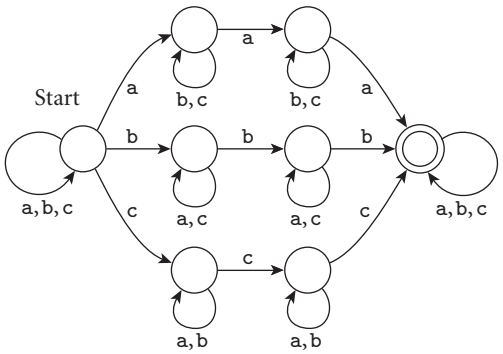

Figure 2.35 NFA corresponding to the DFA of Figure C-2.34.

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 $\delta : Q \times \Gamma \times \{\Sigma \cup \{\epsilon\}\} \rightarrow Q \times \Gamma^*$, 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 $\delta(q,\epsilon,Z)$ is defined, N moves to state r and replaces Z with α in the stack, where $(r,\alpha) = \delta(q,\epsilon,Z)$. In this case N does not consume its input symbol. If $\delta(q,\epsilon,Z)$ is undefined, N examines and consumes the current input symbol `a`. It then moves to state s and replaces Z with β, where $(s,\beta) = \delta(q,\mathtt{a},Z)$. 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 $\delta(q,\epsilon,Z)$ 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. ■

EXAMPLE 2.62
Separation of language classes

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). 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. ■

Note that every context-free language that can be parsed deterministically has an SLR(1) grammar. Moreover, any language that can be parsed deterministically and in which no valid string can be extended to create another valid string (this is called the *prefix property*) has an LR(0) grammar. If we restrict our attention to languages with an explicit `$$` marker at end-of-file, then they all have the prefix property, and therefore LR(0) grammars.

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

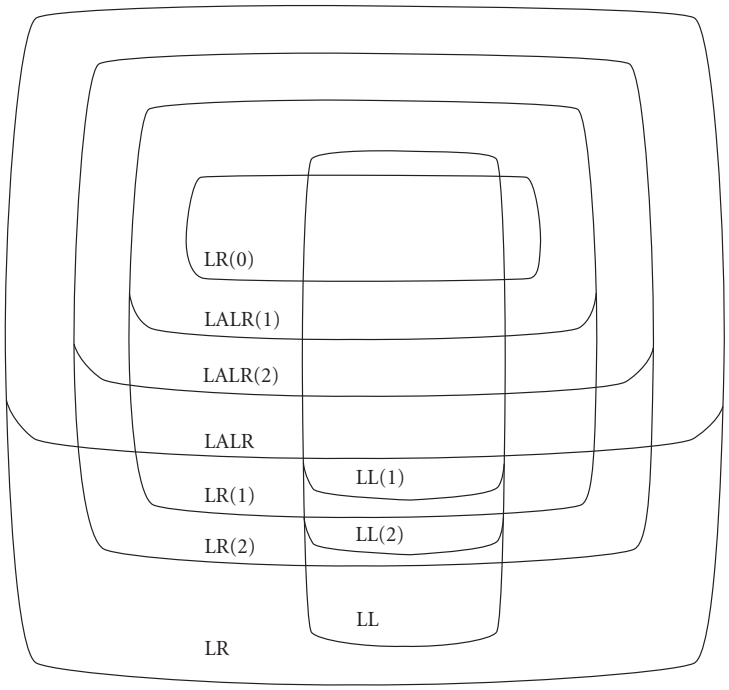

Figure 2.36 Containment relationships among popular grammar classes. In addition to the containments shown, SLL(k) is just inside LL(k), for $k \geq 2$, but has the same relationship to everything else, and SLR(k) is just inside LALR(k), for $k \geq 1$, but has the same relationship to everything else.

LL(2) but not SLL:

$S \longrightarrow \texttt{a}\ A\ \texttt{a} \mid \texttt{b}\ A\ \texttt{b}\ \texttt{a}$
$A \longrightarrow \texttt{b} \mid \epsilon$

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

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

LR(0) but not LL:

$S \longrightarrow A\ \texttt{b}$
$A \longrightarrow A\ \texttt{a} \mid \texttt{a}$

SLL(1) but not LALR:

$S \longrightarrow A\ \texttt{a} \mid B\ \texttt{b} \mid \texttt{c}\ C$
$C \longrightarrow A\ \texttt{b} \mid B\ \texttt{a}$
$A \longrightarrow D$
$B \longrightarrow D$
$D \longrightarrow \epsilon$

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

$S \longrightarrow A\ \texttt{a}^{k-1}\ \texttt{b} \mid B\ \texttt{a}^{k-1}\ \texttt{c}$
$A \longrightarrow \epsilon$
$B \longrightarrow \epsilon$

LALR(1) but not SLR:

$S \longrightarrow \texttt{b}\ A\ \texttt{b} \mid A\ \texttt{c} \mid \texttt{a}\ \texttt{b}$
$A \longrightarrow \texttt{a}$

LR(1) but not LALR:

$S \longrightarrow \texttt{a}\ C\ \texttt{a} \mid \texttt{b}\ C\ \texttt{b} \mid \texttt{a}\ D\ \texttt{b} \mid \texttt{b}\ D\ \texttt{a}$
$C \longrightarrow \texttt{c}$
$D \longrightarrow \texttt{c}$

Unambiguous but not LR:

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

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

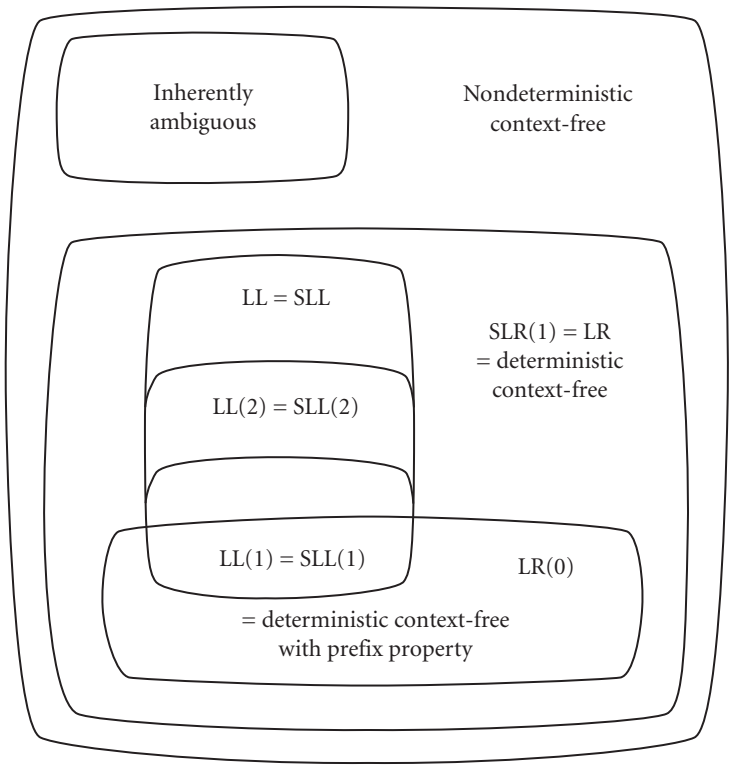

Figure 2.38 Containment relationships among popular language classes.

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

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

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

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

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

✓ CHECK YOUR UNDERSTANDING

56. What formal machine captures the behavior of a scanner? A parser?

57. State three ways in which a real scanner differs from the formal machine.

58. What are the formal components of a DFA?

59. Outline the algorithm used to construct a regular expression equivalent to a given DFA.

60. 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?

61. How many states are there in an LL(1) PDA? An SLR(1) PDA? Explain.

62. What are the *viable prefixes* of a CFG?

63. Summarize the proof that a DFA cannot recognize arbitrarily nested constructs.

64. Explain the difference between LL and SLL parsing.

65. Is every LL(1) grammar also LR(1)? Is it LALR(1)?

66. Does every LR language have an SLR(1) grammar?

67. Why are the containment relationships among grammar classes more complex than those among language classes?