2 Programming Language Syntax

2.3.5 Recovering from 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 an Algol-family language:

```
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—or perhaps that is able to start an expression, in the hope that what was deleted was an ignorable prefix. 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 (and maybe even the expression) that follow the erroneous tokens.

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 not (input_token ∈ FIRST(foo) or (EPS(foo) and input_token ∈ FOLLOW(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 subroutines can be moved into an error-checking subroutine:

```
procedure check_for_error(sym)
    if not (input_token $\in$ FIRST(sym) or EPS(sym) and input_token $\in$ FOLLOW(sym))
        report_error()
        repeat
            delete_token()
        until input_token $\in$ (FIRST(sym) $\cup$ FOLLOW(sym) $\cup$ {$$})
```
■

Context-Specific Look-Ahead

Though simple, the recovery algorithm just described has an unfortunate tendency, when *foo* $\longrightarrow \epsilon$, 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 ("The Characteristic Finite-State Machine and LR Parsing Variants," Section 2.3.4).

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    C := D`), the innermost parsing routine will predict *factor_tail* $\longrightarrow \epsilon$, 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* $\longrightarrow$ `id :=` *expr*, but *not* into the second, inner expr, which is called as part of the production *factor* $\longrightarrow$ `(` *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(sym, follow_set)
    if not (input_token ∈ FIRST(sym) or (EPS(sym) and input_token ∈ follow_set))
        report_error()
        repeat
            delete_token()
        until input_token ∈ FIRST(sym) ∪ follow_set ∪ starter_set ∪ {$$}
```

```
procedure expr(follow_set)
    check_for_error(*expr*, follow_set)
    case input_token of
        . . . : . . .
        . . . : . . .                  valid starting tokens
        . . . : . . .
        otherwise return
```

Context-specific FOLLOW sets are tracked dynamically during the parse of a given input. Initially, in the augmenting production $S \longrightarrow \textit{program}$ $$, the context-specific FOLLOW set for *program* is {$$}. Thus when calling the recursive descent routine for program, we pass {$$} as parameter. Then, within each routine, we determine the FOLLOW sets to pass to other routines based on whatever comes next in the current right-hand side, potentially augmented by what was already passed as the FOLLOW set of the current left-hand side. Specifically, suppose we are currently executing the recursive descent routine for symbol A, called with context-specific FOLLOW set S. Suppose further that we have realized (predicted) that we are in the production $A \longrightarrow \alpha\ B\ \beta$ and we are about to call the routine for symbol B. If $\beta \Longrightarrow^* \epsilon$, we will pass FIRST($\beta$) $\cup$ S as the context-specific FOLLOW set for B. If β cannot generate ϵ, we will simply pass FIRST(β).

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 9.4). 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.

EXAMPLE 2.49
Exceptions in a recursive descent parser

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:

```
procedure statement()
    try
        . . .                    –– code to parse a statement
    except when syntax_error
        loop
            if next_token ∈ FIRST(*statement*)
                statement()        –– try again
                return
            elsif next_token ∈ FOLLOW(*statement*)
                return
            else delete_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` part of an `if`... `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]. 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]

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 $A \longrightarrow \varepsilon$, the FIRST set of the third symbol B from the top or, if it would cause us to predict $B \longrightarrow \varepsilon$, the FIRST set of the fourth symbol from the top, and so on.

The FMQ error-repair algorithm requires the compiler writer to assign an insertion cost $C(\texttt{t})$ and a deletion cost $D(\texttt{t})$ to every token `t`. (Since we cannot change where the input ends, we have $C(\texttt{\$\$}) = D(\texttt{\$\$}) = \infty$.) In any given error situation, the algorithm chooses the least cost combination of insertions and 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 $w = a_1 a_2 \ldots a_n$, 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,

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

Clearly $S(\texttt{a}) = \texttt{a}\ \forall$ tokens `a`.

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

EXAMPLE 2.51 Insertion-only repair in FMQ

To find a least-cost insertion that will repair a given error, we execute the function find_insertion, shown in Figure C-2.31. 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

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.

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

Figure 2.31 **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 $X_n, \ldots X_2, X_1$,
    –– with $X_n$ at top-of-stack,
    –– and that the input stream consists of tokens $a_1, a_2, a_3, \ldots$
    i := 0        –– number of tokens we're considering deleting
    best_ins := ??
    best_del := 0
    loop
        cur_ins := find_insertion($a_{i+1}$)
        if $C$(cur_ins) + $D(a_1 \ldots a_i)$ < $C$(best_ins) + $D(a_1 \ldots a_{\text{best\_del}})$
            –– better repair found
            best_ins := cur_ins
            best_del := i
        i +:= 1
        if $D(a_1 \ldots a_i)$ > $C$(best_ins) + $D(a_1 \ldots a_{\text{best\_del}})$
            –– no better repair is possible
            return ⟨best_ins, best_del⟩
```

Figure 2.32 **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.**

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 2.52
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 the obvious way. We then embed calls to find_insertion in a second loop, shown in Figure C-2.32. 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 reenables the reporting of errors after successfully shifting three real tokens of input. A semantic action routine can reenable 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 2.25. 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 `id`s. 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,

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.

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");}
```

In the CFSM of Figure 2.26, 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

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

2 Programming Language Syntax

2.4 Theoretical Foundations

As noted in the main text, scanners and parsers are based on the finite automata and pushdown automata that form the bottom two levels of the Chomsky language hierarchy. At each level of the hierarchy, machines can be either *deterministic* or *nondeterministic*. A deterministic automaton always performs the same operation in a given situation. A nondeterministic automaton can perform any of a *set* of operations. A nondeterministic machine is said to accept a string if there exists a choice of operation in each situation that will eventually lead to the machine saying "yes." As it turns out, nondeterministic and deterministic finite automata are equally powerful: every DFA is, by definition, a degenerate NFA, and the construction in Example 2.14 demonstrates that for any NFA we can create a DFA that accepts the same language. The same is not true of push-down automata: there are context-free languages that are accepted by an NPDA but not by any DPDA. Fortunately, DPDAs suffice in practice to accept the syntax of real programming languages. Practical scanners and parsers are always deterministic.

2.4.1 Finite Automata

Precisely defined, a deterministic finite automaton (DFA) M consists of (1) a finite set Q of *states*, (2) a finite alphabet Σ of input symbols, (3) a distinguished *initial* state $q_1 \in Q$, (4) a set of distinguished *final* states $F \subseteq Q$, and (5) a *transition function* $\delta : Q \times \Sigma \rightarrow Q$ that chooses a new state for M based on the current state and the current input symbol. M begins in state q_1. One by one it consumes its input symbols, using δ to move from state to state. When the final symbol has been consumed, M is interpreted as saying "yes" if it is in a state in F; otherwise it is interpreted as saying "no." We can extend δ in the obvious way to take strings, rather than symbols, as inputs, allowing us to say that M accepts string x if $\delta(q_1, x) \in F$. We can then define $L(M)$, the language accepted by M,

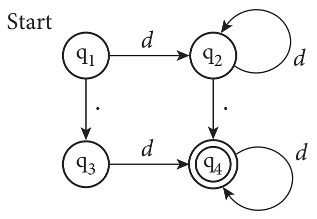

Figure 2.33 **Minimal DFA for the language consisting of all strings of decimal digits containing a single decimal point.** Adapted from Figure 2.10 in the main text. The symbol d here is short for "0, 1, 2, 3, 4, 5, 6, 7, 8, 9".

to be the set $\{x \mid \delta(q_1, x) \in F\}$. In a nondeterministic finite automaton (NFA), the transition function, δ, is multivalued: the automaton can move to any of a *set* of possible states from a given state on a given input. In addition, it may move from one state to another "spontaneously"; such transitions are said to take input symbol ε.

EXAMPLE 2.56
Formal DFA for $d^*(\,.\,d \mid d\,.\,)\,d^*$

We can illustrate these definitions with an example. Consider the circles-and-arrows automaton of Figure C-2.33 (adapted from Figure 2.10 in the main text). This is the minimal DFA accepting strings of decimal digits containing a single decimal point. $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$ is the machine's input alphabet. $Q = \{q_1, q_2, q_3, q_4\}$ is the set of states; q_1 is the initial state; $F = \{q_4\}$ (a singleton in this case) is the set of final states. The transition function can be represented by a set of triples $\delta = \{(q_1, 0, q_2), \ldots, (q_1, 9, q_2), (q_1, ., q_3), (q_2, 0, q_2), \ldots, (q_2, 9, q_2), (q_2, ., q_4), (q_3, 0, q_4), \ldots, (q_3, 9, q_4), (q_4, 0, q_4), \ldots, (q_4, 9, q_4)\}$. In each triple $(q_i, \mathtt{a}, q_j)$, $\delta(q_i, \mathtt{a}) = q_j$. ■

Given the constructions of Examples 2.12 and 2.14, we know that there exists an NFA that accepts the language generated by any given regular expression, and a DFA equivalent to any given NFA. To show that regular expressions and finite automata are of equivalent expressive power, all that remains is to demonstrate that there exists a regular expression that generates the language accepted by any given DFA. We illustrate the required construction below for our decimal strings example (Figure C-2.33). More formal and general treatment of all the regular language constructions can be found in standard automata theory texts [HMU07, Sip13].

From a DFA to a Regular Expression

To construct a regular expression equivalent to a given DFA, we employ a dynamic programming algorithm that builds solutions to successively more complicated subproblems from a table of solutions to simpler subproblems. We begin with a set of simple regular expressions that describe the transition function, δ. For all states i, we define

$$r^0_{ii} = \mathtt{a}_1 \mid \mathtt{a}_2 \mid \ldots \mid \mathtt{a}_m \mid \varepsilon$$

where $\{\texttt{a}_1 \mid \texttt{a}_2 \mid \ldots \mid \texttt{a}_m\} = \{\texttt{a} \mid \delta(q_i, \texttt{a}) = q_i\}$ is the set of characters labeling the "self-loop" from state q_i back to itself. If there is no such self-loop, $r_{ij}^0 = \varepsilon$. Similarly, for $i \neq j$, we define

$$r_{ij}^0 = \texttt{a}_1 \mid \texttt{a}_2 \mid \ldots \mid \texttt{a}_m$$

where $\{\texttt{a}_1 \mid \texttt{a}_2 \mid \ldots \mid \texttt{a}_m\} = \{\texttt{a} \mid \delta(q_i, \texttt{a}) = q_j\}$ is the set of characters labeling the arc from q_i to q_j. If there is no such arc, r_{ij}^0 is the empty regular expression. (Note the difference here: we can stay in state q_i by not accepting any input, so ε is always one of the alternatives in r_{ii}^0, but not in r_{ij}^0 when $i \neq j$.)

Given these r^0 expressions, the dynamic programming algorithm inductively calculates expressions r_{ij}^k with larger superscripts. In each, k names the highest-numbered state through which control may pass on the way from q_i to q_j. We assume that states are numbered starting with q_1, so when $k = 0$ we must transition directly from q_i to q_j, with no intervening states.

EXAMPLE 2.57

Reconstructing a regular expression for the decimal string DFA

In our small example DFA, $r_{11}^0 = r_{33}^0 = \varepsilon$, and $r_{22}^0 = r_{44}^0 = \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \mid \varepsilon$, which we will abbreviate $d \mid \varepsilon$. Similarly, $r_{13}^0 = r_{24}^0 = \texttt{.}$, and $r_{12}^0 = r_{34}^0 = d$. Expressions r_{14}^0, r_{21}^0, r_{23}^0, r_{31}^0, r_{32}^0, r_{41}^0, r_{42}^0, and r_{43}^0 are all empty.

For $k > 0$, the r_{ij}^k expressions will generally generate multicharacter strings. At each step of the dynamic programming algorithm, we let

$$r_{ij}^k = r_{ij}^{k-1} \mid r_{ik}^{k-1} r_{kk}^{k-1}{}^{*} r_{kj}^{k-1}$$

That is, to get from q_i to q_j without going through any states numbered higher than k, we can either go from q_i to q_j without going through any state numbered higher than $k-1$ (which we already know how to do), or else we can go from q_i to q_k (without going through any state numbered higher than $k-1$), travel out from q_k and back again an arbitrary number of times (never visiting a state numbered higher than $k-1$ in between), and finally go from q_k to q_j (again without visiting a state numbered higher than $k-1$). If any of the constituent regular expressions is empty, we omit its term of the outermost alternation. At the end, our overall answer is $r_{1f_1}^n \mid r_{1f_2}^n \mid \ldots \mid r_{1f_t}^n$, where $n = |Q|$ is the total number of states and $F = \{q_{f_1}, q_{f_2}, \ldots, q_{f_t}\}$ is the set of final states.

Because $r_{11}^0 = \varepsilon$ and there are no transitions from States 2, 3, or 4 to State 1, nothing changes in the first inductive step in our example; that is, $\forall i\,[\,r_{ii}^1 = r_{ii}^0\,]$. The second step is a bit more interesting. Since we are now allowed to go through State 2, we have $r_{22}^2 = r_{22}^2\ r_{22}^2{}^{*} r_{22}^2 = (\,d \mid \varepsilon\,) \mid (\,d \mid \varepsilon\,)\,(\,d \mid \varepsilon\,)^{*}(\,d \mid \varepsilon\,) = d^{*}$. Because r_{21}^1, r_{23}^1, r_{32}^1, and r_{42}^1 are empty, however, r_{11}^2, r_{33}^2, and r_{44}^2 remain the same as r_{11}^1, r_{33}^1, and r_{44}^1. In a similar vein, we have

$$\begin{aligned}
r_{12}^2 &= d \mid d\,(\,d \mid \varepsilon\,)^{*}(\,d \mid \varepsilon\,) \;=\; d^{+} \\
r_{14}^2 &= d\,(\,d \mid \varepsilon\,)^{*}\,\texttt{.} \;=\; d^{+}\,\texttt{.} \\
r_{24}^2 &= \texttt{.} \mid (\,d \mid \varepsilon\,)\,(\,d \mid \varepsilon\,)^{*}\,\texttt{.} \;=\; d^{*}\,\texttt{.}
\end{aligned}$$

Missing transitions and empty expressions from the previous step leave $r_{13}^2 = r_{13}^1 = \texttt{.}$ and $r_{34}^2 = r_{34}^1 = d$. Expressions r_{21}^2, r_{23}^2, r_{31}^2, r_{32}^2, r_{41}^2, r_{42}^2, and r_{43}^2 remain empty.

In the third inductive step, we have

$$\begin{aligned} r_{13}^3 &= \texttt{.} \mid \texttt{.}\ \varepsilon^* \varepsilon = \texttt{.} \\ r_{14}^3 &= d^+ \texttt{.} \mid \texttt{.}\ \varepsilon^* d = d^+ \texttt{.} \mid \texttt{.}\ d \\ r_{34}^3 &= d \mid \varepsilon \varepsilon^* d = d \end{aligned}$$

All other expressions remain unchanged from the previous step.

Finally, we have

$$\begin{aligned} r_{14}^4 &= (\, d^+ \texttt{.} \mid \texttt{.}\ d \,) \mid (\, d^+ \texttt{.} \mid \texttt{.}\ d \,)(\, d \mid \varepsilon \,)^*(\, d \mid \varepsilon \,) \\ &= (\, d^+ \texttt{.} \mid \texttt{.}\ d \,) \mid (\, d^+ \texttt{.} \mid \texttt{.}\ d \,)\ d^* \\ &= (\, d^+ \texttt{.} \mid \texttt{.}\ d \,)\ d^* \\ &= d^+ \texttt{.}\ d^* \mid \texttt{.}\ d^+ \end{aligned}$$

Since F has a single member (q_4), this expression is our final answer. ■

Space Requirements

In Section 2.2.1 we noted without proof that the conversion from an NFA to a DFA may lead to exponential blow-up in the number of states. Certainly this did not happen in our decimal string example: the NFA of Figure 2.8 has 14 states, while the equivalent DFA of Figure 2.9 has only 7, and the minimal DFA of Figures 2.10 and C-2.33 has only 4.

EXAMPLE 2.58
A regular language with a large minimal DFA

Consider, however, the subset of `( a | b | c )*` in which some letter appears at least three times. The minimal DFA for this language has 28 states. As shown in Figure C-2.34, 27 of these are states in which we have seen i, j, and k `a`s, `b`s, and `c`s, respectively. The 28th (and only final) state is reached once we have seen at least three of some specific character.

By contrast, there exists an NFA for this language with only eight states, as shown in Figure C-2.35. It requires that we "guess," at the outset, whether we will see three `a`s, three `b`s, or three `c`s. It mirrors the structure of the natural regular expression `( a | b | c )* a ( a | b | c )* a ( a | b | c )* a ( a | b | c )* | ( a | b | c )* b ( a | b | c )* b ( a | b | c )* b ( a | b | c )* | ( a | b | c )* c ( a | b | c )* c ( a | b | c )* c ( a | b | c )*`. ■

EXAMPLE 2.59
Exponential DFA blow-up

Of course, the eight-state NFA does not emerge directly from the construction of Figure 2.7; that construction produces a 52-state machine with a certain amount of redundancy, and with many extraneous states and epsilon productions. But consider the similar subset of `( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )*` in which some digit appears at least ten times. The minimal DFA for this language has 10,000,000,001 states: a non-final state for each combination of zeros through nines with less than ten of each, and a single final state reached once any digit has appeared at least ten times. One possible regular expression for this language is

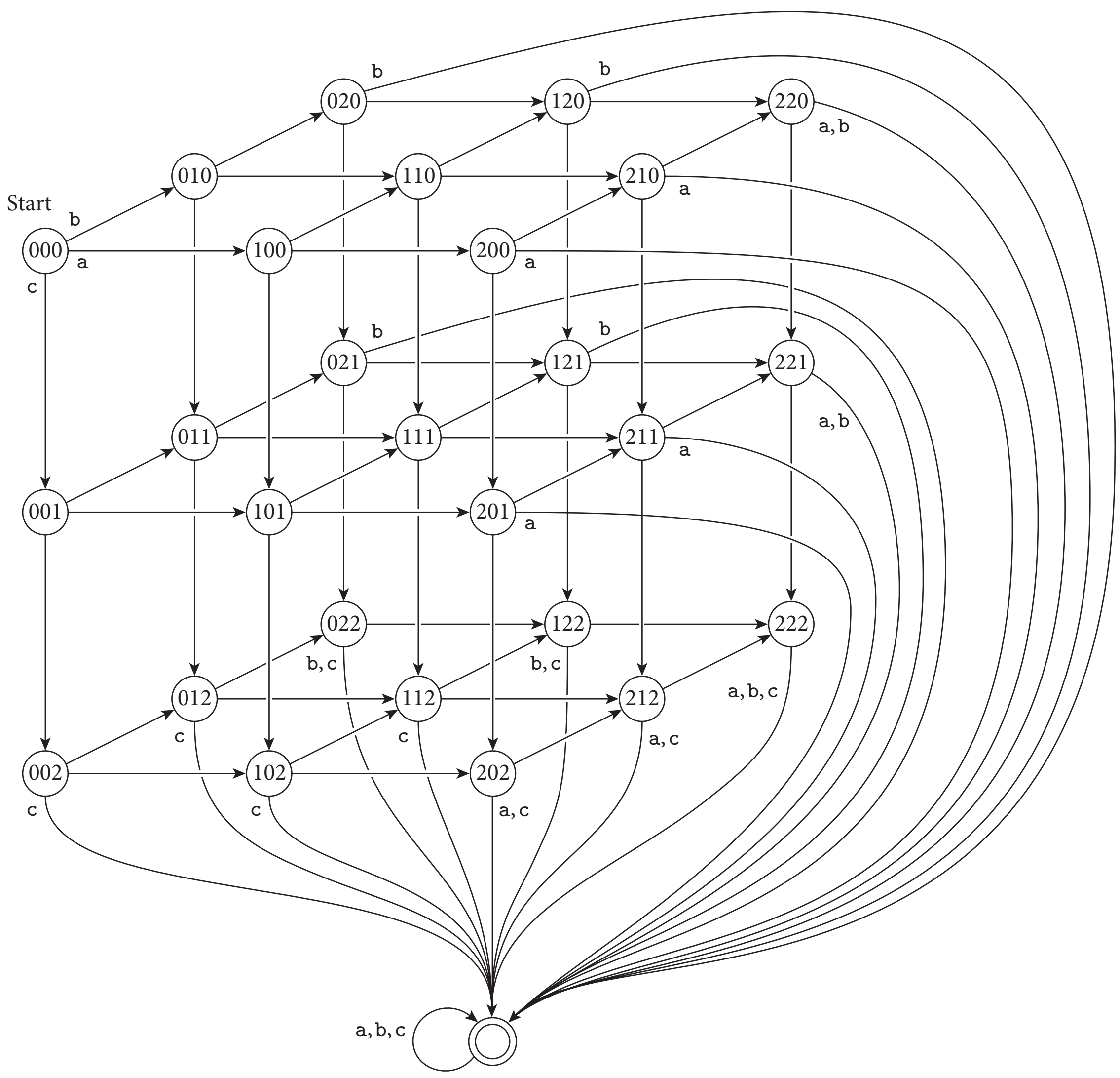

Figure 2.34 DFA for the language consisting of all strings in (a | b | c)* in which some letter appears at least three times. State name *ijk* indicates that *i* as, *j* bs, and *k* cs have been seen so far. Within the cubic portion of the figure, most edge labels are elided: a transitions move to the right, b transitions go back into the page, and c transitions move down.

```
 ((0|1|...|9)* 0 (0|1|...|9)* 0 (0|1|...|9)* 0 (0|1|...|9)* 0
  (0|1|...|9)* 0 (0|1|...|9)* 0 (0|1|...|9)* 0 (0|1|...|9)* 0
  (0|1|...|9)* 0 (0|1|...|9)* 0 (0|1|...|9)*)
| ((0|1|...|9)* 1 (0|1|...|9)* 1 (0|1|...|9)* 1 (0|1|...|9)* 1
  (0|1|...|9)* 1 (0|1|...|9)* 1 (0|1|...|9)* 1 (0|1|...|9)* 1
  (0|1|...|9)* 1 (0|1|...|9)* 1 (0|1|...|9)*)
| ...
```

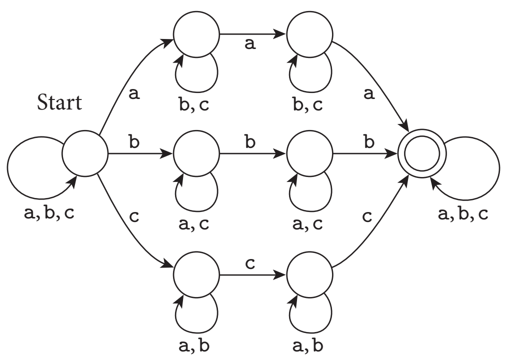

Figure 2.35 NFA corresponding to the DFA of Figure C-2.34.

```
| ((0|1|...|9)* 9 (0|1|...|9)* 9 (0|1|...|9)* 9 (0|1|...|9)* 9
   (0|1|...|9)* 9 (0|1|...|9)* 9 (0|1|...|9)* 9 (0|1|...|9)* 9
   (0|1|...|9)* 9 (0|1|...|9)* 9 (0|1|...|9)*)
```

Our construction would yield a very large NFA for this expression, but clearly many orders of magnitude smaller than ten billion states! ■

2.4.2 Push-Down Automata

A deterministic push-down automaton (DPDA) N consists of (1) Q, (2) Σ, (3) q_1, and (4) F, as in a DFA, plus (6) a finite alphabet Γ of stack symbols, (7) a distinguished initial stack symbol $Z_1 \in \Gamma$, and (5′) a transition function $\delta : Q \times \Gamma \times \{\Sigma \cup \{\varepsilon\}\} \rightarrow Q \times \Gamma^*$, where Γ^* is the set of strings of zero or more symbols from Γ. N begins in state q_1, with symbol Z_1 in an otherwise empty stack. It repeatedly examines the current state q and top-of-stack symbol Z. If $\delta(q,\varepsilon, Z)$ is defined, N moves to state r and replaces Z with α in the stack, where $(r, \alpha) = \delta(q,\varepsilon, Z)$. In this case N does not consume its input symbol. If $\delta(q,\varepsilon, Z)$ is undefined, N examines and consumes the current input symbol a. It then moves to state s and replaces Z with β, where $(s, \beta) = \delta(q, \text{a}, Z)$. N is interpreted as accepting a string of input symbols if and only if it consumes the symbols and ends in a state in F.

As with finite automata, a nondeterministic push-down automaton (NPDA) is distinguished by a multivalued transition function: an NPDA can choose any of a set of new states and stack symbol replacements when faced with a given state, input, and top-of-stack symbol. If $\delta(q,\varepsilon, Z)$ is nonempty, N can also choose a new state and stack symbol replacement without inspecting or consuming its current input symbol. While we have seen that nondeterministic and deterministic finite automata are equally powerful, this correspondence does not carry over to push-down automata: there are context-free languages that are accepted by an NPDA but not by any DPDA.

The proof that CFGs and NPDAs are equivalent in expressive power is more complex than the corresponding proof for regular expressions and finite automata. The proof is also of limited practical importance for compiler construction; we do not present it here. While it is possible to create an NPDA for any CFL, simulating that NPDA may in some cases require exponential time to recognize strings in the language. (The $O(n^3)$ algorithms mentioned in Section 2.3 do not take the form of PDAs.) Practical programming languages can all be expressed with LL or LR grammars, which can be parsed with a (deterministic) PDA in linear time.

An LL(1) PDA is very simple. Because it makes decisions solely on the basis of the current input token and top-of-stack symbol, its state diagram is trivial. All but one of the transitions is a self-loop from the initial state to itself. A final transition moves from the initial state to a second, final state when it sees `$$` on the input and the stack. As we noted in Section 2.3.4, the state diagram for an SLR(1) or LALR(1) parser is substantially more interesting: it's the characteristic finite-state machine (CFSM). Full LR(1) parsers have similar machines, but usually with many more states, due to the need for path-specific look-ahead.

A little study reveals that if we define every state to be accepting, then the CFSM, without its stack, is a DFA that recognizes the grammar's *viable prefixes*. These are all the strings of grammar symbols that can begin a sentential form in the canonical (right-most) derivation of some string in the language, and that do not extend beyond the end of the handle. The algorithms to construct LL(1) and SLR(1) PDAs from suitable grammars were given in Sections 2.3.3 and 2.3.4.

2.4.3 Grammar and Language Classes

EXAMPLE 2.60
0^n1^n is not a regular language

As we noted in Section 2.1.2, a scanner is incapable of recognizing arbitrarily nested constructs. The key to the proof is to realize that we cannot count an arbitrary number of left-bracketing symbols with a finite number of states. Consider, for example, the problem of accepting the language 0^n1^n. Suppose there is a DFA M that accepts this language. Suppose further that M has m states. Now suppose we feed M a string of $m + 1$ zeros. By the *pigeonhole principle* (you can't distribute m objects among $p < m$ pigeonholes without putting at least two objects in some pigeonhole), M must enter some state q_i twice while scanning this string. Without loss of generality, let us assume it does so after seeing j zeros and again after seeing k zeros, for $j \neq k$. Since we know that M accepts the string 0^j1^j and the string 0^k1^k, and since it is in precisely the same state after reading 0^j and 0^k, we can deduce that M must also accept the strings 0^j1^k and 0^k1^j. Since these strings are not in the language, we have a contradiction: M cannot exist. ■

Within the family of context-free languages, one can prove similar theorems about the constructs that can and cannot be recognized using various parsing algorithms. Though almost all real parsers get by with a single token of look-ahead, it is possible in principle to use more than one, thereby expanding the set of grammars that can be parsed in linear time. In the top-down case we can redefine FIRST and FOLLOW sets to contain pairs of tokens in a more or less straightforward fashion. If we do this, however, we encounter a more serious version of the immediate error detection problem described in Section C-2.3.5. There we saw that the use of context-independent FOLLOW sets could cause us to overlook a syntax error until after we had needlessly predicted one or more epsilon productions. Context-specific FOLLOW sets solved the problem, but did not change the set of *valid* programs that could be parsed with one token of look-ahead. If we define LL(k) to be the set of all grammars that can be parsed predictively using the top-of-stack symbol and k tokens of look-ahead, then it turns out that for $k > 1$ we must adopt a context-specific notion of FOLLOW sets in order to parse correctly. The algorithm of Section 2.3.3, which is based on context-independent FOLLOW sets, is actually known as SLL (simple LL), rather than true LL. For $k = 1$, the LL(1) and SLL(1) algorithms can parse the same set of grammars. For $k > 1$, LL is strictly more powerful. Among the bottom-up parsers, the relationships among SLR(k), LALR(k), and LR(k) are somewhat more complicated, but extra look-ahead always helps.

EXAMPLE 2.61
Separation of grammar classes

Containment relationships among the classes of grammars accepted by popular linear-time algorithms appear in Figure C-2.36. The LR class (no suffix) contains every grammar G for which there exists a k such that $G \in$ LR(k); LL, SLL, SLR, and LALR are similarly defined. Grammars can be found in every region of the figure. Examples appear in Figure C-2.37. Proofs that they lie in the regions claimed are deferred to Exercise C-2.35. ■

For any context-free grammar G and parsing algorithm P, we say that G is a P grammar (e.g., an LL(1) grammar) if it can be parsed using that algorithm. By extension, for any context-free *language* L, we say that L is a P language if there exists a P grammar for L (this may not be the grammar we were given).

EXAMPLE 2.62
Separation of language classes

Containment relationships among the classes of languages accepted by the popular parsing algorithms appear in Figure C-2.38. Again, languages can be found in every region. Examples appear in Figure C-2.39; proofs are deferred to Exercise C-2.36. ■

It turns out that every context-free language that can be parsed deterministically has an SLR(1) grammar. In fact, any language that can parsed deterministically and in which no valid string can be extended to create another valid string (this is called the *prefix property*) has what is called an LR(0) grammar—one that can be parsed with no lookahead whatsoever! In the CFSM for such a grammar, any state containing an item with a • at the end will have no other item with a • in the middle. When such a state is reached, the parser can blindly reduce. If our scanner appends an explicit $$ marker at end-of-file, it is easy to see that our (augmented) language will have the prefix property, and an LR(0) grammar must exist. At the same time, LR(0) grammars tend to be large and unintuitive. Among other things, they must generally avoid any epsilon productions: if an item $A \longrightarrow \varepsilon\bullet$ shares a state with an item in which the dot precedes a terminal, we won't be able to tell whether to "recognize" ε without peeking ahead. Moreover, for any given grammar, LR(0) parsers have no space or time advantage over SLR(1) or LALR(1). As a result, LR(0) tends not to be used in practice.

The relationships among language classes are not as rich as the relationships among grammar classes. Most real programming languages can be parsed by any

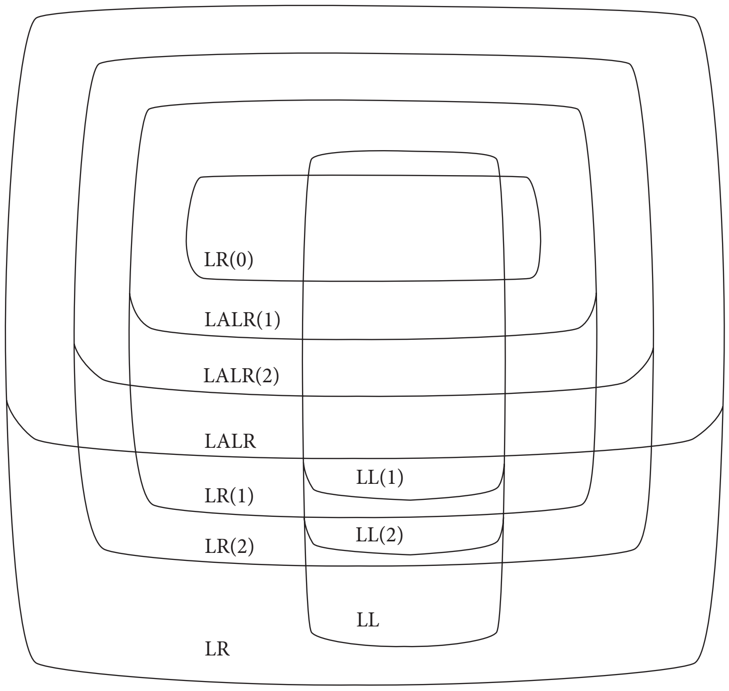

Figure 2.36 **Containment relationships among popular grammar classes.** Beyond the containments shown, SLL(k) is just inside LL(k), for $k \geq 2$; SLR(k) is just inside LALR(k), for $k \geq 1$.

LL(2) but not SLL:

$$\begin{aligned} S &\longrightarrow \texttt{a}\ A\ \texttt{a} \mid \texttt{b}\ A\ \texttt{b}\ \texttt{a} \\ A &\longrightarrow \texttt{b} \mid \varepsilon \end{aligned}$$

SLL(k) but not LL($k-1$):

$$S \longrightarrow \texttt{a}^{k-1}\ \texttt{b} \mid \texttt{a}^{k}$$

LR(0) but not LL:

$$\begin{aligned} S &\longrightarrow A\ \texttt{b} \\ A &\longrightarrow A\ \texttt{a} \mid \texttt{a} \end{aligned}$$

SLL(1) but not LALR:

$$\begin{aligned} S &\longrightarrow A\ \texttt{a} \mid B\ \texttt{b} \mid \texttt{c}\ C \\ C &\longrightarrow A\ \texttt{b} \mid B\ \texttt{a} \\ A &\longrightarrow D \\ B &\longrightarrow D \\ D &\longrightarrow \varepsilon \end{aligned}$$

SLL(k) and SLR(k) but not LR($k-1$):

$$\begin{aligned} S &\longrightarrow A\ \texttt{a}^{k-1}\ \texttt{b} \mid B\ \texttt{a}^{k-1}\ \texttt{c} \\ A &\longrightarrow \varepsilon \\ B &\longrightarrow \varepsilon \end{aligned}$$

LALR(1) but not SLR:

$$\begin{aligned} S &\longrightarrow \texttt{b}\ A\ \texttt{b} \mid A\ \texttt{c} \mid \texttt{a}\ \texttt{b} \\ A &\longrightarrow \texttt{a} \end{aligned}$$

LR(1) but not LALR:

$$\begin{aligned} S &\longrightarrow \texttt{a}\ C\ \texttt{a} \mid \texttt{b}\ C\ \texttt{b} \mid \texttt{a}\ D\ \texttt{b} \mid \texttt{b}\ D\ \texttt{a} \\ C &\longrightarrow \texttt{c} \\ D &\longrightarrow \texttt{c} \end{aligned}$$

Unambiguous but not LR:

$$S \longrightarrow \texttt{a}\ S\ \texttt{a} \mid \varepsilon$$

Figure 2.37 **Examples of grammars in various regions of Figure C-2.36.**

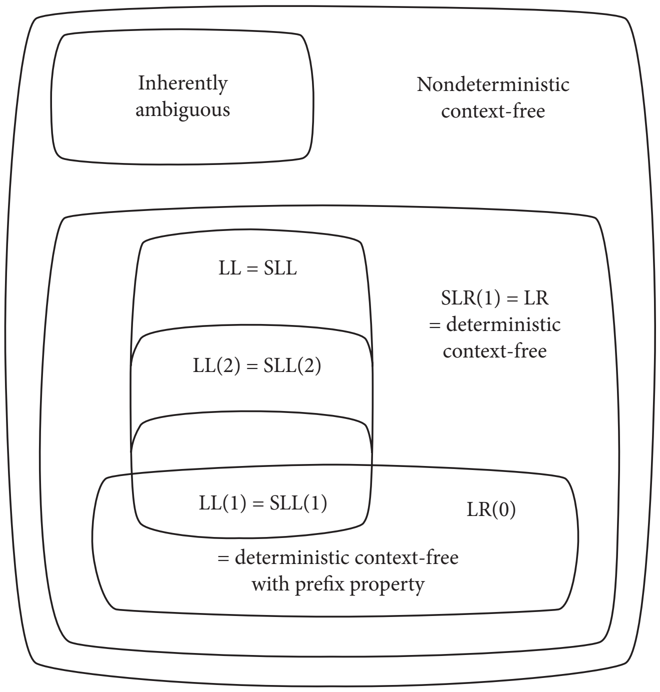
Inherently ambiguous
Nondeterministic context-free
LL = SLL
SLR(1) = LR = deterministic context-free
LL(2) = SLL(2)
LL(1) = SLL(1)
LR(0)
= deterministic context-free with prefix property

Figure 2.38 Containment relationships among popular language classes.

Nondeterministic language:
$\{\texttt{a}^n\texttt{b}^n\texttt{c} : n \ge 1\} \cup \{\texttt{a}^n\texttt{b}^{2n}\texttt{d} : n \ge 1\}$

Inherently ambiguous language:
$\{\texttt{a}^i\texttt{b}^j\texttt{c}^k : i = j \text{ or } j = k\,;\ i, j, k \ge 1\}$

Language with LL(k) grammar but no LL($k-1$) grammar:
$\{\texttt{a}^n(\texttt{b} \mid \texttt{c} \mid \texttt{b}^k\texttt{d})^n : n \ge 1\}$

Language with LR(0) grammar but no LL grammar:
$\{\texttt{a}^n\texttt{b}^n : n \ge 1\} \cup \{\texttt{a}^n\texttt{c}^n : n \ge 1\}$

Figure 2.39 Examples of languages in various regions of Figure C-2.38.

of the popular parsing algorithms, though the grammars are not always pretty, and special-purpose "hacks" may sometimes be required when a language is almost, but not quite, in a given class. The principal advantage of the more powerful parsing algorithms (e.g., full LR) is that they can parse a wider variety of grammars for a given language. In practice this flexibility makes it easier for the compiler writer to find a grammar that is intuitive and readable, and that facilitates the creation of semantic action routines.

CHECK YOUR UNDERSTANDING

55. What formal machine captures the behavior of a scanner? A parser?
56. State three ways in which a real scanner differs from the formal machine.
57. What are the formal components of a DFA?
58. Outline the algorithm used to construct a regular expression equivalent to a given DFA.
59. What is the inherent “big-O” complexity of parsing with a simulated NPDA? Why is this worse than the $O(n^3)$ time mentioned in Section 2.3?
60. How many states are there in an LL(1) PDA? An SLR(1) PDA? Explain.
61. What are the *viable prefixes* of a CFG?
62. Summarize the proof that a DFA cannot recognize arbitrarily nested constructs.
63. Explain the difference between LL and SLL parsing.
64. Is every LL(1) grammar also LR(1)? Is it LALR(1)?
65. Does every LR language have an SLR(1) grammar?
66. Why are there never any epsilon productions in an LR(0) grammar?
67. Why are the containment relationships among grammar classes more complex than those among language classes?

2 Programming Language Syntax

2.6 Exercises

2.31 Give an example of an erroneous program fragment in which consideration of semantic information (e.g., types) might help one make a good choice between two plausible "corrections" of the input.

2.32 Give an example of an erroneous program fragment in which the "best" correction would require one to "back up" the parser (i.e., to undo recent predictions/matches or shifts/reductions).

2.33 Extend your solution to exercise 2.21 to implement Wirth's syntax error recovery mechanism

(a) with global FOLLOW sets, as in Example C-2.45.

(b) with local FOLLOW sets, as in Example C-2.47.

(c) with avoidance of "starter symbol" deletion, as in Example C-2.48.

2.34 Extend your solution to exercise 2.21 to implement exception-based syntax error recovery, as in Example C-2.49.

2.35 Prove that the grammars in Figure C-2.37 lie in the regions claimed.

2.36 (Difficult) Prove that the languages in Figure C-2.39 lie in the regions claimed.

2.37 Prove that regular expressions and *left-linear grammars* are equally powerful. A left-linear grammar is a context-free grammar in which every right-hand side contains at most one nonterminal, and then only at the left-most end.

2 Programming Language Syntax

2.7 Explorations

2.46 Experiment with syntax errors in your favorite compiler. Feed the compiler deliberate errors and comment on the quality of the recovery or repair. How often does it do the "right thing"? How often does it generate cascading errors? Speculate as to what sort of recovery or repair algorithm it might be using.

2.47 Spelling mistakes (typos in keywords and identifiers) are a common source of syntax and static semantic errors. Identifying such errors—and guessing what the user meant to type—could result in significantly better error recovery. Discuss how you might go about incorporating spelling correction into some existing error recovery system. (Hint: You might want to consult Morgan's early paper on this subject [Mor70].)