current statement whenever we see an error, unless the error happens inside a parenthesized expression, in which case we will attempt to resume parsing after the closing parenthesis. We could then add the following production:

```
factor → ( error )
        {printf("parsing resumed at end of
                parenthesized expression\n");}
```

3 The syntax shown here is not the same as that accepted by yacc/bison, but is used for the sake of consistency with earlier material.

In the CFSM of Figure 2.25, it would then be possible in State 8 to shift `error`, delete some tokens, shift `)`, recognize *factor*, and continue parsing the surrounding expression. Of course, if the erroneous expression contains nested parentheses, the parser may not skip all of it, and a cascading error may still occur. ■

Because yacc/bison creates LALR parsers, it automatically employs context-specific look-ahead, and does not usually suffer from the immediate error detection problem. (A full LR parser would do slightly better.) In an SLR parser, a good error recovery algorithm needs to employ the same trick we used in the top-down case. Specifically, we buffer all stack changes and calls to semantic action routines until the shift routine accepts a real token of input. If we discover an error before we accept a real token, we undo the stack changes and throw away the buffered calls to semantic routines. Then we can pretend we recognized the error when a full LR parser would have.

✓ CHECK YOUR UNDERSTANDING

45. Why is syntax error recovery important?
 46. What are *cascading errors*?
 47. What is *panic mode*? What is its principal weakness?
 48. What is the advantage of *phrase-level recovery* over panic mode?
 49. What is the *immediate error detection problem*, and how can it be addressed?
 50. Describe two situations in which context-specific FOLLOW sets may be useful.
 51. Outline Wirth's mechanism for error recovery in recursive descent parsers. Compare this mechanism to exception-based recovery.
 52. What are *error productions*? Why might a parser that incorporates a high-quality, general-purpose error recovery algorithm still benefit from using such productions?
 53. Outline the FMQ algorithm. In what sense is the algorithm optimal?
 54. Why is error recovery more difficult in bottom-up parsers than it is in top-down parsers?
 55. Describe the error recovery mechanism employed by yacc/bison.
-