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 2.12 (page 51) 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

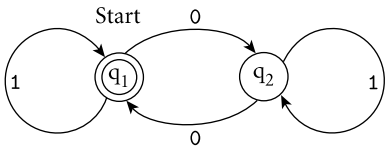

Figure 2.32 Minimal DFA for the language consisting of all strings of zeros and ones in which the number of zeros is even. Reprinted from Figure 2.10 in the main text.

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, 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 2.53
Formal DFA for `(1*01*0)*1*`

We can illustrate these definitions with an example. Consider the circles-and-arrows automaton of Figure 2.32 (reprinted from Figure 2.10 in the main text). This is the minimal DFA accepting strings of zeros and ones in which the number of zeros is even. $\Sigma = \{\texttt{0}, \texttt{1}\}$ is the machine's input alphabet. $Q = \{q_1, q_2\}$ is the set of states; q_1 is the initial state; $F = \{q_1\}$ is the set of final states. The transition function can be represented by a set of triples $\delta = \{(q_1, \texttt{0}, q_2), (q_1, \texttt{1}, q_1), (q_2, \texttt{0}, q_1), (q_2, \texttt{1}, q_2)\}$. In each triple $(q_i, \texttt{a}, q_j)$, $\delta(q_i, \texttt{a}) = q_j$.

Given the constructions of Examples 2.10 and 2.12, 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 "even number of zeros" example (Figure 2.32). More formal and general treatment of all the regular language constructions can be found in standard automata theory texts [HMU01, Sip97].

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. More precisely, we begin with a set of simple regular expressions that describe the transition function, δ. For all states i, we define

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

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, $r_{ii}^0 = \epsilon$.

Similarly, for $i \neq j$, we define

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

where $\{\mathtt{a}_1 \mid \mathtt{a}_2 \mid \ldots \mid \mathtt{a}_m\} = \{\mathtt{a} \mid \delta(q_i, \mathtt{a}) = q_j\}$ 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: no intervening states.

EXAMPLE 2.54

Reconstructing the regular expression for a 2-state DFA

In our tiny example DFA, $r_{11}^0 = r_{22}^0 = \mathtt{1} \mid \epsilon$, and $r_{12}^0 = r_{21}^0 = \mathtt{0}$. For $k > 0$, the r_{ij}^k expressions will generally generate multicharacter strings. At each step of the dynamic programming algorithm, we let $r_{ij}^k = r_{ij}^{k-1} \mid r_{ik}^{k-1}\, r_{kk}^{k-1}\star\, r_{kj}^{k-1}$. 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. In the first inductive step in our example,

$$\begin{aligned}
r_{11}^1 &= (\,\mathtt{1} \mid \epsilon\,) \mid (\,\mathtt{1} \mid \epsilon\,)(\,\mathtt{1} \mid \epsilon\,)^*(\,\mathtt{1} \mid \epsilon\,) \\
r_{12}^1 &= \mathtt{0} \mid (\,\mathtt{1} \mid \epsilon\,)(\,\mathtt{1} \mid \epsilon\,)^*\,\mathtt{0} \\
r_{22}^1 &= (\,\mathtt{1} \mid \epsilon\,) \mid \mathtt{0}(\,\mathtt{1} \mid \epsilon\,)^*\,\mathtt{0} \\
r_{21}^1 &= \mathtt{0} \mid \mathtt{0}(\,\mathtt{1} \mid \epsilon\,)^*(\,\mathtt{1} \mid \epsilon\,)
\end{aligned}$$

In the second and final inductive step,

$$\begin{aligned}
r_{11}^2 = {} & ((\,\mathtt{1} \mid \epsilon\,) \mid (\,\mathtt{1} \mid \epsilon\,)(\,\mathtt{1} \mid \epsilon\,)^*(\,\mathtt{1} \mid \epsilon\,)) \mid \\
& (\,\mathtt{0} \mid (\,\mathtt{1} \mid \epsilon\,)(\,\mathtt{1} \mid \epsilon\,)^*\,\mathtt{0}\,) \\
& ((\,\mathtt{1} \mid \epsilon\,) \mid \mathtt{0}(\,\mathtt{1} \mid \epsilon\,)^*\,\mathtt{0}\,)^* \\
& (\,\mathtt{0} \mid \mathtt{0}(\,\mathtt{1} \mid \epsilon\,)^*(\,\mathtt{1} \mid \epsilon\,))
\end{aligned}$$

Since F has a single member (q_1), this expression is our final answer. Obviously it isn't in a minimal form, but it is correct. ■

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 $(s, \beta) = \delta(q, \texttt{a}, Z)$. 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, 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 2.3.3 (page 85), 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 (rightmost) 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 2.3.2 and 2.3.3.

2.4.3 Grammar and Language Classes

EXAMPLE 2.55
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 2.3.4. 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 2.3.2, 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 2.56
Separation of grammar classes

Containment relationships among the classes of grammars accepted by popular linear-time algorithms appear in Figure 2.33. The LR class (no suffix) contains every grammar G for which there exists a k such that $G \in \text{LR}(k)$; LL, SLL, SLR, and LALR are similarly defined. Grammars can be found in every re-

gion of the figure. Examples appear in Figure 2.34. Proofs that they lie in the regions claimed are deferred to Exercise 2.26.

EXAMPLE 2.57
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 2.35. Again, languages can be found in every region. Examples appear in Figure 2.36; proofs are deferred to Exercise 2.27.

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.

✓ 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 an 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)?

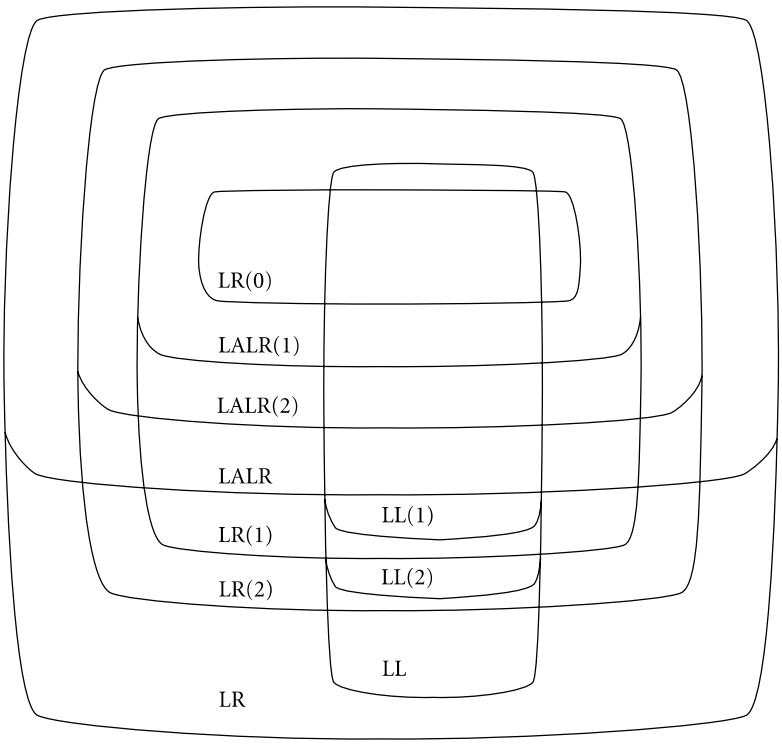

Figure 2.33 **Containment relationships among popular grammar classes.** In addition to the containments shown, SLL(k) is just inside LL(k), for $k \geq 2$, but has the same relationship to everything else, and SLR(k) is just inside LALR(k), for $k \geq 1$, but has the same relationship to everything 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 2.34 Examples of grammars in various regions of Figure 2.33.

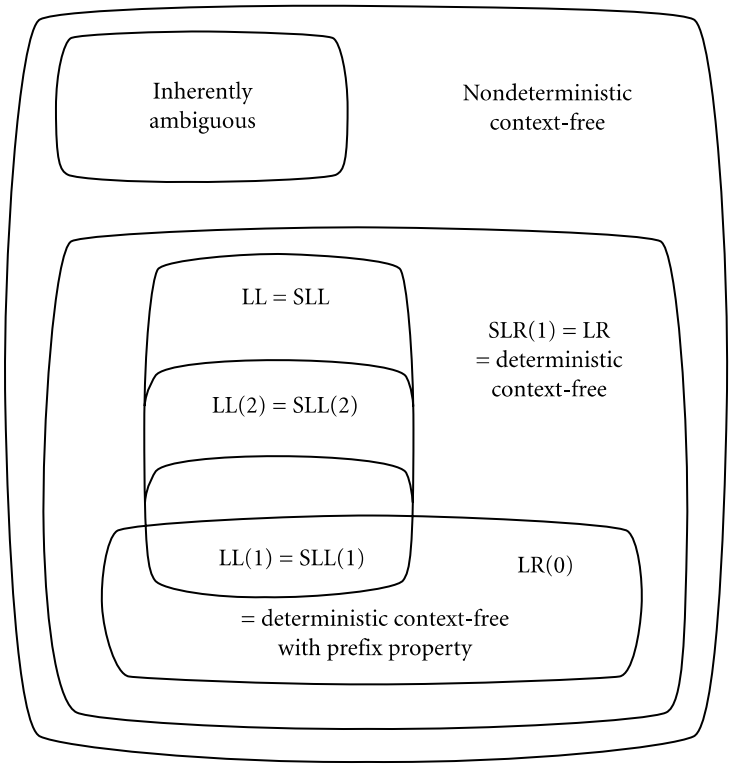

Figure 2.35 Containment relationships among popular language classes.

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

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

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

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

Figure 2.36 Examples of languages in various regions of Figure 2.35.

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

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