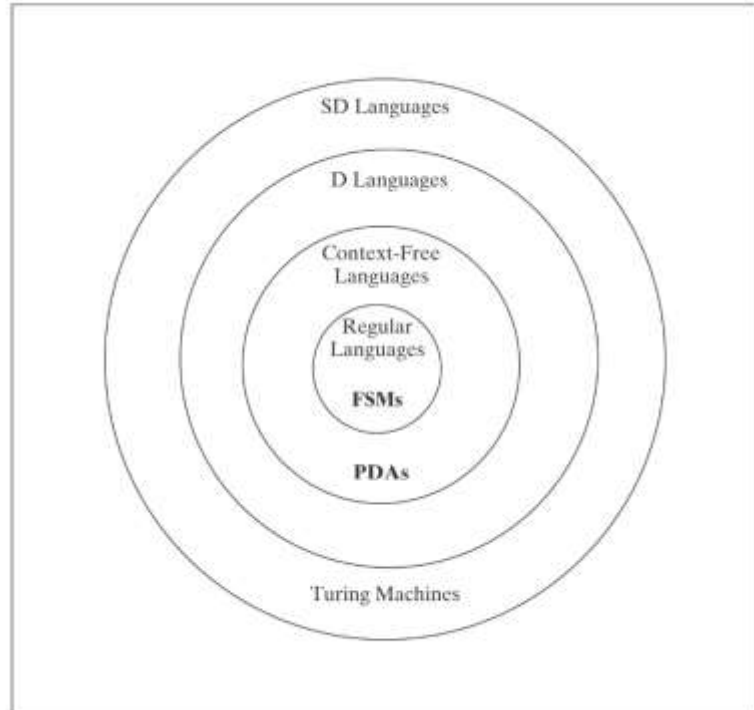


Part III: Context-Free Languages and Pushdown Automata

In this section, we move out one level and explore the class of context-free languages.

This class is important. For most programming languages, the set of syntactically legal statements is (except possibly for type checking) a context-free language. The set of well-formed Boolean queries is a context-free language. A great deal of the syntax of English can be described in the context-free framework that we are about to discuss. To describe these languages, we need more power than the regular language definition allows. For example, to describe both programming language statements and Boolean queries requires the ability to specify that parentheses be balanced. Yet we showed in Section 8.4 that it is not possible to define a regular language that contains exactly the set of strings of balanced parentheses.



We will begin our discussion of the context-free languages by defining a grammatical formalism that can be used to describe every language in the class (which, by the way, does include the language of balanced parentheses). Then, in Chapter 12, we will return to the question of defining machines that can accept strings in the language. At that point, we'll see that the pushdown automaton, an NDFSM augmented with a single stack, can accept exactly the class of context-free languages that we are about to describe. In Chapter 13, we will see that the formalisms that we have presented stop short of the full power that is provided by a more general computational model. So we'll see that there are straightforward languages that are not context-free. But, because of the restrictions that the context-free formalism imposes, it will turn out to be possible to define algorithms that perform at least the most basic operations on context-free languages, including deciding whether a string is in a language. We'll summarize those algorithms in Chapters 14 and 15.

The theory that we are about to present for the context-free languages is not as straightforward and elegant as the one that we have just described for the regular languages. We'll see, for example, that there doesn't exist an algorithm that compares two pushdown automata to see if they are equivalent. Given an arbitrary context-free grammar G , there doesn't exist a linear-time algorithm that decides whether a string w is an element of $L(G)$. But there does exist such an algorithm if we restrict our attention to a useful subset of the context-free languages. The context-free languages are not closed under many common operations like intersection and complement.

On the other hand, because the class of context-free languages includes most programming languages, query languages, and a host of other languages that we use daily to communicate with computers, it is worth taking the time to work through the theory that is presented here, even though it is less clear than the one we were able to build in Part II.

11 Context-Free Grammars

We saw, in our discussion of the regular languages in Part II, that there are substantial advantages to using descriptive frameworks (in that case, FSMs, regular expressions, and regular grammars) that offer less power and flexibility than a general purpose programming language provides. Because the frameworks were restrictive, we were able to describe a large class of useful operations that could be performed on the languages that we defined.

We will begin our discussion of the context-free languages with another restricted formalism, the context-free grammar. But before we define it, we will pause and answer the more general question, “What is a grammar?”

11.1 Introduction to Rewrite Systems and Grammars

We’ll begin with a very general computational model: Define a *rewrite system* (also called a *production system* or a *rule-based system*) to be a list of rules and an algorithm for applying them. Each rule has a left-hand side and a right-hand side. For example, the following could be rewrite-system rules:

$$\begin{aligned} S &\rightarrow aSb \\ aS &\rightarrow \varepsilon \\ aSb &\rightarrow bSabbSa \end{aligned}$$

In the discussion that follows, we will focus on rewrite system that operate on strings. But the core ideas that we will present can be used to define rewrite systems that operate on richer data structures. Of course, such data structures can be represented as strings, but the power of many practical rule-based systems comes from their ability to manipulate other structures directly.

Expert systems, § 774, are programs that perform tasks in domains like engineering, medicine, and business, that require expertise when done by people. Many kinds of expertise can naturally be modeled as sets of condition/action rules. So many expert systems are built using tools that support rule-based programming.

Rule based systems are also used to model business practices, § 774, and as the basis for reasoning about the behavior of nonplayer characters in computer games, § 791.

When a rewrite system R is invoked on some initial string w , it operates as follows:

simple-rewrite(R : rewrite system, w : initial string) =

1. Set *working-string* to w .
2. Until told by R to halt do:
 - 2.1. Match the left-hand side of some rule against some part of *working-string*.
 - 2.2. Replace the matched part of *working-string* with the right-hand side of the rule that was matched.
3. Return *working-string*.

If *simple-rewrite*(R , w) can return some string s then we’ll say that R can *derive* s from w or that there exists a *derivation* in R of s from w .

Rewrite systems can model natural growth processes, as occur, for example, in plants. In addition, evolutionary algorithms can be applied to rule sets. Thus rewrite systems can model evolutionary processes, § 809.

We can define a particular *rewrite-system formalism* by specifying the form of the rules that are allowed and the algorithm by which they will be applied. In most of the rewrite-system formalisms that we will consider, a rule is simply a pair of strings. If the string on the left-hand side matches, it is replaced by the string on the right-hand side.

But more flexible forms are also possible. For example, variables may be allowed. Let x be a variable. Then consider the rule:

$$axa \rightarrow aa$$

This rule will squeeze out whatever comes between a pair of a 's.

Another useful form allows regular expressions as left-hand sides. If we do that, we can write rules like the following, which squeezes out b 's between a 's:

$$ab^*ab^*a \rightarrow aaa$$

The extended form of regular expressions that is supported in programming languages like Perl is often used to write substitution rules. [C 792](#).

In addition to describing the form of its rules, a rewrite-system formalism must describe how its rules will be applied. In particular, a rewrite-system formalism will define the conditions under which *simple-rewrite* will halt and the method by which it will choose a match in step 2.1. For example, one rewrite-system formalism might specify that any rule that matches may be chosen. A different formalism might specify that the rules have to be tried in the order in which they are written, with the first one that matches being the one that is chosen next.

Rewrite systems can be used to define functions. In this case, we write rules that operate on an input string to produce the required output string. And rewrite systems can be used to define languages. In this case, we define a unique start symbol. The rules then apply and we will say that the language L that is generated by the system is exactly the set of strings, over L 's alphabet, that can be derived by *simple-rewrite* from the start symbol.

A rewrite-system formalism can be viewed as a programming language and some such languages turn out to be useful. For example, Prolog, [C 762](#), supports a style of programming called logic programming. A logic program is a set of rules that correspond to logical statements of the form A if B . The interpreter for a logic program reasons backwards from a goal (such as A), chaining rules together until each right-hand side has been reduced to a set of facts (axioms) that are already known to be true.

The study of rewrite systems has played an important role in the development of the theory of computability. We'll see in Part V that there exist rewrite-system formalisms that have the same computational power as the Turing machine, both with respect to computing functions and with respect to defining languages. In the rest of our discussion in this chapter, however, we will focus just on their use to define languages.

A rewrite system that is used to define a language is called a **grammar**. If G is a grammar, let $L(G)$ be the language that G generates. Like every rewrite system, every grammar contains a list (almost always treated as a set, i.e., as an unordered list) of rules. Also, like every rewrite system, every grammar works with an alphabet, which we can call V . In the case of grammars, we will divide V into two subsets:

- a **terminal alphabet**, generally called Σ , which contains the symbols that make up the strings in $L(G)$, and
- a **nonterminal alphabet**, the elements of which will function as working symbols that will be used while the grammar is operating. These symbols will disappear by the time the grammar finishes its job and generates a string.

One final thing is required to specify a grammar. Each grammar has a unique start symbol, often called S .

Grammars can be used to describe phenomena as different as English, [C 743](#), programming languages like Java, [C 664](#), music, [C 776](#), dance, [C 808](#), the growth of living organisms, [C 809](#), and the structure of RNA, [C 737](#).

A **grammar formalism** (like any rewrite-system formalism) specifies the form of the rules that are allowed and the algorithm by which they will be applied. The grammar formalisms that we will consider vary in the form of the rules that they allow. With one exception (Lindenmayer systems, which we'll describe in Section 24.4), all of the grammar formalisms that we will consider include a control algorithm that ignores rule order. Any rule that matches may be applied next.

To generate strings in $L(G)$, we invoke *simple-rewrite* (G, S). *Simple-rewrite* will begin with S and will apply the rules of G , which can be thought of (given the control algorithm we just described) as licenses to replace one string by another. At each step of one of its derivations, some rule whose left-hand side matches somewhere in *working-string* is selected. The substring that matched is replaced by the rule's right-hand side, generating a new value for *working string*.

Grammars can be used to define languages that, in turn, define sets of things that don't look at all like strings. For example, SVG, \mathbb{C} 808, is a language that is used to describe two-dimensional graphics. SVG can be described with a context-free grammar.

We will use the symbol \Rightarrow to indicate steps in a derivation. So, for example, suppose that G has the start symbol S and the rules $S \rightarrow aSb$, $S \rightarrow bSa$, and $S \rightarrow \epsilon$. Then a derivation could begin with:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow \dots$$

At each step, it is possible that more than one rule's left-hand side matches the working string. It is also possible that a rule's left-hand side matches the working string in more than one way. In either case, there is a derivation corresponding to each alternative. It is precisely the existence of these choices that enables a grammar to generate more than one string.

Continuing with our example, there are three choices at the next step:

$$\begin{array}{ll} S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb & \text{(using the first rule),} \\ S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabSabb & \text{(using the second rule), and} \\ S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb & \text{(using the third rule).} \end{array}$$

The derivation process may end whenever one of the following things happens:

- (1) The working string no longer contains any nonterminal symbols (including, as a special case, when the working string is ϵ), or
- (2) There are nonterminal symbols in the working string but none of them appears on the left-hand side of any rule in the grammar. For example, if the working string were $AaBb$, this would happen if no rule had A or B as its left-hand side.

In the first case, but not the second, we say that the working string is **generated** by the grammar. Thus, the **language** that a grammar generates includes only strings over the terminal alphabet, i.e., strings in Σ^* . In the second case, we have a blocked or non-terminated derivation but no generated string.

It is also possible that, in a particular case, neither (1) nor (2) is achieved. Suppose, for example, that a grammar contained only the rules $S \rightarrow Ba$ and $B \rightarrow bB$, with S the start symbol. Then all derivations proceed in the following way:

$$S \Rightarrow Ba \Rightarrow bBa \Rightarrow bbBa \Rightarrow bbbBa \Rightarrow bbbbBa \Rightarrow \dots$$

The working string is always rewriteable (in only one way, as it happens), and so this grammar can produce no terminated derivations consisting entirely of terminal symbols (i.e., generated strings). Thus this grammar generates the language \emptyset .

11.2 Context-Free Grammars and Languages

We've already seen our first specific grammar formalism. In Chapter 7, we defined a regular grammar to be one in which every rule must:

- have a left-hand side that is a single nonterminal, and
- have a right-hand side that is ϵ or a single terminal or a single terminal followed by a single nonterminal.

We now define a *context-free grammar* (or CFG) to be a grammar in which each rule must:

- have a left-hand side that is a single nonterminal, and
- have a right-hand side.

To simplify the discussion that follows, define an *A* rule, for any nonterminal symbol *A*, to be a rule whose left-hand side is *A*.

Next we must define a control algorithm of the sort we described at the end of the last section. A derivation will halt whenever no rule's left-hand side matches against *working-string*. At every step, any rule that matches may be chosen.

Context-free grammar rules may have any (possibly empty) sequence of symbols on the right-hand side. Because the rule format is more flexible than it is for regular grammars, the rules are more powerful. We will soon show some examples of languages that can be generated with context-free grammars but that can not be generated with regular ones.

All of the following are allowable context-free grammar rules (assuming appropriate alphabets):

$$\begin{aligned} S &\rightarrow aSb \\ S &\rightarrow \epsilon \\ T &\rightarrow T \\ S &\rightarrow aSbbTT \end{aligned}$$

The following are not allowable context-free grammar rules:

$$\begin{aligned} ST &\rightarrow aSb \\ a &\rightarrow aSb \\ \epsilon &\rightarrow a \end{aligned}$$

The name for these grammars, "context-free," makes sense because, using these rules, the decision to replace a nonterminal by some other sequence is made without looking at the context in which the nonterminal occurs. In Chapters 23 and 24 we will consider less restrictive grammar formalisms in which the left-hand sides of the rules may contain several symbols. For example, the rule $aSa \rightarrow aTa$ would be allowed. This rule says that *S* can be replaced by *T* when it is surrounded by *a*'s. One of those formalisms is called "context-sensitive" because its rules allow context to be considered.

Programming language syntax is typically described using context-free grammars, as we'll see below and in $\text{C } 664$.

Formally, a context-free grammar *G* is a quadruple (V, Σ, R, S) , where:

- *V* is the rule alphabet, which contains nonterminals (symbols that are used in the grammar but that do not appear in strings in the language) and terminals,
- Σ (the set of terminals) is a subset of *V*,
- *R* (the set of rules) is a finite subset of $(V - \Sigma) \times V^*$, and
- *S* (the start symbol) can be any element of $V - \Sigma$.

Let's now look at an important property that gives context-free grammars the power to define languages that aren't regular. A rule in a grammar G is **self-embedding** iff it is of the form $X \rightarrow w_1 Y w_2$, where $Y \Rightarrow_{G^*} w_3 X w_4$ and both $w_1 w_3$ and $w_4 w_2$ are in Σ^+ . A grammar is self-embedding iff it contains at least one self-embedding rule. So now we require that a nonempty string be generated on each side of the nested X . The grammar we presented for *Bal* is self-embedding because it contains the rule $S \rightarrow (S)$. The grammar we presented for $A^n B^n$ is self-embedding because it contains the rule $S \rightarrow a S b$. The presence of a rule like $S \rightarrow a S$ does not by itself make a grammar self-embedding. But the rule $S \rightarrow a T$ is self-embedding in any grammar G that also contains the rule $T \rightarrow S b$, since $S \rightarrow a T$ and $T \Rightarrow_{G^*} S b$. Self-embedding grammars are able to define languages like *Bal*, $A^n B^n$, and others whose strings must contain pairs of matching regions, often of the form $uv^i xy^j z$. No regular language can impose such a requirement on its strings.

The fact that a grammar G is self-embedding does not guarantee that $L(G)$ isn't regular. There might be a different grammar G' that also defines $L(G)$ and that is not self-embedding. For example $G_1 = (\{S, a\}, \{a\}, \{S \rightarrow \varepsilon, S \rightarrow a, S \rightarrow a S a\}, S)$ is self-embedding, yet it defines the regular language a^* . However, we note the following two important facts:

- If a grammar G is not self-embedding then $L(G)$ is regular. Recall that our definition of regular grammars did not allow self-embedding.
- If a language L has the property that every grammar that defines it is self-embedding, then L is not regular.

The rest of the grammars that we will present in this chapter are self-embedding.

Example 11.3 Even Length Palindromes

Consider $\text{PalEven} = \{ww^R : w \in \{a, b\}^*\}$, the language of even-length palindromes of a 's and b 's. We showed in Example 8.11 that PalEven is not regular. But it is context-free because it can be generated by the grammar $G = (\{S, a, b\}, \{a, b\}, R, S)$, where:

$$R = \{ S \rightarrow a S a \\ S \rightarrow b S b \\ S \rightarrow \varepsilon \}.$$

Example 11.4 Equal Numbers of a 's and b 's

Let $L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$. We showed in Example 8.14 that L is not regular. But it is context-free because it can be generated by the grammar $G = (\{S, a, b\}, \{a, b\}, R, S)$, where:

$$R = \{ S \rightarrow a S b \\ S \rightarrow b S a \\ S \rightarrow S S \\ S \rightarrow \varepsilon \}.$$

These simple examples are interesting because they capture, in a couple of lines, the power of the context-free grammar formalism. But our real interest in context-free grammars comes from the fact that they can describe useful and powerful languages that are substantially more complex.

It quickly becomes apparent, when we start to build larger grammars, that we need a more flexible grammar-writing notation. We'll use the following two extensions when they are helpful:

- The symbol $|$ should be read as "or". It allows two or more rules to be collapsed into one. So the following single rule is equivalent to the four rules we wrote in Example 11.4:

$$S \rightarrow a S b \mid b S a \mid S S \mid \varepsilon$$

- We often require nonterminal alphabets that contain more symbols than there are letters. To solve that problem, we will allow a nonterminal symbol to be any sequence of characters surrounded by angle brackets. So `<program>` and `<variable>` could be nonterminal symbols using this convention.

BNF (or Backus Naur form) is a widely used grammatical formalism that exploits both of these extensions. It was created in the late 1950s as a way to describe the programming language ALGOL 60. It has since been extended and several dialects developed $\text{C } 664$.

Example 11.5 BNF for a Small Java Fragment

Because BNF was originally designed when only a small character set was available, it uses the three symbol sequence `::=` in place of \rightarrow . The following BNF-style grammar describes a highly simplified and very small subset of Java:

```
<block> ::= {<stmt-list>} | {}
<stmt-list> ::= <stmt> | <stmt-list> <stmt>
<stmt> ::= <block> | while (<cond>) <stmt> | if (<cond>) <stmt> |
do <stmt> while (<cond>); | <assignment-stmt>; |
return | return <expression> | <method-invocation>;
```

The rules of this grammar make it clear that the following block may be legal in Java (assuming that the appropriate declarations have occurred):

```
{    while (x < 12) {
        hippo.pretend(x);
        x = x + 2;
    }}
```

On the other hand, the following block is not legal:

```
{    while x < 12}) (
        hippo.pretend(x);
        x = x + 2;
    }}
```

Many other kinds of practical languages are also context-free. For example, HTML can be described with a context-free grammar using a BNF-style grammar. $\text{C } 805$.

Example 11.6 A Fragment of an English Grammar

Much of the structure of an English sentence can be described by a (large) context-free grammar. For historical reasons, linguistic grammars typically use a slightly different notational convention. Nonterminals will be written as strings whose first symbol is an upper case letter. So the following grammar describes a tiny fragment of English. The symbol *NP* will derive noun phrases; the symbol *VP* will derive verb phrases:

```
S → NP VP
NP → the Nominal | a Nominal | Nominal / ProperNoun | NP PP
Nominal → N | Adjs N
N → cat | dogs | bear | girl | chocolate | rifle
ProperNoun → Chris | Fluffy
Adjs → Adj Adjs | Adj
Adj → young | older | smart
VP → V | V NP | VP PP
V → like | likes | thinks | shot | smells
```


$PP \rightarrow Prep NP$
 $Prep \rightarrow with$

Is English (or German or Chinese) really context-free? $\text{C } 747$.

11.3 Designing Context-Free Grammars

In this section, we offer a few simple strategies for designing straightforward context-free grammars. Later we'll see that some grammars are better than others (for various reasons) and we'll look at techniques for finding "good" grammars. For now, we will focus on finding some grammar.

The most important rule to remember in designing a context-free grammar to generate a language L is the following:

- If L has the property that every string in it has two regions and those regions must bear some relationship to each other (such as being of the same length), then the two regions must be generated in tandem. Otherwise, there is no way to enforce the necessary constraint.

Keeping that rule in mind, there are two simple ways to generate strings:

- To generate a string with multiple regions that must occur in some fixed order but do not have to correspond to each other, use a rule of the form:

$A \rightarrow BC \dots$

This rule generates two regions, and the grammar that contains it will then rely on additional rules to describe how to form a B region and how to form a C region. Longer rules, like $A \rightarrow BCDE$, can be used if additional regions are necessary.

The outside-in structure of context-free grammars makes them well suited to describing physical things, like RNA molecules, that fold. $\text{C } 737$.

- To generate a string with two regions that must occur in some fixed order and that must correspond to each other, start at the outside edges of the string and generate toward the middle. If there is an unrelated region in between the related ones, it must be generated after the related regions have been produced.

Example 11.7 Concatenating Independent Sublanguages

Let $L = \{a^n b^n c^m : n, m \geq 0\}$. Here, the c^m portion of any string in L is completely independent of the $a^n b^n$ portion, so we should generate the two portions separately and concatenate them together. So let $G = (\{S, N, C, a, b, c\}, \{a, b, c\}, R, S)$ where:

$R = \{ S \rightarrow NC$	$/*$ generate the two independent portions.
$N \rightarrow aNb$	$/*$ generate the $a^n b^n$ portion, from the outside in.
$N \rightarrow \epsilon$	
$C \rightarrow cC$	$/*$ generate the c^m portion.
$C \rightarrow \epsilon \}$.	

Example 11.8 The Kleene Star of a Language

Let $L = \{a^{n_1} b^{n_1} a^{n_2} b^{n_2} \dots a^{n_k} b^{n_k} : k \geq 0 \text{ and } \forall i (n_i \geq 0)\}$. For example, the following strings are in L : ϵ , $abab$, $aabbbaabbbbabab$. Note that $L = \{a^n b^n : n \geq 0\}^*$, which gives a clue how to write the grammar we need. We know how to produce individual elements of $\{a^n b^n : n \geq 0\}$, and we know how to concatenate regions together. So a solution

is $G = (\{S, M, a, b\}, \{a, b\}, R, S)$ where:

$$\begin{array}{ll}
 R = \{ S \rightarrow MS & /* \text{each } M \text{ will generate one } \{a^n b^n : n \geq 0\} \text{ region.} \\
 \quad S \rightarrow \varepsilon & \\
 \quad M \rightarrow aMb & /* \text{generate one region.} \\
 \quad M \rightarrow \varepsilon \}. &
 \end{array}$$

11.4 Simplifying Context-Free Grammars ✪

In this section, we present two algorithms that may be useful for simplifying context-free grammars.

Consider the grammar $G = (\{S, A, B, C, D, a, b\}, \{a, b\}, R, S)$, where:

$$\begin{array}{l}
 R = \{ S \rightarrow AB \mid AC \\
 \quad A \rightarrow aAb \mid \varepsilon \\
 \quad B \rightarrow bA \\
 \quad C \rightarrow bCa \\
 \quad D \rightarrow AB \}.
 \end{array}$$

G contains two useless variables: C is useless because it is not able to generate any strings in Σ^* . (Every time a rule is applied to a C , a new C is added.) D is useless because it is unreachable, via any derivation, from S . So any rules that mention either C or D can be removed from G without changing the language that is generated. We present two algorithms, one to find and remove variables like C that are unproductive, and one to find and remove variables like D that are unreachable.

Given a grammar $G = (V, \Sigma, R, S)$, we define *removeunproductive*(G) to create a new grammar G' , where $L(G') = L(G)$ and G' does not contain any unproductive symbols. Rather than trying to find the unproductive symbols directly, *removeunproductive* will find and mark all the productive ones. Any that are left unmarked at the end are unproductive. Initially, all terminal symbols will be marked as productive since each of them generates a terminal string (itself). A nonterminal symbol will be marked as productive when it is discovered that there is at least one way to rewrite it as a sequence of productive symbols. So *removeunproductive* effectively moves backwards from terminals, marking nonterminals along the way.

removeunproductive(G : CFG) =

1. $G' = G$.
2. Mark every nonterminal symbol in G' as unproductive.
3. Mark every terminal symbol in G' as productive.
4. Until one entire pass has been made without any new symbol being marked do:
 - For each rule $X \rightarrow \alpha$ in R do:
 - If every symbol in α has been marked as productive and X has not yet been marked as productive, then mark X as productive.
5. Remove from $V_{G'}$ every unproductive symbol.
6. Remove from $R_{G'}$ every rule with an unproductive symbol on either the left-hand side or the right-hand side.
7. Return G' .

Removeunproductive must halt because there is only some finite number of nonterminals that can be marked as productive. So the maximum number of times it can execute step 4 is $|V - \Sigma|$. Clearly $L(G') \subseteq L(G)$ since G' can produce no derivations that G could not have produced. And $L(G') = L(G)$ because the only derivations that G can perform but G' cannot are those that do not end with a terminal string.

Notice that it is possible that S is unproductive. This will happen precisely in case $L(G) = \emptyset$. We will use this fact in Section 14.1.2 to show the existence of a procedure that decides whether or not a context-free language is empty.

Next we'll define an algorithm for getting rid of unreachable symbols like D in the grammar we presented above. Given a grammar $G = (V, \Sigma, R, S)$, we define *removeunreachable*(G) to create a new grammar G' , where $L(G') = L(G)$ and G' does not contain any unreachable nonterminal symbols. What *removeunreachable* does is to move forward from S , marking reachable symbols along the way.

removeunreachable(G : CFG) =

1. $G' = G$.
2. Mark S as reachable.
3. Mark every other nonterminal symbol as unreachable.
4. Until one entire pass has been made without any new symbol being marked do:
 For each rule $X \rightarrow \alpha A \beta$ (where $A \in V - \Sigma$ and $\alpha, \beta \in V^*$) in R do:
 If X has been marked as reachable and A has not, then mark A as reachable.
5. Remove from $V_{G'}$ every unreachable nonterminal symbol.
6. Remove from $R_{G'}$ every rule with an unreachable symbol on the left-hand side.
7. Return G' .

Removeunreachable must halt because there is only some finite number of nonterminals that can be marked as reachable. So the maximum number of times it can execute step 4 is $|V - \Sigma|$. Clearly $L(G') \subseteq L(G)$ since G' can produce no derivations that G could not have produced. And $L(G') = L(G)$ because every derivation that can be produced by G can also be produced by G' .

11.5 Proving That a Grammar is Correct ✪

In the last couple of sections, we described some techniques that are useful in designing context-free languages and we argued that the grammars that we built were correct (i.e., that they correctly describe languages with certain properties). But, given some language L and a grammar G , can we actually prove that G is correct (i.e., that it generates exactly the strings in L)? To do so, we need to prove two things:

1. G generates only strings in L , and
2. G generates all the strings in L .

The most straightforward way to do step 1 is to imagine the process by which G generates a string as the following loop (a version of *simple-rewrite*, using st in place of *working-string*):

1. $st = S$.
2. Until no nonterminals are left in st do:
 Apply some rule in R to st .
3. Output st .

Then we construct a loop invariant I and show that:

- I is true when the loop begins,
- I is maintained at each step through the loop (i.e., by each rule application), and
- $I \wedge (st \text{ contains only terminal symbols}) \rightarrow st \in L$.

Step 2 is generally done by induction on the length of the generated strings.

Example 11.9 The Correctness of the $A^n B^n$ Grammar

In Example 11.2, we considered the language $A^n B^n$. We built for it the grammar $G = (\{S, a, b\}, \{a, b\}, R, S)$, where:

$$R = \{ S \rightarrow aSb \quad (1)$$

$$S \rightarrow \varepsilon \}. \quad (2)$$

We now show that G is correct. We first show that every string w in $L(G)$ is in A^nB^n : Let st be the working string at any point in a derivation in G . We need to define I so that it captures the two features of every string in A^nB^n : the number of a 's equals the number of b 's and the letters are in the correct order. So we let I be:

$$(\#_a(st) = \#_b(st)) \wedge (st \in a^*(S \cup \varepsilon) b^*).$$

Now we prove:

- I is true when $st = S$: in this case, $\#_a(st) = \#_b(st) = 0$ and st is of the correct form.
- If I is true before a rule fires, then it is true after the rule fires: to prove this, we consider the rules one at a time and show that each of them preserves I . Rule (1) adds one a and one b to st , so it does not change the difference between the number of a 's and the number of b 's. Further, it adds the a to the left of S and the b to the right of S , so if the form constraint was satisfied before applying the rule it still is afterwards. Rule (2) adds nothing so it does not change either the number of a 's or b 's or their locations.
- If I is true and st contains only terminal symbols, then $st \in A^nB^n$: in this case, st possesses the three properties required of all strings in A^nB^n : they are composed only of a 's and b 's, ($\#_a(st) = \#_b(st)$), and all a 's come before all b 's.

Next we show that every string w in A^nB^n can be generated by G : Every string in A^nB^n is of even length, so we will prove the claim only for strings of even length. The proof is by induction on $|w|$:

- Base case: if $|w| = 0$, then $w = \varepsilon$, which can be generated by applying rule (2) to S .
- Prove: if every string in A^nB^n of length k , where k is even, can be generated by G , then every string in A^nB^n of length $k + 2$ can also be generated. Notice that, for any even k , there is exactly one string in A^nB^n of length k : $a^{k/2}b^{k/2}$. There is also only one string of length $k + 2$, namely $aa^{k/2}b^{k/2}b$, that can be generated by first applying rule (1) to produce aSb , and then applying to S whatever rule sequence generated $a^{k/2}b^{k/2}$. By the induction hypothesis, such a sequence must exist. ■

Example 11.10 The Correctness of the Equal a 's and b 's Grammar

In Example 11.4 we considered the language $L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$. We built for it the grammar $G = (\{S, a, b\}, \{a, b\}, R, S)$, where:

$$R = \{ S \rightarrow aSb \quad (1)$$

$$S \rightarrow bSa \quad (2)$$

$$S \rightarrow SS \quad (3)$$

$$S \rightarrow \varepsilon \}. \quad (4)$$

This time it is perhaps less obvious that G is correct. In particular, does it generate every sequence where the number of a 's equals the number of b 's? The answer is yes, which we now prove.

To make it easy to describe this proof, we define the following function:

$$\Delta(w) = \#_a(w) - \#_b(w).$$

Note that string w is in L iff $w \in \{a, b\}^*$ and $\Delta(w) = 0$.

We begin by showing that every string w in $L(G)$ is in L : Again, let st be the working string at any point in a derivation in G . Let I be:

$$st \in \{a, b, S\}^* \wedge \Delta(st) = 0.$$

Now we prove:

- I is true when $st = S$. In this case, $\#_a(st) = \#_b(st) = 0$. So $\Delta(st) = 0$.
- If I is true before a rule fires, then it is true after the rule fires. The only symbols that can be added by any rule are a , b , and S . Rules (1) and (2) each add one a and one b to st , so neither of them changes $\Delta(st)$. Rules (3) and (4) add neither a 's nor b 's to the working string, so $\Delta(st)$ does not change.
- If I is true and st contains only terminal symbols, then $st \in L$. In this case, st possesses the two properties required of all strings in L : They are composed only of a 's and b 's and $\Delta(st) = 0$.

It is perhaps less obviously true that G generates every string in L . Can we be sure that there are no permutations that it misses? Yes, we can. We next show that every string w in L can be generated by G . Every string in L is of even length, so we will prove the claim only for strings of even length. The proof is by induction on $|w|$.

- Base case: if $|w| = 0$, $w = \varepsilon$, which can be generated by applying rule (4) to S .
- Prove that if every string in L of length $\leq k$, where k is even, can be generated by G , then every string w in L of length $k + 2$ can also be generated. Since w has length $k + 2$, it can be rewritten as one of the following: axb , bxa , axa , or bxb , for some $x \in \{a, b\}^*$. $|x| = k$. We consider two cases:
 - $w = axb$ or bxa . If $w \in L$, then $\Delta(w) = 0$ and so $\Delta(x)$ must also be 0. $|x| = k$. So, by the induction hypothesis, G generates x . Thus G can also generate w : it first applies either rule (1) (if $w = axb$) or rule (2) (if $w = bxa$). It then applies to S whatever rule sequence generated x . By the induction hypothesis, such a sequence must exist.
 - $w = axa$, or bxb . We consider the former case. The argument is parallel for the latter. Note that any string in L , of either of these forms, must have length at least 4. We will show that $w = vy$, where both v and y are in L , $2 \leq |v| \leq k$, and $2 \leq |y| \leq k$. If that is so, then G can generate w by first applying rule (3) to produce SS , and then generating v from the first S and y from the second S . By the induction hypothesis, it must be possible for it to do that since both v and y have length $\leq k$.

To find v and y , we can imagine building w (which we've rewritten as axa) up by concatenating one character at a time on the right. After adding only one character, we have just a . $\Delta(a) = 1$. Since $w \in L$, $\Delta(w) = 0$. So $\Delta(ax) = -1$ (since it is missing the final a of w). The value of Δ changes by exactly 1 each time a symbol is added to a string. Since Δ is positive when only a single character has been added and becomes negative by the time the string ax has been built, it must at some point before then have been 0. Let v be the shortest nonempty prefix of w to have a value of 0 for Δ . Since v is nonempty and only even length strings can have Δ equal to 0, $2 \leq |v|$. Since Δ became 0 sometime before w became ax , v must be at least two characters shorter than w (it must be missing at least the last character of x plus the final a), so $|v| \leq k$. Since $\Delta(v) = 0$, $v \in L$. Since $w = vy$, we know bounds on the length of y : $2 \leq |y| \leq k$. Since $\Delta(w) = 0$ and $\Delta(v) = 0$, $\Delta(y)$ must also be 0 and so $y \in L$. ■

11.6 Derivations and Parse Trees

Context-free grammars do more than just describe the set of strings in a language. They provide a way of assigning an internal structure to the strings that they derive. This structure is important because it, in turn, provides the starting point for assigning meanings to the strings that the grammar can produce.

The grammatical structure of a string is captured by a *parse tree*, which records which rules were applied to which nonterminals during the string's derivation. In Chapter 15, we will explore the design of programs, called *parsers*, that, given a grammar G and a string w , decide whether $w \in L(G)$ and, if it is, create a parse tree that captures the process by which G could have derived w .

A parse tree, derived by a grammar $G = (V, \Sigma, R, S)$, is a rooted, ordered tree in which:

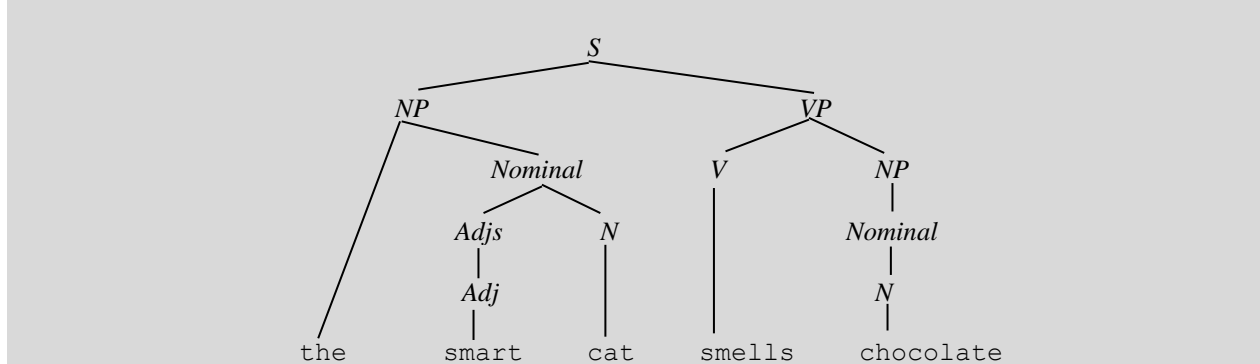
- Every leaf node is labeled with an element of $\Sigma \cup \{\varepsilon\}$,
- The root node is labeled S ,

- Every other node is labeled with some element of $V - \Sigma$, and
- If m is a nonleaf node labeled X and the children of m are labeled x_1, x_2, \dots, x_n , then R contains the rule $X \rightarrow x_1, x_2, \dots, x_n$.

Define the **branching factor** of a grammar G to be length (the number of symbols) of the longest right-hand side of any rule in G . Then the branching factor of any parse tree generated by G is less than or equal to the branching factor of G .

Example 11.11 The Parse Tree of a Simple English Sentence

Consider again the fragment of an English grammar that we wrote in Example 11.6. That grammar can be used to produce the following parse tree for the sentence the smart cat smells chocolate:



Notice that, in Example 11.11, the constituents (the subtrees) correspond to objects (like some particular cat) that have meaning in the world that is being described. It is clear from the tree that this sentence is not about cat smells or smart cat smells.

Because parse trees matter, it makes sense, given a grammar G , to distinguish between:

- G 's **weak generative capacity**, defined to be the set of strings, $L(G)$, that G generates, and
- G 's **strong generative capacity**, defined to be the set of parse trees that G generates.

When we design grammars it will be important that we consider both their weak and their strong generative capacities.

In our last example, the process of deriving the sentence the smart cat smells chocolate began with:

$$S \Rightarrow NP VP \Rightarrow \dots$$

Looking at the parse tree, it isn't possible to tell which of the following happened next:

$$\begin{aligned}
 S &\Rightarrow NP VP \Rightarrow \text{The Nominal VP} \Rightarrow \\
 S &\Rightarrow NP VP \Rightarrow NP V NP \Rightarrow
 \end{aligned}$$

Parse trees are useful precisely because they capture the important structural facts about a derivation but throw away the details of the order in which the nonterminals were expanded.

While it's true that the order in which nonterminals are expanded has no bearing on the structure that we wish to assign to a string, order will become important when we attempt to define algorithms that work with context-free grammars. For example, in Chapter 15 we will consider various parsing algorithms for context-free languages. Given an input string w , such algorithms must work systematically through the space of possible derivations in search of one that could have generated w . To make it easier to describe such algorithms, we will define two useful families of derivations:

- A **left-most derivation** is one in which, at each step, the leftmost nonterminal in the working string is chosen for expansion.
- A **right-most derivation** is one in which, at each step, the rightmost nonterminal in the working string is chosen for expansion.

Returning to the `smart cat` example above:

- A left-most derivation is:

$S \Rightarrow NP VP \Rightarrow \text{the } Nominal VP \Rightarrow \text{the } Adjs N VP \Rightarrow \text{the } Adj N VP \Rightarrow \text{the smart } N VP \Rightarrow$
 $\text{the smart cat } VP \Rightarrow \text{the smart cat } V NP \Rightarrow \text{the smart cat smells } NP \Rightarrow$
 $\text{the smart cat smells } Nominal \Rightarrow \text{the smart cat smells } N \Rightarrow$
 $\text{the smart cat smells chocolate}$

- A right-most derivation is:

$S \Rightarrow NP VP \Rightarrow NP V NP \Rightarrow NP V Nominal \Rightarrow NP V N \Rightarrow NP V \text{ chocolate} \Rightarrow$
 $NP \text{ smells chocolate} \Rightarrow \text{the } Nominal \text{ smells chocolate} \Rightarrow$
 $\text{the } Adjs N \text{ smells chocolate} \Rightarrow \text{the } Adj s \text{ cat smells chocolate} \Rightarrow$
 $\text{the } Adj \text{ cat smells chocolate} \Rightarrow$
 $\text{the smart cat smells chocolate}$

11.7 Ambiguity

Sometimes a grammar may produce more than one parse tree for some (or all) of the strings it generates. When this happens we say that the grammar is ambiguous. More precisely, a grammar G is **ambiguous** iff there is at least one string in $L(G)$ for which G produces more than one parse tree.

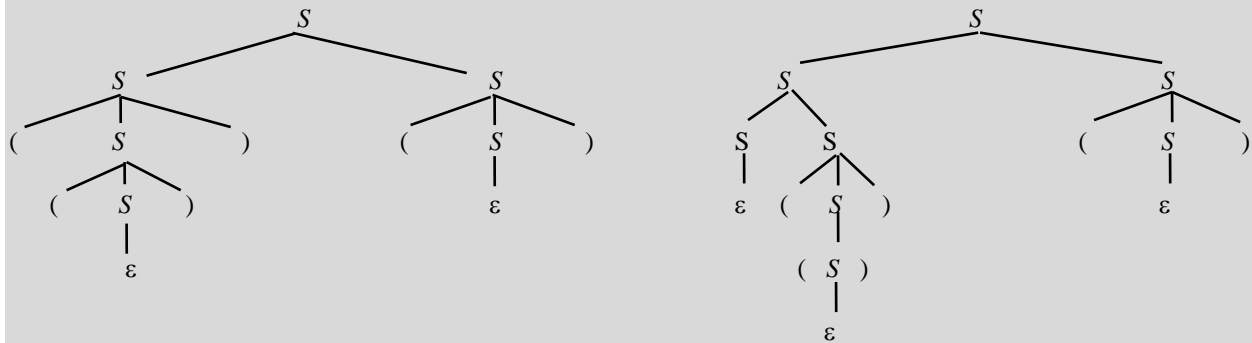
It is easy to write ambiguous grammars if we are not careful. In fact, we already have.

Example 11.12 The Balanced Parentheses Grammar is Ambiguous

Recall the language $Bal = \{w \in \{\}, \{\}^* : \text{the parentheses are balanced}\}$, for which we wrote the grammar $G = (\{S, \}, \{\}, \{\}, \{R, S\})$, where:

$$R = \{ S \rightarrow (S) \\ S \rightarrow SS \\ S \rightarrow \varepsilon \}.$$

G can produce both of the following parse trees for the string $((()())$:



In fact, G can produce an infinite number of parse trees for the string $((()())$.

A grammar G is unambiguous iff, for all strings w , at every point in a leftmost or rightmost derivation of w , only one rule in G can be applied. The grammar that we just presented in Example 11.12 clearly fails to meet this requirement. For example, here are two leftmost derivations of the string $()()$:

- $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()()$.
- $S \Rightarrow SS \Rightarrow SSS \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()()$.

11.7.2 Why Is Ambiguity a Problem?

Why are we suddenly concerned with ambiguity? Regular grammars can also be ambiguous. And regular expressions can often derive a single string in several distinct ways.

Example 11.13 Regular Expressions Grammars Can Be Ambiguous

Let $L = \{w \in \{a, b\}^* : w \text{ contains at least one } a\}$. L is regular. It can be defined with both a regular expression and a regular grammar. We show two ways in which the string aaa can be generated from the regular expression we have written and two ways in which it can be generated by the regular grammar:

Regular Expression

$(a \cup b)^*a(a \cup b)^*$.

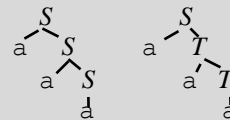
choose a from $(a \cup b)$, then
 choose a from $(a \cup b)$, then
 choose a , then
 choose ε from $(a \cup b)^*$.

or

choose ε from $(a \cup b)^*$, then
 choose a , then
 choose a from $(a \cup b)$, then
 choose a from $(a \cup b)$.

Regular Grammar

- $S \rightarrow a$
- $S \rightarrow bS$
- $S \rightarrow aS$
- $S \rightarrow aT$
- $T \rightarrow a$
- $T \rightarrow b$
- $T \rightarrow aT$
- $T \rightarrow bT$



We had no reason to be concerned with ambiguity when we were discussing regular languages because, for most applications of them, we don't care about assigning internal structure to strings. With context-free languages, we usually do care about internal structure because, given a string w , we want to assign meaning to w . We almost always want to assign a unique such meaning. It is generally difficult, if not impossible, to assign a unique meaning without a unique parse tree. So an ambiguous grammar, which fails to produce a unique parse tree, is a problem, as we'll see in our next example.

Example 11.14 An Ambiguous Expression Grammar

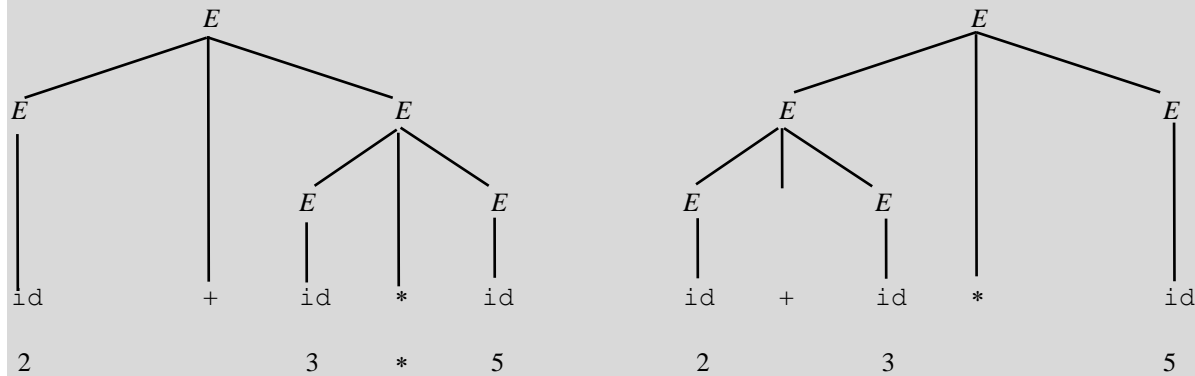
Consider E_{opr} , which we'll define to be the language of simple arithmetic expressions of the kind that could be part of anything from a small calculator to programming language. We can define E_{opr} with the following context-free grammar $G = (\{E, \text{id}, +, *, (,)\}, \{\text{id}, +, *, (,)\}, R, E)$, where:

$$R = \{ E \rightarrow E + E \\ E \rightarrow E * E \\ E \rightarrow (E) \\ E \rightarrow \text{id} \}.$$

So that we can focus on the issues we care about, we've used the terminal symbol id as a shorthand for any of the numbers or variables that can actually occur as the operands in the expressions that G generates. Most compilers and

interpreters for expression languages handle the parsing of individual operands in a first pass, called lexical analysis, which can be done with an FSM. We'll return to this topic in Chapter 15.

Consider the string $2 + 3 * 5$, which we will write as $\text{id} + \text{id} * \text{id}$. Using G , we can get two parses for this string:



Should an evaluation of this expression return 17 or 25? (See Example 11.19 for a different expression grammar that fixes this problem.)

Natural languages, like English and Chinese, are not explicitly designed. So it isn't possible to go in and remove ambiguity from them. See Example 11.22 and § 752.

Designers of practical languages must be careful that they create languages for which they can write unambiguous grammars.

11.7.3 Inherent Ambiguity

In many cases, when confronted with an ambiguous grammar G , it is possible to construct a new grammar G' that generates $L(G)$ and that has less (or no) ambiguity. Unfortunately, it is not always possible to do this. There exist context-free languages for which no unambiguous grammar exists. We call such languages *inherently ambiguous*.

Example 11.15 An Inherently Ambiguous Language

Let $L = \{a^i b^j c^k : i, j, k \geq 0, i = j \text{ or } j = k\}$. An alternative way to describe it is $\{a^n b^n c^m : n, m \geq 0\} \cup \{a^n b^m c^m : n, m \geq 0\}$. Every string in L has either (or both) the same number of a's and b's or the same number of b's and c's. L is inherently ambiguous. One grammar that describes it is $G = (\{S, S_1, S_2, A, B, a, b, c\}, \{a, b, c\}, R, S)$, where:

$$\begin{aligned}
 R = \{ & S \rightarrow S_1 \mid S_2 \\
 & S_1 \rightarrow S_1 c \mid A \quad /* \text{Generate all strings in } \{a^n b^n c^m : n, m \geq 0\}. \\
 & A \rightarrow aAb \mid \varepsilon \\
 & S_2 \rightarrow aS_2 B \quad /* \text{Generate all strings in } \{a^n b^m c^m : n, m \geq 0\}. \\
 & B \rightarrow bBc \mid \varepsilon \}.
 \end{aligned}$$

Now consider the strings in $A^n B^n C^n = \{a^n b^n c^n : n \geq 0\}$. They have two distinct derivations, one through S_1 and the other through S_2 . It is possible to prove that L is inherently ambiguous: given any grammar G that generates L there is at least one string with two derivations in G .

Example 11.16 Another Inherently Ambiguous Language

Let $L = \{a^i b^j a^k b^l : i, j, k, l \geq 0, i = k \text{ or } j = l\}$. L is also inherently ambiguous.

Unfortunately, there are no clean fixes for the ambiguity problem for context-free languages. In Section 22.5 we'll see that both of the following problems are undecidable:

- Given a context-free grammar G , is G ambiguous?
- Given a context-free language L , is L inherently ambiguous?

11.7.4 Techniques for Reducing Ambiguity ✦

Despite the negative theoretical results that we have just mentioned, it is usually very important, when we are designing practical languages and their grammars, that we come up with a language that is not inherently ambiguous and a grammar for it that is unambiguous. Although there exists no general purpose algorithm to test for ambiguity in a grammar or to remove it when it is found (since removal is not always possible), there do exist heuristics that we can use to find some of the more common sources of ambiguity and remove them. We'll consider here three grammar structures that often lead to ambiguity:

- ϵ rules like $S \rightarrow \epsilon$.
- rules like $S \rightarrow SS$ or $E \rightarrow E + E$. In other words recursive rules whose right-hand sides are symmetric and contain at least two copies of the nonterminal on the left-hand side.
- rule sets that lead to ambiguous attachment of optional postfixes.

Eliminating ϵ -Rules

In Example 11.12, we showed a grammar for the balanced parentheses language. That grammar is highly ambiguous. Its major problem is that it is possible to apply the rule $S \rightarrow SS$ arbitrarily often, generating unnecessary instances of S , which can then be wiped out without a trace using the rule $S \rightarrow \epsilon$. If we could eliminate the rule $S \rightarrow \epsilon$, we could eliminate that source of ambiguity. We'll call any rule whose right-hand side is ϵ an **ϵ -rule**.

We'd like to define an algorithm that could remove ϵ -rules from a grammar G without changing the language that G generates. Clearly if $\epsilon \in L(G)$, that won't be possible. Only an ϵ -rule can generate ϵ . However, it is possible to define an algorithm that eliminates ϵ -rules from G and leaves $L(G)$ unchanged except that, if $\epsilon \in L(G)$, it will be absent from the language generated by the new grammar. We will show such an algorithm. Then we'll show a simple way to add ϵ back in, when necessary, without adding back the kind of ϵ -rules that cause ambiguity.

Let $G = (V, \Sigma, R, S)$ be any context-free grammar. The following algorithm constructs a new grammar G' such that $L(G') = L(G) - \{\epsilon\}$ and G' contains no ϵ -rules:

removeEps(G : CFG) =

1. Let $G' = G$.
2. Find the set N of nullable variables in G' . A variable X is **nullable** iff either:
 - (1) there is a rule $X \rightarrow \epsilon$, or
 - (2) there is a rule $X \rightarrow PQR\dots$ such that P, Q, R, \dots are all nullable.
 So compute N as follows:
 - 2.1. Set N to the set of variables that satisfy (1).
 - 2.2. Until an entire pass is made without adding anything to N do:

Evaluate all other variables with respect to (2). If any variable satisfies (2) and is not in N , insert it.
3. Define a rule to be **modifiable** iff it is of the form $P \rightarrow \alpha Q \beta$ for some Q in N and any α, β in V^* . Since Q is nullable, it could be wiped out by the application of ϵ -rules. But those rules are about to be deleted. So one possibility should be that Q just doesn't get generated in the first place. To make that happen requires adding new rules. So, repeat until G' contains no modifiable rules that haven't been processed:
 - 3.1. Given the rule $P \rightarrow \alpha Q \beta$, where $Q \in N$, add the rule $P \rightarrow \alpha \beta$ if it is not already present and if $\alpha \beta \neq \epsilon$ and if $P \neq \alpha \beta$. This last check prevents adding the useless rule $P \rightarrow P$, which would otherwise be generated if the original grammar contained, for example, the rule $P \rightarrow PQ$ and Q were nullable.
4. Delete from G' all rules of the form $X \rightarrow \epsilon$.
5. Return G' .

If *removeEps* halts, $L(G') = L(G) - \{\varepsilon\}$ and G' contains no ε -rules. And *removeEps* must halt. Since step 2 must add a nonterminal to N at each pass and it cannot add any symbol more than once, it must halt within $|V - \Sigma|$ passes. Step 3 may have to be done once for every rule in G and once for every new rule that it adds. But note that, whenever it adds a new rule, that rule has a shorter right-hand side than the rule from which it came. So the number of new rules that can be generated by some original rule in G is finite. So step 3 can execute only a finite number of times.

Example 11.17 Eliminating ε -Rules

Let $G = (\{S, T, A, B, C, a, b, c\}, \{a, b, c\}, R, S)$, where:

$$R = \{ S \rightarrow aTa \\ T \rightarrow ABC \\ A \rightarrow aA \mid C \\ B \rightarrow Bb \mid C \\ C \rightarrow c \mid \varepsilon \}.$$

On input G , *removeEps* behaves as follows: Step 2 finds the set N of nullable variables by initially setting N to $\{C\}$. On its first pass through step 2.2 it adds A and B to N . On the next pass, it adds T (since now A , B , and C are all in N). On the next pass, no new elements are found, so step 2 halts with $N = \{C, A, B, T\}$. Step 3 adds the following new rules to G' :

$$\begin{array}{ll} S \rightarrow aa & /* \text{ Since } T \text{ is nullable.} \\ T \rightarrow BC & /* \text{ Since } A \text{ is nullable.} \\ T \rightarrow AC & /* \text{ Since } B \text{ is nullable.} \\ T \rightarrow AB & /* \text{ Since } C \text{ is nullable.} \\ T \rightarrow C & /* \text{ From } T \rightarrow BC, \text{ since } B \text{ is nullable. Or from } T \rightarrow AC. \\ T \rightarrow B & /* \text{ From } T \rightarrow BC, \text{ since } C \text{ is nullable. Or from } T \rightarrow AB. \\ T \rightarrow A & /* \text{ From } T \rightarrow AC, \text{ since } C \text{ is nullable. Or from } T \rightarrow AB. \\ A \rightarrow a & /* \text{ Since } A \text{ is nullable.} \\ B \rightarrow b & /* \text{ Since } B \text{ is nullable.} \end{array}$$

Finally, step 4 deletes the rule $C \rightarrow \varepsilon$.

Sometimes $L(G)$ contains ε and it is important to retain it. To handle this case, we present the following algorithm, which constructs a new grammar G'' , such that $L(G'') = L(G)$. If $L(G)$ contains ε , then G'' will contain a single ε -rule that can be thought of as being “quarantined”. Its sole job is to generate the string ε . It can have no interaction with the other rules of the grammar.

atmostoneEps(G : CFG) =

1. $G'' = \text{removeEps}(G)$.
2. If S_G is nullable then: /* This means that $\varepsilon \in L(G)$.
 - 2.1. Create in G'' a new start symbol S^* .
 - 2.2. Add to $R_{G''}$ the two rules: $S^* \rightarrow \varepsilon$ and $S^* \rightarrow S_G$.
3. Return G'' .

Example 11.18 Eliminating ε -Rules from the Balanced Prens Grammar

We again consider $\text{Bal} = \{w \in \{\}, \{\}^* : \text{the parentheses are balanced}\}$ and the grammar $G = (\{S, \}, \{\}, \{\}, \{\}, R, S)$, where:

$$R = \{ S \rightarrow (S) \quad (1) \\ S \rightarrow SS \quad (2) \\ S \rightarrow \varepsilon \quad (3) \}.$$

We would like to eliminate the ambiguity in G . Since $\varepsilon \in L(G)$, we call $atmostoneEps(G)$, which begins by applying $removeEps$ to G :

- In step 2, $N = \{S\}$.
- In step 3, rule (1) causes us to add the rule $S \rightarrow ()$. Rule (2) causes us to consider adding the rule $S \rightarrow S$, but we omit adding rules whose right-hand sides and left-hand sides are the same.
- In step 4, we delete the rule $S \rightarrow \varepsilon$.

So $removeEps(G)$ returns the grammar $G' = (\{S, (), \{, \}, \{, \}, R, S)$, where $R =$

$$\{ S \rightarrow (S) \\ S \rightarrow () \\ S \rightarrow SS \}.$$

In its step 2, $atmostoneEps$ creates the new start symbol S^* . In step 3, it adds the two rules $S^* \rightarrow \varepsilon$, $S^* \rightarrow S$. So $atmostoneEps$ returns the grammar $G'' = (\{S^*, S, (), \{, \}, \{, \}, R, S^*)$, where:

$$R = \{ S^* \rightarrow \varepsilon \\ S^* \rightarrow S \\ S \rightarrow (S) \\ S \rightarrow () \\ S \rightarrow SS \}.$$

The string $()()()$ has only one parse in G'' .

Eliminating Symmetric Recursive Rules

The new grammar that we just built for Bal is better than our original one. But it is still ambiguous. The string $()()()$ has two parses, shown in Figure 11.1. The problem now is the rule $S \rightarrow SS$, which must be applied $n-1$ times to generate a sequence of n balanced parentheses substrings. But, at each time after the first, there is a choice of which existing S to split.

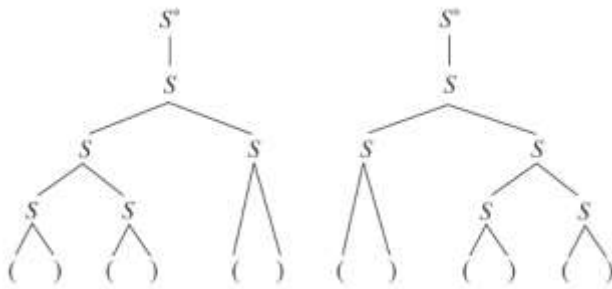


Figure 11.1 Two parse trees for the string $()()()$

The solution to this problem is to rewrite the grammar so that there is no longer a choice. We replace the rule $S \rightarrow SS$ with one of the following rules:

$$S \rightarrow SS_1 \quad /* \text{ force branching to the left.} \\ S \rightarrow S_1S \quad /* \text{ force branching to the right.}$$

Then we add the rule $S \rightarrow S_1$ and replace the rules $S \rightarrow (S)$ and $S \rightarrow ()$ with the rules $S_1 \rightarrow (S)$ and $S_1 \rightarrow ()$. What we have done is to change the grammar so that branching can occur only in one direction. Every S that is generated can branch, but no S_1 can. When all the branching has happened, S rewrites to S_1 and the rest of the derivation can occur.

So one unambiguous grammar for Bal is $G = (\{S, \}, \{ \}, \{ \}, \{ \}, R, S)$, where:

$$\begin{aligned}
 R = \{ & S^* \rightarrow \varepsilon & (1) \\
 & S^* \rightarrow S & (2) \\
 & S \rightarrow SS_1 & (3) & /* Force branching to the left. \\
 & S \rightarrow S_1 & (4) \\
 & S_1 \rightarrow (S) & (5) \\
 & S_1 \rightarrow () \}. & (6)
 \end{aligned}$$

The technique that we just used for Bal is useful in any situation in which ambiguity arises from a recursive rule whose right-hand side contains two or more copies of the left-hand side. An important application of this idea is to expression languages, like the language of arithmetic expressions that we introduced in Example 11.14.

Example 11.19 An Unambiguous Expression Grammar

Consider again the language E_{expr} , which we defined with the following context-free grammar $G = (\{E, \text{id}, +, *, (, \}), \{\text{id}, +, *, (, \}), R, E)$, where:

$$\begin{aligned}
 R = \{ & E \rightarrow E + E \\
 & E \rightarrow E * E \\
 & E \rightarrow (E) \\
 & E \rightarrow \text{id} \}.
 \end{aligned}$$

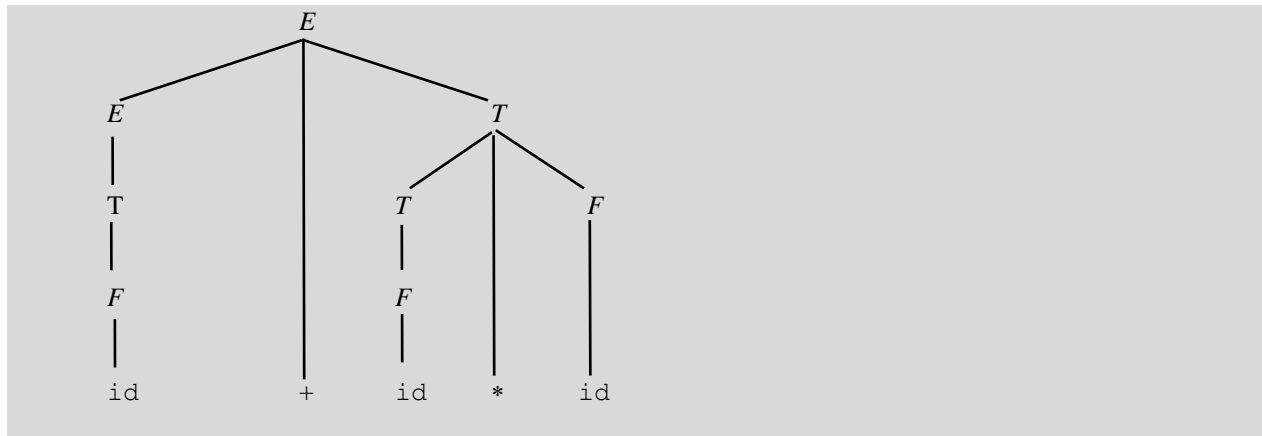
G is ambiguous in two ways:

1. It fails to specify associativity. So, for example, there are two parses for the string $\text{id} + \text{id} + \text{id}$, corresponding to the bracketings $(\text{id} + \text{id}) + \text{id}$ and $\text{id} + (\text{id} + \text{id})$.
2. It fails to define a precedence hierarchy for the operators $+$ and $*$. So, for example, there are two parses for the string $\text{id} + \text{id} * \text{id}$, corresponding to the bracketings $(\text{id} + \text{id}) * \text{id}$ and $\text{id} + (\text{id} * \text{id})$.

The first of these problems is analogous to the one we just solved for Bal. We could apply that solution here, but then we'd still have the second problem. We can solve both of them with the following grammar $G' = (\{E, T, F, \text{id}, +, *, (, \}), \{\text{id}, +, *, (, \}), R, E)$, where:

$$\begin{aligned}
 R = \{ & E \rightarrow E + T \\
 & E \rightarrow T \\
 & T \rightarrow T * F \\
 & T \rightarrow F \\
 & F \rightarrow (E) \\
 & F \rightarrow \text{id} \}.
 \end{aligned}$$

Just as we did for Bal, we have forced branching to go in a single direction (to the left) when identical operators are involved. And, by adding the levels T (for term) and F (for factor) we have defined a precedence hierarchy: times has higher precedence than plus does. Using G' , there is now a single parse for the string $\text{id} + \text{id} * \text{id}$:



Ambiguous Attachment

The third source of ambiguity that we will consider arises when constructs with optional fragments are nested. The problem in such cases is then, “Given an instance of the optional fragment, at what level of the parse tree should it be attached?”

Probably the most often described instance of this kind of ambiguity is known as the *dangling else problem*. Suppose that we define a programming language with an `if` statement that can have either of the following forms:

```

<stmt> ::= if <cond> then <stmt>
<stmt> ::= if <cond> then <stmt> else <stmt>

```

In other words, the `else` clause is optional. Then the following statement, with just a single `else` clause, has two parses:

```

if cond1 then if cond2 then st1 else st2

```

In the first parse, the single `else` clause goes with the first `if`. (So it attaches high in the parse tree.) In the second parse, the single `else` clause goes with the second `if`. (In this case, it attaches lower in the parse tree.)

Example 11.20 The Dangling Else Problem in Java

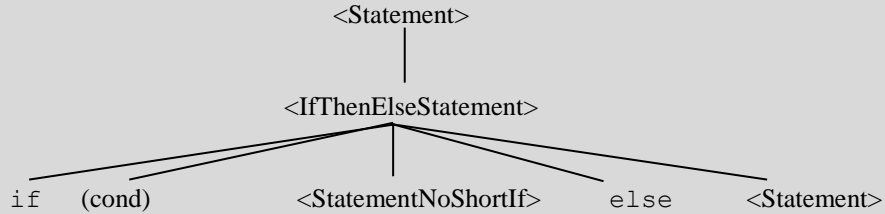
Most programming languages that have the dangling else problem (including C, C++, and Java) specify that each `else` goes with the innermost `if` to which it can be attached. The Java grammar forces this to happen by changing the rules to something like these (presented here in a simplified form that omits many of the statement types that are allowed):

```

<Statement> ::= <IfThenStatement> | <IfThenElseStatement> | <IfThenElseStatementNoShortIf>
<StatementNoShortIf> ::= <block> | <IfThenElseStatementNoShortIf> | ...
<IfThenStatement> ::= if ( <Expression> ) <Statement>
<IfThenElseStatement> ::= if ( <Expression> ) <StatementNoShortIf> else <Statement>
<IfThenElseStatementNoShortIf> ::= if ( <Expression> ) <StatementNoShortIf> else <StatementNoShortIf>

```

In this grammar, there is a special class of statements called `<StatementNoShortIf>`. These are statements that are guaranteed not to end with a short (i.e., `else-less if` statement). The grammar uses this class to guarantee that, if a top-level `if` statement has an `else` clause, then any embedded `if` must also have one. To see how this works, consider the following parse tree:



The top-level `if` statement claims the `else` clause for itself by guaranteeing that there will not be an embedded `if` that is missing an `else`. If there were, then that embedded `if` would grab the one `else` clause there is.

For a discussion of another way in which programming languages can solve this problem, see [C 668](#).

Attachment ambiguity is also a problem for parsers for natural languages such as English, as we'll see in Example 11.22

Proving that a Grammar is Unambiguous

While it is undecidable, *in general*, whether a grammar is ambiguous or unambiguous, it may be possible to prove that a *particular* grammar is either ambiguous or unambiguous. A grammar G can be shown to be ambiguous by exhibiting a single string for which G produces two parse trees. To see how it might be possible to prove that G is unambiguous, recall that G is unambiguous iff every string derivable in G has a single leftmost derivation. So, if we can show that, during any leftmost derivation of any string $w \in L(G)$, exactly one rule can be applied, then G is unambiguous.

Example 11.21 The Final Balanced Prens Grammar is Unambiguous

We return to the final grammar G that we produced for Bal. $G = (\{S, S_1, ()\}, \{(), \{(), R, S\})$, where:

- $$\begin{aligned}
 R = \{ & S^* \rightarrow \varepsilon & (1) \\
 & S^* \rightarrow S & (2) \\
 & S \rightarrow SS_1 & (3) \\
 & S \rightarrow S_1 & (4) \\
 & S_1 \rightarrow (S) & (5) \\
 & S_1 \rightarrow () \} & (6)
 \end{aligned}$$

We prove that G is unambiguous. Given the leftmost derivation of any string w in $L(G)$, there is, at each step of the derivation, a unique symbol, which we'll call X , that is the leftmost nonterminal in the working string. Whatever X is, it must be expanded by the next rule application, so the only rules that may be applied next are those with X on the left-hand side. There are three nonterminals in G . We show, for each of them, that the rules that expand them never compete in the leftmost derivation of a particular string w . We do the two easy cases first:

- S^* : the only place that S^* may occur in a derivation is at the beginning. If $w = \varepsilon$, then rule (1) is the only one that can be applied. If $w \neq \varepsilon$, then rule (2) is the only one that can be applied.
- S_1 : if the next two characters to be derived are $()$, S_1 must expand by rule 6. Otherwise, it must expand by rule 5.

In order to discuss S , we first define, for any matched set of parentheses m , the siblings of m to be the smallest set that includes any matched set p adjacent to m and all of p 's siblings. So, for example, consider the string:

$$\begin{array}{cccc}
 (& (&) &) & (&) \\
 \underline{1} & \underline{2} & \underline{3} & \underline{4} & & \\
 & & & & \underline{5} &
 \end{array}$$

The set $()$ labeled 1 has a single sibling, 2. The set $((()))$ labeled 5 has two siblings, 3 and 4. Now we can consider S . We observe that:

- S must generate a string in Bal and so it must generate a matched set, possibly with siblings.
- So the first terminal character in any string that S generates is $($. Call the string that starts with that $($ and ends with the $)$ that matches it, s .
- The only thing that S_1 can generate is a single matched set of parentheses that has no siblings.
- Let n be the number of siblings of s . In order to generate those siblings, S must expand by rule (3) exactly n times (producing n copies of S_1) before it expands by rule (4) to produce a single S_1 , which will produce s . So, at every step in a derivation, let p be the number of occurrences of S_1 to the right of S . If $p < n$, S must expand by rule 3. If $p = n$, S must expand by rule 4.

Going Too Far

We must be careful, in getting rid of ambiguity, that we don't do so at the expense of being able to generate the parse trees that we want. In both the arithmetic expression example and the dangling else case, we were willing to force one interpretation. Sometimes, however, that is not an acceptable solution.

Example 11.22 Throwing Away The Parses That We Want

Let's return to the small English grammar that we showed in Example 11.6. That grammar is ambiguous. It has an ambiguous attachment problem, similar to the dangling else problem. Consider the following two sentences:

- Chris likes the girl with a cat.
- Chris shot the bear with a rifle.

Each of these sentences has two parse trees because, in each case, the prepositional phrase *with a N*, can be attached either to the immediately preceding *NP* (the girl or the bear) or to the *VP*. The correct interpretation for the first sentence is that there is a girl with a cat and Chris likes her. In other words, the prepositional phrase attaches to the *NP*. Almost certainly, the correct interpretation for the second sentence is that there is a bear (with no rifle) and Chris used a rifle to shoot it. In other words, the prepositional phrase attaches to the *VP*. See § 752 for additional discussion of this example.

For now, the key point is that we could solve the ambiguity problem by eliminating one of the choices for *PP* attachment. But then, for one of our two sentences, we'd get a parse tree that corresponds to nonsense. In other words, we might still have a grammar with the required weak generative capacity. But we would no longer have one with the required strong generative capacity. The solution to this problem is to add some additional mechanism to the context-free framework. That mechanism must be able to choose the parse that corresponds to the most likely meaning.

English parsers must have ways to handle various kinds of attachment ambiguities, including those caused by prepositional phrases and relative clauses. § 752.

11.8 Normal Forms ★

So far, we've imposed no restrictions on the form of the right-hand side of our grammar rules, although we have seen that some kinds of rules, like those whose right-hand side is ϵ , can make grammars harder to use. In this section, we consider what happens if we carry the idea of getting rid of ϵ -productions a few steps farther.

Normal forms for queries and data can simplify database processing. § 690. Normal forms for logical formulas can simplify automated reasoning in artificial intelligence systems, § 760, and in program verification systems. § 679.

Let C be any set of data objects. For example, C might be the set of context-free grammars. Or it could be the set of syntactically valid logical expressions or a set of database queries. We'll say that a set F is a **normal form** for C iff it possesses the following two properties:

- For every element c of C , except possibly a finite set of special cases, there exists some element f of F such that f is equivalent to c with respect to some set of tasks.
- F is simpler than the original form in which the elements of C are written. By “simpler” we mean that at least some tasks are easier to perform on elements of F than they would be on elements of C .

We define normal forms in order to make other tasks easier. For example, it might be easier to build a parser if we could make some assumptions about the form of the grammar rules that the parser will use. Recall that, in Section 5.8, we introduced the notion of a canonical form for a set of objects. A normal form is a weaker notion, since it does not require that there be a unique representation for each object in C , nor does it require that “equivalent” objects map to the same representation. So it is sometimes possible to define useful normal forms when no useful canonical form exists. We’ll now do that for context-free grammars.

11.8.1 Normal Forms for Grammars

We’ll define the following two useful normal forms for context-free grammars:

- **Chomsky Normal Form:** in a Chomsky normal form grammar $G = (V, \Sigma, R, S)$, all rules have one of the following two forms:
 - $X \rightarrow a$, where $a \in \Sigma$, or
 - $X \rightarrow BC$, where B and C are elements of $V - \Sigma$.

Every parse tree that is generated by a grammar in Chomsky normal form has a branching factor of exactly 2, except at the branches that lead to the terminal nodes, where the branching factor is 1. This property makes Chomsky normal form grammars useful in several ways, including:

- Parsers can exploit efficient data structures for storing and manipulating binary trees.
- Every derivation of a string w contains $|w|-1$ applications of some rule of the form $X \rightarrow BC$, and $|w|$ applications of some rule of the form $X \rightarrow a$. So it is straightforward to define a decision procedure to determine whether w can be generated by a Chomsky normal form grammar G .

In addition, because the form of all the rules is so restricted, it is easier than it would otherwise be to define other algorithms that manipulate grammars.

- **Greibach Normal Form:** in a Greibach normal form grammar $G = (V, \Sigma, R, S)$, all rules have the following form:
 - $X \rightarrow a\beta$, where $a \in \Sigma$ and $\beta \in (V - \Sigma)^*$.

In every derivation that is produced by a grammar in Greibach normal form, precisely one terminal is generated for each rule application. This property is useful in several ways, including:

- Every derivation of a string w contains $|w|$ rule applications. So again it is straightforward to define a decision procedure to determine whether w can be generated by a Greibach normal form grammar G .
- As we’ll see in Theorem 14.2, Greibach normal form grammars can easily be converted to pushdown automata with no ϵ -transitions. This is useful because such PDAs are guaranteed to halt.

Theorem 11.1 Chomsky Normal Form

Theorem: Given a context-free grammar G , there exists an equivalent Chomsky normal form grammar G_C such that $L(G_C) = L(G) - \{\epsilon\}$.

Proof: The proof is by construction, using the algorithm *converttoChomsky* presented below. ■

Theorem 11.2 Greibach Normal Form

Theorem: Given a context-free grammar G , there exists an equivalent Greibach normal form grammar G_G such that $L(G_G) = L(G) - \{\epsilon\}$.

Proof: The proof is also by construction. We present it in § 630. ■

11.8.2 Converting to a Normal Form

Normal forms are useful if there exists a procedure for converting an arbitrary object into a corresponding object that meets the requirements of the normal form. Algorithms to convert grammars into normal forms generally begin with a grammar G and then operate in a series of steps as follows:

1. Apply some transformation to G to get rid of undesirable property 1. Show that the language generated by G is unchanged.
2. Apply another transformation to G to get rid of undesirable property 2. Show that the language generated by G is unchanged *and* that undesirable property 1 has not been reintroduced.
3. Continue until the grammar is in the desired form.

Because it is possible for one transformation to undo the work of an earlier one, the order in which the transformation steps are performed is often critical to the correctness of the transformation algorithm.

One transformation that we will exploit in converting grammars both to Chomsky normal form and to Greibach normal form is based on the following observation. Consider a grammar that contains the three rules:

$$\begin{aligned} X &\rightarrow aYc \\ Y &\rightarrow b \\ Y &\rightarrow ZZ \end{aligned}$$

We can construct an equivalent grammar by replacing the X rule with the rules:

$$\begin{aligned} X &\rightarrow abc \\ X &\rightarrow aZZc \end{aligned}$$

Instead of letting X generate an instance of Y , X immediately generates whatever Y could have generated. The following theorem generalizes this claim.

Theorem 11.3 Rule Substitution

Theorem: Let $G = (V, \Sigma, R, S)$ be a context-free grammar that contains a rule r of the form $X \rightarrow \alpha Y \beta$, where α and β are elements of V^* and $Y \in (V - \Sigma)$. Let $Y \rightarrow \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_n$ be all of G 's rules whose left-hand side is Y . And let G' be the result of removing from R the rule r and replacing it by the rules $X \rightarrow \alpha \gamma_1 \beta, X \rightarrow \alpha \gamma_2 \beta, \dots, X \rightarrow \alpha \gamma_n \beta$. Then $L(G') = L(G)$.

Proof: We first show that every string in $L(G)$ is also in $L(G')$: suppose that w is in $L(G)$. If G can derive w without using rule r , then G' can do so in exactly the same way. If G can derive w using rule r , then one of its derivations has the following form, for some value of k between 1 and n :

$$S \Rightarrow \dots \Rightarrow \delta X \phi \Rightarrow \delta \alpha Y \beta \phi \Rightarrow \delta \alpha \gamma_k \beta \phi \Rightarrow \dots \Rightarrow w.$$

Then G' can derive w with the derivation:

$$S \Rightarrow \dots \Rightarrow \delta X \phi \Rightarrow \delta \alpha \gamma_k \beta \phi \Rightarrow \dots \Rightarrow w.$$

Next we show that only strings in $L(G)$ can be in $L(G')$. This must be so because the action of every new rule $X \rightarrow \alpha\gamma_k\beta$ could have been performed in G by applying the rule $X \rightarrow \alpha Y \beta$ and then the rule $Y \rightarrow \gamma_k$. ■

11.8.3 Converting to Chomsky Normal Form

There exists a straightforward four-step algorithm that converts a grammar $G = (V, \Sigma, R, S)$ into a new grammar G_C such that G_C is in Chomsky normal form and $L(G_C) = L(G) - \{\epsilon\}$. Define:

converttoChomsky(G : CFG) =

1. Let G_C be the result of removing from G all ϵ -rules, using the algorithm *removeEps*, defined in Section 11.7.3.
2. Let G_C be the result of removing from G_C all unit productions (rules of the form $A \rightarrow B$), using the algorithm *removeUnits* defined below. It is important that *removeUnits* run after *removeEps* since *removeEps* may introduce unit productions. Once this step has been completed, all rules whose right-hand sides have length 1 are in Chomsky normal form (i.e., they are composed of a single terminal symbol).
3. Let G_C be the result of removing from G_C all rules whose right-hand sides have length greater than 1 and include a terminal (e.g., $A \rightarrow aB$ or $A \rightarrow BaC$). This step is simple and can be performed by the algorithm *removeMixed* given below. Once this step has been completed, all rules whose right-hand sides have length 1 or 2 are in Chomsky normal form.
4. Let G_C be the result of removing from G_C all rules whose right-hand sides have length greater than 2 (e.g., $A \rightarrow BCDE$). This step too is simple. It can be performed by the algorithm *removeLong* given below.
5. Return G_C .

A **unit production** is a rule whose right-hand side consists of a single nonterminal symbol. The job of *removeUnits* is to remove all unit productions and to replace them by a set of other rules that accomplish the job previously done by the unit productions. So, for example, suppose that we start with a grammar G that contains the following rules:

$$\begin{aligned} S &\rightarrow XY \\ X &\rightarrow A \\ A &\rightarrow B \mid a \\ B &\rightarrow b \end{aligned}$$

Once we get rid of unit productions, it will no longer be possible for X to become A (and then B) and thus to go on to generate a or b . So X will need the ability to go directly to a and b , without any intermediate steps. We can define *removeUnits* as follows:

removeUnits(G : CFG) =

1. Let $G' = G$.
2. Until no unit productions remain in G' do:
 - 2.1. Choose some unit production $X \rightarrow Y$.
 - 2.2. Remove it from G' .
 - 2.3. Consider only rules that still remain in G' . For every rule $Y \rightarrow \beta$, where $\beta \in V^*$, do:
Add to G' the rule $X \rightarrow \beta$ unless that is a rule that has already been removed once.
3. Return G' .

Notice that we have not bothered to check to make sure that we don't insert a rule that is already present. Since R , the set of rules, is a set, inserting an element that is already in the set has no effect.

At each step of its operation, *removeUnits* is performing the kind of rule substitution described in Theorem 11.3. (It happens that both α and β are empty.) So that theorem tells us that, at each step, the language generated by G' is unchanged from the previous step. If *removeUnits* halts, it is clear that all unit productions have been removed. It is less obvious that *removeUnits* can be guaranteed to halt. At each step, one unit production is removed, but several new rules may be added, including new unit productions. To see that *removeUnit* must halt, we observe that there is

a bound $= |V - \Sigma|^2$ on the number of unit productions that can be formed from a fixed set $V - \Sigma$ of nonterminals. At each step, *removeUnits* removes one element from that set and that element can never be reinserted. So *removeUnits* must halt in at most $|V - \Sigma|^2$ steps.

Example 11.23 Removing Unit Productions

Let $G = (V, \Sigma, R, S)$, where:

$$R = \{ S \rightarrow XY \\ X \rightarrow A \\ A \rightarrow B \mid a \\ B \rightarrow b \\ Y \rightarrow T \\ T \rightarrow Y \mid c \}.$$

The order in which *removeUnits* chooses unit productions to remove doesn't matter. We'll consider one order it could choose:

Remove $X \rightarrow A$. Since $A \rightarrow B \mid a$, add $X \rightarrow B \mid a$.

Remove $X \rightarrow B$. Add $X \rightarrow b$.

Remove $Y \rightarrow T$. Add $Y \rightarrow Y \mid c$. Notice that we've added $Y \rightarrow Y$, which is useless, but it will be removed later.

Remove $Y \rightarrow Y$. Consider adding $Y \rightarrow T$, but don't since it has previously been removed.

Remove $A \rightarrow B$. Add $A \rightarrow b$.

Remove $T \rightarrow Y$. Add $T \rightarrow c$, but with no effect since it was already present.

At this point, the rules of G are:

$$S \rightarrow XY \\ A \rightarrow a \mid b \\ B \rightarrow b \\ T \rightarrow c \\ X \rightarrow a \mid b \\ Y \rightarrow c$$

No unit productions remain, so *removeUnits* halts.

We must now define the two straightforward algorithms that are required by steps 3 and 4 of the conversion algorithm that we sketched above. We begin by defining:

removeMixed(G : CFG) =

1. Let $G' = G$.
2. Create a new nonterminal T_a for each terminal a in Σ .
3. Modify each rule in G' whose right-hand side has length greater than 1 and that contains a terminal symbol by substituting T_a for each occurrence of the terminal a .
4. Add to G' , for each T_a , the rule $T_a \rightarrow a$.
5. Return G' .

Example 11.24 Removing Mixed Productions

The result of applying *removeMixed* to the grammar:

$$\begin{aligned}
A &\rightarrow a \\
A &\rightarrow aB \\
A &\rightarrow BaC \\
A &\rightarrow BbC
\end{aligned}$$

is the grammar:

$$\begin{aligned}
A &\rightarrow a \\
A &\rightarrow T_a B \\
A &\rightarrow BT_a C \\
A &\rightarrow BT_b C \\
T_a &\rightarrow a \\
T_b &\rightarrow b
\end{aligned}$$

Finally we define *removeLong*. The idea for *removeLong* is simple. If there is a rule with n symbols on its right-hand side, replace it with a set of rules. The first rule generates the first symbol followed by a new symbol that will correspond to “the rest”. The next rule rewrites that symbol as the second of the original symbols, followed by yet another new one, again corresponding to “the rest”, and so forth, until there are only two symbols left to generate. So we define:

removeLong(G : CFG) =

1. Let $G' = G$.
2. For each G' rule r^k of the form $A \rightarrow N_1 N_2 N_3 N_4 \dots N_n$, $n > 2$, create new nonterminals $M^k_2, M^k_3, \dots, M^k_{n-1}$.
3. In G' , replace r^k with the rule $A \rightarrow N_1 M^k_2$.
4. To G' , add the rules $M^k_2 \rightarrow N_2 M^k_3, M^k_3 \rightarrow N_3 M^k_4, \dots, M^k_{n-1} \rightarrow N_{n-1} N_n$.
5. Return G' .

When we illustrate this algorithm, we typically omit the superscripts on the M 's, and, instead, guarantee that we use distinct nonterminals by using distinct subscripts.

Example 11.25 Removing Rules with Long Right-Hand Sides

The result of applying *removeLong* to the single rule grammar:

$$A \rightarrow BCDEF$$

is the grammar with rules:

$$\begin{aligned}
A &\rightarrow BM_2 \\
M_2 &\rightarrow CM_3 \\
M_3 &\rightarrow DM_4 \\
M_4 &\rightarrow EF
\end{aligned}$$

We can now illustrate the four steps of *converttoChomsky*.

Example 11.26 Converting a Grammar to Chomsky Normal Form

Let $G = (\{S, A, B, C, a, c\}, \{A, B, C\}, R, S)$, where:

$$\begin{aligned}
R = \{ &S \rightarrow aACa \\
&A \rightarrow B \mid a \\
&B \rightarrow C \mid c \\
&C \rightarrow cC \mid \varepsilon \}.
\end{aligned}$$

We convert G to Chomsky normal form. Step 1 applies *removeEps* to eliminate ε -productions. We compute N , the set of nullable variables. Initially $N = \{C\}$. Because of the rule $B \rightarrow C$, we add B . Then, because of the rule $A \rightarrow B$, we add A . So $N = \{A, B, C\}$. Since both A and C are nullable, we derive three new rules from the first original rule, giving us:

$$S \rightarrow aACa \mid aAa \mid aCa \mid aa$$

We add $A \rightarrow \varepsilon$ and $B \rightarrow \varepsilon$, but both of them will disappear at the end of this step. We also add $C \rightarrow c$. So *removeEps* returns the rule set:

$$\begin{aligned} S &\rightarrow aACa \mid aAa \mid aCa \mid aa \\ A &\rightarrow B \mid a \\ B &\rightarrow C \mid c \\ C &\rightarrow cC \mid c \end{aligned}$$

Next we apply *removeUnits*:

Remove $A \rightarrow B$. Add $A \rightarrow C \mid c$.
 Remove $B \rightarrow C$. Add $B \rightarrow cC$ (and $B \rightarrow c$, but it was already there).
 Remove $A \rightarrow C$. Add $A \rightarrow cC$ (and $A \rightarrow c$, but it was already there).

So *removeUnits* returns the rule set:

$$\begin{aligned} S &\rightarrow aACa \mid aAa \mid aCa \mid aa \\ A &\rightarrow a \mid c \mid cC \\ B &\rightarrow c \mid cC \\ C &\rightarrow cC \mid c \end{aligned}$$

Next we apply *removeMixed*, which returns the rule set:

$$\begin{aligned} S &\rightarrow T_aACT_a \mid T_aAT_a \mid T_aCT_a \mid T_aT_a \\ A &\rightarrow a \mid c \mid T_cC \\ B &\rightarrow c \mid T_cC \\ C &\rightarrow T_cC \mid c \\ T_a &\rightarrow a \\ T_c &\rightarrow c \end{aligned}$$

Finally, we apply *removeLong*, which returns the rule set:

$$\begin{array}{llll} S \rightarrow T_aS_1 & S \rightarrow T_aS_3 & S \rightarrow T_aS_4 & S \rightarrow T_aT_a \\ S_1 \rightarrow AS_2 & S_3 \rightarrow AT_a & S_4 \rightarrow CT_a & \\ S_2 \rightarrow CT_a & & & \\ \hline A &\rightarrow a \mid c \mid T_cC \\ B &\rightarrow c \mid T_cC \\ C &\rightarrow T_cC \mid c \\ T_a &\rightarrow a \\ T_c &\rightarrow c \end{array}$$

From Example 11.26 we see that the Chomsky normal form version of a grammar may be longer than the original grammar was. How much longer? And how much time may be required to execute the conversion algorithm? We

can answer both of these questions by answering them for each of the steps that the conversion algorithm executes. Let n be the length of an original grammar G . Then we have:

1. Use *removeEps* to remove ϵ -rules: suppose that G contains a rule of the form $X \rightarrow A_1 A_2 A_3 \dots A_k$. If all of the variables A_1 through A_k are nullable, this single rule will be rewritten as $2^k - 1$ rules (since each of the k nonterminals can either be present or not, except that they cannot all be absent). Since k can grow as n , we have that the length of the grammar that *removeEps* produces (and thus the amount of time that *removeEps* requires) is $\mathcal{O}(2^n)$. In this worst case, the conversion algorithm becomes impractical for all but toy grammars. We can prevent this worst case from occurring though. Suppose that all right-hand sides can be guaranteed to be short. For example, suppose they all have length at most 2. Then no rule will be rewritten as more than 3 rules. We can make this guarantee if we modify *converttoChomsky* slightly. We will run *removeLong* as step 1 rather than as step 4. Note that none of the other steps can create a rule whose right-hand side is longer than the right-hand side of some rule that already exists. So it is not necessary to rerun *removeLong* later. With this change, *removeEps* runs in linear time.
2. Use *removeUnits* to remove unit productions: we've already shown that this step must halt in at most $|V - \Sigma|^2$ steps. Each of those steps takes constant time and may create one new rule. So the length of the grammar that *removeUnits* produces, as well as the time required for it to run, is $\mathcal{O}(n^2)$.
3. Use *removeMixed* to remove rules with right-hand sides of length greater than 1 and that contain a terminal symbol: this step runs in linear time and constructs a grammar whose size is $\mathcal{O}(n)$.
4. Use *removeLong* to remove rules with long right-hand sides: this step runs in linear time and constructs a grammar whose size is $\mathcal{O}(n)$.

So, if we change *converttoChomsky* so that it does step 4 first, its time complexity is $\mathcal{O}(n^2)$ and the size of the grammar that it produces is also $\mathcal{O}(n^2)$.

11.8.4 The Price of Normal Forms

While normal forms are useful for many things, as we will see over the next few chapters, it is important to keep in mind that they exact a price and it's one that we may or may not be willing to pay, depending on the application. If G is an arbitrary context-free grammar and G' is an equivalent grammar in Chomsky (or Greibach) normal form, then G and G' generate the same set of strings, but only in rare cases (for example if G happened already to be in normal form) do they assign to those strings the same parse trees. Thus, while converting a grammar to a normal form has no effect on its weak generative capacity, it may have a significant effect on its strong generative capacity.

11.9 Island Grammars

Suppose that we want to parse strings that possess one or more of the following properties:

- Some (perhaps many) of them are ill-formed. In other words, while there may be a grammar that describes what strings are "supposed to look like", there is no guarantee that the actual strings we'll see conform to those rules. Consider, for example, any grammar you can imagine for English. Now imagine picking up the phone and hearing something like, "Um, I uh need a copy of uh my bill for er Ap, no May, I think, or June, maybe all of them uh, I guess that would work." Or consider a grammar for HTML. It will require that tags be properly nested. But strings like `<i>bold italic</i>` show up not infrequently in HTML documents. Most browsers will do the right thing with them, so they never get debugged.
- We simply don't know enough about them to build an exact model, although we do know something about some patterns that we think the strings will contain.
- They may contain substrings in more than one language. For example, bi(multi)lingual people often mix their speech. We even give names to some of the resulting hybrids: Spanglish, Japlish, Hinglish, etc. Or consider a typical Web page. It may contain fragments of HTML, Java script, or other languages, interleaved with each other. Even when parsing strings that are all in the same "language", dialectical issues may arise. For example, in response to the question, "Are you going to fix dinner tonight?" an American speaker of English might say, "I

could,” while a British speaker of English might say, “I could do.” Similarly, in analyzing legacy software, there are countless dialects of languages like Fortran and Cobol.

- They may contain some substrings we care about, interleaved with other substrings we don’t care about and don’t want to waste time parsing. For example, when parsing an XML document to determine its top level structure, we may have no interest in the text or even in many of the tags.

Island grammars can play a useful role in reverse engineering software systems € 688.

In all of these cases, the role of any grammar we might build is different than the role a grammar plays, say, in a compiler. In the latter case, the grammar is prescriptive. A compiler can simply reject inputs that do not conform to the grammar it is given. Contrast that with a tool whose job is to analyze legacy software or handle customer phone calls. Such a tool must do the best it can with the input that it sees. When building tools of that sort, it may make sense to exploit what is called an island grammar. An *island grammar* is a grammar that has two parts:

- A set of detailed rules that describe the fragments that we care about. We’ll call these fragments *islands*.
- A set of flexible rules that can match everything else. We’ll call everything else the *water*.

A very simple form of island grammar is a regular expression that just describes the patterns that we seek. A regular expression matcher ignores those parts of the input string that do not match the patterns. But suppose that the patterns we are looking for cannot be described with regular expressions. For example, they may require balanced parentheses. Or suppose that we want to assign structure to the islands. In that case, we need something more powerful than a regular expression (or a regular grammar). One way to view a context-free island grammar is that it is a hybrid between a context-free grammar and a set of regular expressions.

To see how island grammars work, consider the problem of examining legacy software to determine patterns of static subroutine invocation. To solve this problem, we could use the following island grammar, which is a simplification and modification of one presented in [Moonen 2001]:

- | | | |
|-----|---|--------------|
| [1] | $\langle \text{input} \rangle \rightarrow \langle \text{chunk} \rangle^*$ | |
| [2] | $\langle \text{chunk} \rangle \rightarrow \text{CALL } \langle \text{id} \rangle (\langle \text{expr} \rangle)$ | {cons(CALL)} |
| [3] | $\langle \text{chunk} \rangle \rightarrow \text{CALL ERROR } (\langle \text{expr} \rangle)$ | {reject} |
| [4] | $\langle \text{chunk} \rangle \rightarrow \langle \text{water} \rangle$ | |
| [5] | $\langle \text{water} \rangle \rightarrow \Sigma^*$ | {avoid} |

- Rule 1 says that a complete input file is a set of chunks. The next three rules describe three kinds of chunks:
 - Rule 2 describes the chunks we are trying to find. Assume that another set of rules (such as the ones we considered in Example 11.19) defines the valid syntax for expressions. Those rules may exploit the full power of a context-free grammar, for example to guarantee that parenthesized expressions are properly nested. Then rule 2 will find well-formed function calls. The action associated with it, {cons(CALL)}, tells the parser what kind of node to build whenever this rule is used.
 - Rule 3 describes chunks that, although they could be formed by rule 2, are structures that we know we are not interested in. In this case, there is a special kind of error call that we want to ignore. The action {reject} says that whenever this rule matches, its result should be ignored.
 - Rule 4 describes water, i.e., the chunks that correspond to the parts of the program that aren’t CALL statements.
- Rule 5 is used to generate the water. But notice that it has the {avoid} action associated with it. That means that it will not be used to match any text that can be matched by some other, non-avoiding rule.

Island grammars can be exploited by appropriately crafted parsers. But we should note here, to avoid confusion, that there is also a somewhat different notion, called *island parsing*, in which the goal is to use a standard grammar to produce a complete parse given an input string. But, while conventional parsers read and analyze their inputs left-to-right, an island parser first scans its input looking for one or more regions where it seems likely that a correct parse tree can be built. Then it grows the parse tree outward from those “islands” of (relative) certainty. If the input is ill-formed (as is likely to happen, for example, in the case of spoken language understanding), then the final output of

the parser will be a sequence of islands, rather than a complete parse. So island grammars and island parsing are both techniques for coping with ill-formed and unpredictable inputs. Island grammars approach the task by specifying, at grammar-writing time, which parts of the input should be analyzed and which should be ignored. Island parsers, in this other sense, approach the task by using a full grammar and deciding, at parse time, which input fragments appear to be parsable and which don't.

11.10 Stochastic Context-Free Grammars ✪

Recall that, at the end of our discussion of finite state machines in Chapter 5, we introduced the idea of a stochastic FSM: an NDFSM whose transitions have been augmented with probabilities that describe some phenomenon that we want to model. We can apply that same idea to context-free grammars: We can add probabilities to grammar rules and so create a *stochastic context-free grammar* (also called a *probabilistic context-free grammar*) that generates strings whose distribution matches some naturally occurring distribution with which we are concerned.

A stochastic context-free grammar can be used to generate random English text that may seem real enough to fool some people 🖨️.

A stochastic context-free grammar G is a quintuple (V, Σ, R, S, D) , where:

- V is the rule alphabet, which contains nonterminals (symbols that are used in the grammar but that do not appear in strings in the language) and terminals,
- Σ (the set of terminals) is a subset of V ,
- R (the set of rules) is a finite subset of $(V - \Sigma) \times V^*$,
- S (the start symbol) can be any element of $V - \Sigma$,
- D is a function from R to $[0 - 1]$. So D assigns a probability to each rule in R . D must satisfy the requirement that, for every nonterminal symbol X , the sum of the probabilities associated with all rules whose left-hand side is X must be 1.

Example 11.27 A Simple Stochastic Grammar

Recall $\text{PalEven} = \{ww^R : w \in \{a, b\}^*\}$, the language of even-length palindromes of a's and b's. Suppose that we want to describe the specific case in which a's occur three times as often as b's do. Then we might write the grammar $G = (\{S, a, b\}, \{a, b\}, R, S, D)$, where R and D are defined as follows:

$$\begin{array}{ll} S \rightarrow aSa & [.72] \\ S \rightarrow bSb & [.24] \\ S \rightarrow \varepsilon & [.04] \end{array}$$

Given a grammar G and a string s , the probability of a particular parse tree t is the product of the probabilities associated with the rules that were used to generate it. In other words, if we let C be the collection (in which duplicates count) of rules that were used to generate t and we let $\text{Pr}(r)$ be the probability associated with rule r , then:

$$\text{Pr}(t) = \prod_{r \in C} \text{Pr}(r).$$

Stochastic context-free grammars play an important role in natural language processing. © 754.

Stochastic grammars can be used to answer two important kinds of questions:

- In an error-free environment, we know that we need to analyze a particular string s . So we want to solve the following problem: Given s , find the most likely parse tree for it.
- In a noisy environment, we may not be sure exactly what string we need to analyze. For example, suppose that it is possible that there have been spelling errors, so the true string is similar but not identical to the one we have observed. Or suppose that there may have been transmission errors. Or suppose that we have transcribed a spoken

string and it is possible that we didn't hear it correctly. In all of these cases we want to solve the following problem: Given a set of possible true strings X and an observed string o , find the particular string s (and possibly also the most likely parse for it) that is most likely to have been the one that was actually generated. Note that the probability of generating any particular string w is the sum of the probabilities of generating each possible parse tree for w . In other words, if T is the set of possible parse trees for w , then the total probability of generating w is:

$$\Pr(w) = \sum_{t \in T} \Pr(t).$$

Then the sentence s that is most likely to have been generated, given the observation o , is the one with the highest conditional probability given o . Recall that argmax of w returns the value of the argument w that maximizes the value of the function it is given. So the highest probability sentence s is:

$$\begin{aligned} s &= \operatorname{argmax}_{w \in X} \Pr(w | o) \\ &= \operatorname{argmax}_{w \in X} \frac{\Pr(o | w) \Pr(w)}{\Pr(o)}. \end{aligned}$$

Stochastic context-free grammars can be used model the three-dimensional structure of RNA. © 737.

In Chapter 15, we will discuss techniques for parsing context-free languages that are defined by standard (i.e., without probabilistic information) context-free grammars. Those techniques can be extended to create techniques for parsing using stochastic grammars. So they can be used to answer both of the questions that we just presented.

11.11 Exercises

- 1) Let $\Sigma = \{a, b\}$. For the languages that are defined by each of the following grammars, do each of the following:
 - (i) List five strings that are in L .
 - (ii) List five strings that are not in L (or as many as there are, whichever is greater).
 - (iii) Describe L concisely. You can use regular expressions, expressions using variables (e.g., $a^n b^n$), or set theoretic expressions (e.g., $\{x: \dots\}$).
 - (iv) Indicate whether or not L is regular. Prove your answer.
 - a) $S \rightarrow aS \mid Sb \mid \varepsilon$
 - b) $S \rightarrow aSa \mid bSb \mid a \mid b$
 - c) $S \rightarrow aS \mid bS \mid \varepsilon$
 - d) $S \rightarrow aS \mid aSbS \mid \varepsilon$
- 2) Let G be the grammar of Example 11.12. Show a third parse tree that G can produce for the string $((()())$.
- 3) Consider the following grammar G :

$$S \rightarrow 0S1 \mid SS \mid 10$$

Show a parse tree produced by G for each of the following strings:

- a) 010110.
- b) 00101101.

- 4) Consider the following context free grammar G :

$$\begin{aligned} S &\rightarrow aSa \\ S &\rightarrow T \\ S &\rightarrow \varepsilon \\ T &\rightarrow bT \\ T &\rightarrow cT \\ T &\rightarrow \varepsilon \end{aligned}$$

One of these rules is redundant and could be removed without altering $L(G)$. Which one?

- 5) Using the simple English grammar that we showed in Example 11.6, show two parse trees for each of the following sentences. In each case, indicate which parse tree almost certainly corresponds to the intended meaning of the sentence:
- The bear shot Fluffy with the rifle.
 - Fluffy likes the girl with the chocolate.
- 6) Show a context-free grammar for each of the following languages L :
- BalDelim = $\{w : w \text{ is a string of delimiters: } (,), [,], \{, \}, \text{ that are properly balanced}\}$.
 - $\{a^i b^j : 2i = 3j + 1\}$.
 - $\{a^i b^j : 2i \neq 3j + 1\}$.
 - $\{w \in \{a, b\}^* : \#_a(w) = 2 \#_b(w)\}$.
 - $L = \{w \in \{a, b\}^* : w = w^R\}$.
 - $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } (i \neq j \text{ or } j \neq k)\}$.
 - $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } (k \leq i \text{ or } k \leq j)\}$.
 - $\{w \in \{a, b\}^* : \text{every prefix of } w \text{ has at least as many } a\text{'s as } b\text{'s}\}$.
 - $\{a^m b^n : m \geq n, m-n \text{ is even}\}$.
 - $\{a^m b^n c^p d^q : m, n, p, q \geq 0 \text{ and } m + n = p + q\}$.
 - $\{xc^n : x \in \{a, b\}^* \text{ and } (\#_a(x) = n \text{ or } \#_b(x) = n)\}$.
 - $\{b_i \# b_{i+1}^R : b_i \text{ is the binary representation of some integer } i, i \geq 0, \text{ without leading zeros}\}$. (For example $101\#011 \in L$.)
 - $\{x^R \# y : x, y \in \{0, 1\}^* \text{ and } x \text{ is a substring of } y\}$.
- 7) Let G be the ambiguous expression grammar of Example 11.14. Show at least three different parse trees that can be generated from G for the string $id+id*id*id$.
- 8) Consider the unambiguous expression grammar G' of Example 11.19.
- Trace the derivation of the string $id+id*id*id$ in G' .
 - Add exponentiation ($**$) and unary minus ($-$) to G' , assigning the highest precedence to unary minus, followed by exponentiation, multiplication, and addition, in that order.
- 9) Let $L = \{w \in \{a, b, \cup, \varepsilon, (,), *, +\}^* : w \text{ is a syntactically legal regular expression}\}$.
- Write an unambiguous context-free grammar that generates L . Your grammar should have a structure similar to the arithmetic expression grammar G' that we presented in Example 11.19. It should create parse trees that:
 - Associate left given operators of equal precedence, and
 - Correspond to assigning the following precedence levels to the operators (from highest to lowest):
 - $*$ and $+$
 - concatenation
 - \cup
 - Show the parse tree that your grammar will produce for the string $(a \cup b) ba^*$.

- 10) Let $L = \{w \in \{A - Z, \neg, \wedge, \vee, \rightarrow, (,)\}^* : w \text{ is a syntactically legal Boolean expression}\}$.
- Write an unambiguous context-free grammar that generates L and that creates parse trees that that:
 - Associate left given operators of equal precedence, and
 - Correspond to assigning the following precedence levels to the operators (from highest to lowest): \neg , \wedge , \vee , and \rightarrow .
 - Show the parse tree that your grammar will produce for the string $\neg P \vee R \rightarrow Q \rightarrow S$.
- 11) In \mathbb{C} 704, we present a simplified grammar for URIs (Uniform Resource Identifiers), the names that we use to refer to objects on the Web.
- Using that grammar, show a parse tree for:


```
https://www.mystuff.wow/widgets/fradgit#sword
```
 - Write a regular expression that is equivalent to the grammar that we present.
- 12) Prove that each of the following grammars is correct:
- The grammar, shown in Example 11.3, for the language PalEven.
 - The grammar, shown in Example 11.1, for the language Bal.
- 13) For each of the following grammars G , show that G is ambiguous. Then find an equivalent grammar that is not ambiguous.
- $(\{S, A, B, T, a, c\}, \{a, c\}, R, S)$, where $R = \{S \rightarrow AB, S \rightarrow BA, A \rightarrow aA, A \rightarrow ac, B \rightarrow Tc, T \rightarrow aT, T \rightarrow a\}$.
 - $(\{S, a, b\}, \{a, b\}, R, S)$, where $R = \{S \rightarrow \varepsilon, S \rightarrow aSa, S \rightarrow bSb, S \rightarrow aSb, S \rightarrow bSa, S \rightarrow SS\}$.
 - $(\{S, A, B, T, a, c\}, \{a, c\}, R, S)$, where $R = \{S \rightarrow AB, A \rightarrow AA, A \rightarrow a, B \rightarrow Tc, T \rightarrow aT, T \rightarrow a\}$.
 - $(\{S, a, b\}, \{a, b\}, R, S)$, where $R = \{S \rightarrow aSb, S \rightarrow bSa, S \rightarrow SS, S \rightarrow \varepsilon\}$. (G is the grammar that we presented in Example 11.10 for the language $L = \{w \in \{a,b\}^* : \#_a(w) = \#_b(w)\}$.)
 - $(\{S, a, b\}, \{a, b\}, R, S)$, where $R = \{S \rightarrow aSb, S \rightarrow aaSb, S \rightarrow \varepsilon\}$.
- 14) Let G be any context-free grammar. Show that the number of strings that have a derivation in G of length n or less, for any $n > 0$, is finite.
- 15) Consider the fragment of a Java grammar that is presented in Example 11.20. How could it be changed to force each `else` clause to be attached to the outermost possible `if` statement?
- 16) How does the COND form in Lisp, as described in \mathbb{C} 672, avoid the dangling else problem?
- 17) Consider the grammar G' of Example 11.19.
- Convert G' to Chomsky normal form.
 - Consider the string `id*id+id`.
 - Show the parse tree that G' produces for it.
 - Show the parse tree that your Chomsky normal form grammar produces for it.
- 18) Convert each of the following grammars to Chomsky normal form:
- $$S \rightarrow a S a$$

$$S \rightarrow B$$

$$B \rightarrow b b C$$

$$B \rightarrow b b$$

$$C \rightarrow \varepsilon$$

$$C \rightarrow c C$$

- b) $S \rightarrow ABC$
 $A \rightarrow aC \mid D$
 $B \rightarrow bB \mid \varepsilon \mid A$
 $C \rightarrow Ac \mid \varepsilon \mid Cc$
 $D \rightarrow aa$
- c) $S \rightarrow aTVa$
 $T \rightarrow aTa \mid bTb \mid \varepsilon \mid V$
 $V \rightarrow cVc \mid \varepsilon$

12 Pushdown Automata

Grammars define context-free languages. We'd also like a computational formalism that is powerful enough to enable us to build an acceptor for every context-free language. In this chapter, we describe such a formalism.

12.1 Definition of a (Nondeterministic) PDA

A pushdown automaton, or PDA, is a finite state machine that has been augmented by a single stack. In a minute, we will present the formal definition of the PDA model that we will use. But, before we do that, one caveat to readers of other books is in order. There are several competing PDA definitions, from which we have chosen one to present here. All are provably equivalent, in the sense that, for all i and j , if there exists a version $_i$ PDA that accepts some language L then there also exists a version $_j$ PDA that accepts L . We'll return to this issue in Section 12.5, where we will mention a few of the other models and sketch an equivalence proof. For now, simply beware of the fact that other definitions are also in widespread use.

We will use the following definition: A *pushdown automaton* (or *PDA*) M is a sixtuple $(K, \Sigma, \Gamma, \Delta, s, A)$, where:

- K is a finite set of states,
- Σ is the input alphabet,
- Γ is the stack alphabet,
- $s \in K$ is the start state,
- $A \subseteq K$ is the set of accepting states, and
- Δ is the transition relation. It is a finite subset of

$$(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*) \times (K \times \Gamma^*)$$

state	input or ε	string of symbols to pop from top of stack
-------	------------------------	---

state	string of symbols to push on top of stack
-------	--

A *configuration* of a PDA M is an element of $K \times \Sigma^* \times \Gamma^*$. It captures the three things that can make a difference to M 's future behavior:

- its current state,
- the input that is still left to read, and
- the contents of its stack.

The *initial configuration* of a PDA M , on input w , is (s, w, ε) .

We will use the following notational convention for describing M 's stack as a string: The top of the stack is to the left of the string. So:

$$\begin{array}{|c} c \\ a \\ b \end{array} \quad \text{will be written as:} \quad cab$$

If a sequence $c_1c_2\dots c_n$ of characters is pushed onto the stack, they will be pushed rightmost first, so if the value of the stack before the push was s , the value after the push will be $c_1c_2\dots c_ns$.

Analogously to what we did for FSMs, we define the relation *yields-in-one-step*, written \vdash_M . *Yields-in-one-step* relates *configuration* $_1$ to *configuration* $_2$ iff M can move from *configuration* $_1$ to *configuration* $_2$ in one step. Let c be any element of $\Sigma \cup \{\varepsilon\}$, let γ_1, γ_2 and γ be any elements of Γ^* , and let w be any element of Σ^* . Then:

$$(q_1, cw, \gamma_1\gamma) \vdash_M (q_2, w, \gamma_2\gamma) \text{ iff } ((q_1, c, \gamma_1), (q_2, \gamma_2)) \in \Delta.$$

Note two things about what a transition $((q_1, c, \gamma_1), (q_2, \gamma_2))$ says about how M manipulates its stack:

- M may only take the transition if the string γ_1 matches the current top of the stack. If it does, and the transition is taken, then M pops γ_1 and then pushes γ_2 . M cannot “peek” at the top of its stack without popping off the values that it examines.
- If $\gamma_1 = \varepsilon$, then M must match ε against the top of the stack. But ε matches everywhere. So letting γ_1 be ε is equivalent to saying “without bothering to check the current value of the stack”. It is not equivalent to saying, “if the stack is empty.” In our definition, there is no way to say that directly, although we will see that we can create a way by letting M , before it does anything else, push a special marker onto the stack. Then, whenever that marker is on the top of the stack, the stack is otherwise empty.

The relation *yields*, written \vdash_M^* , is the reflexive, transitive closure of \vdash_M . So configuration C_1 yields configuration C_2 iff:

$$C_1 \vdash_M^* C_2.$$

A **computation** by M is a finite sequence of configurations C_0, C_1, \dots, C_n for some $n \geq 0$ such that:

- C_0 is an initial configuration,
- C_n is of the form (q, ε, γ) , for some state $q \in K$ and some string γ in Γ^* , and
- $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \dots \vdash_M C_n$.

Note that we have defined the behavior of a PDA M by a transition relation Δ , not a transition function. Thus we allow nondeterminism. If M is in some configuration (q_1, s, γ) , it is possible that:

- Δ contains exactly one transition that matches. In that case, M makes the specified move.
- Δ contains more than one transition that matches. In that case, M chooses one of them. Each choice defines one computation that M may perform.
- Δ contains no transition that matches. In that case, the computation that led to that configuration halts.

Let C be a computation of M on input $w \in \Sigma^*$. Then we will say that:

- C is an **accepting computation** iff $C = (s, w, \varepsilon) \vdash_M^* (q, \varepsilon, \varepsilon)$, for some $q \in A$. Note the strength of this requirement: A computation accepts only if it runs out of input when it is in an accepting state *and* the stack is empty.
- C is a **rejecting computation** iff $C = (s, w, \varepsilon) \vdash_M^* (q, w', \alpha)$, where C is not an accepting computation and where M has no moves that it can make from (q, w', α) . A computation can reject only if the criteria for accepting have not been met, *and* there are no further moves (including following ε -transitions) that can be taken.

Let w be a string that is an element of Σ^* . Then we will say that:

- M **accepts** w iff *at least one* of its computations accepts.
- M **rejects** w iff all of its computations reject.

The **language accepted by M** , denoted $L(M)$, is the set of all strings accepted by M . Note that it is possible that, on input w , M neither accepts nor rejects.

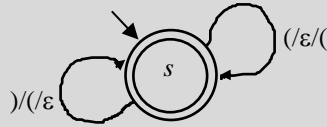
In all the examples that follow, we will draw a transition $((q_1, c, \gamma_1), (q_2, \gamma_2))$ as an arc from q_1 to q_2 , labeled $c/\gamma_1/\gamma_2$. So such a transition should be read to say, “If c matches the input and γ_1 matches the top of the stack, the transition from q_1 to q_2 can be taken, in which case c should be removed from the input, γ_1 should be popped from the stack, and γ_2 should be pushed onto it.” If $c = \varepsilon$, then the transition can be taken without consuming any input. If $\gamma_1 = \varepsilon$, the

transition can be taken without checking the stack or popping anything. If $\gamma_2 = \varepsilon$, nothing is pushed onto the stack when the transition is taken. As we did with FSMs, we will use a double circle to indicate accepting states.

Even very simple PDAs may be able to accept languages that cannot be accepted by any FSM. The power of such machines comes from the ability of the stack to count.

Example 12.1 The Balanced Parentheses Language

Consider again $\text{Bal} = \{w \in \{\}, \{\}^* : \text{the parentheses are balanced}\}$. The following one-state PDA M accepts Bal . M uses its stack to count the number of left parentheses that have not yet been matched. We show M graphically and then as a sextuple:



$M = (K, \Sigma, \Gamma, \Delta, s, A)$, where:

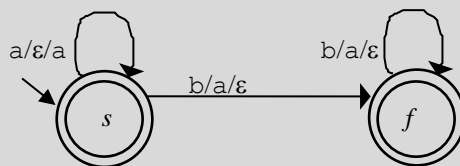
- $K = \{s\}$, (the states)
- $\Sigma = \{(\,)\}$, (the input alphabet)
- $\Gamma = \{(\}$, (the stack alphabet)
- $A = \{s\}$, and (the accepting state)
- $\Delta = \{((s, (\, \varepsilon), (s, (\, \varepsilon)), ((s,), (\, \varepsilon), (s, \varepsilon))\}$.

If M sees a (, it pushes it onto the stack (regardless of what was already there). If it sees a) and there is a (that can be popped off the stack, M does so. If it sees a) and there is no (to pop, M halts without accepting. If, after consuming its entire input string, M 's stack is empty, M accepts. If the stack is not empty, M rejects.

PDAs, like FSMs, can use their states to remember facts about the structure of the string that has been read so far. We see this in the next example.

Example 12.2 $A^n B^n$

Consider again $A^n B^n = \{a^n b^n : n \geq 0\}$. The following PDA M accepts $A^n B^n$. M uses its states to guarantee that it only accepts strings that belong to $a^* b^*$. It uses its stack to count a 's so that it can compare them to the b 's. We show M graphically:



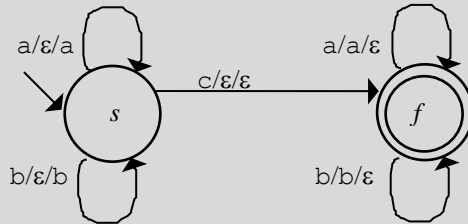
Writing it out, we have $M = (K, \Sigma, \Gamma, \Delta, s, A)$, where:

- $K = \{s, f\}$, (the states)
- $\Sigma = \{a, b\}$, (the input alphabet)
- $\Gamma = \{a\}$, (the stack alphabet)
- $A = \{s, f\}$, and (the accepting states)
- $\Delta = \{((s, a, \varepsilon), (s, a)), ((s, b, a), (f, \varepsilon)), ((f, b, a), (f, \varepsilon))\}$.

Remember that M only accepts if, when it has consumed its entire input string, it is in an accepting state *and* its stack is empty. So, for example, M will reject aaa , even though it will be in state s , an accepting state, when it runs out of input. The stack at that point will contain aaa .

Example 12.3 WcW^R

Let $WcW^R = \{wcw^R : w \in \{a, b\}^*\}$. The following PDA M accepts WcW^R :

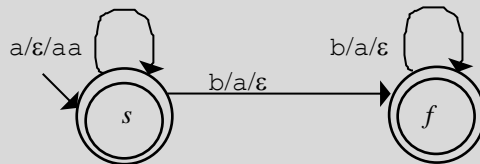


M moves from state s , in which it is recording w , to state f , in which it is checking for w^R , when it sees the character c . Since every string in WcW^R must contain the middle c , state s is not an accepting state.

The definition that we have chosen to use for a PDA is flexible; it allows several symbols to be pushed or popped from the stack in one move. This will turn out to be particularly useful when we attempt to build PDAs that correspond to practical grammars that contain rules like $T \rightarrow T * F$ (the multiplication rule that was part of the arithmetic expression grammar that we defined in Example 11.19). But we illustrate the use of this flexibility here on a simple case.

Example 12.4 A^nB^{2n}

Let $A^nB^{2n} = \{a^n b^{2n} : n \geq 0\}$. The following PDA M accepts A^nB^{2n} by pushing two a 's onto the stack for every a in the input string. Then each b pops a single a :



12.2 Deterministic and Nondeterministic PDAs

The definition of a PDA that we have presented allows nondeterminism. It sometimes makes sense, however, to restrict our attention to deterministic PDAs. In this section we will define what we mean by a deterministic PDA. We also show some examples of the power of nondeterminism in PDAs. Unfortunately, in contrast to the situation with FSMs, and as we will prove in Theorem 13.13, there exist nondeterministic PDAs for which no equivalent deterministic PDA exists.

12.2.1 Definition of a Deterministic PDA

Define a PDA M to be *deterministic* iff there exists no configuration of M in which M has a choice of what to do next. For this to be true, two conditions must hold:

1. Δ_M contains no pairs of transitions that compete with each other.
2. If q is an accepting state of M , then there is no transition $((q, \epsilon, \epsilon), (p, a))$ for any p or a . In other words, M is never forced to choose between accepting and continuing. Any transitions out of an accepting state must either consume input (since, if there is remaining input, M does not have the option of accepting) or pop something from the stack (since, if the stack is not empty, M does not have the option of accepting).

So far, all of the PDAs that we have built have been deterministic. So each machine followed only a single computational path.

12.2.2 Exploiting Nondeterminism

But a PDA may be designed to have multiple competing moves from a single configuration. As with FSMs, the easiest way to envision the operation of a nondeterministic PDA M is as a tree, as shown in Figure 12.1. Each node in the tree corresponds to a configuration of M and each path from the root to a leaf node corresponds to one computation that M might perform.

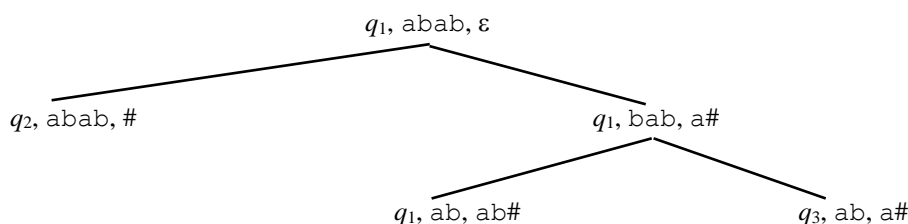
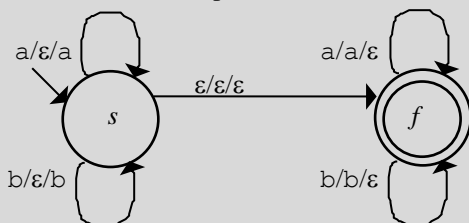


Figure 12.1 Viewing nondeterminism as search through a space of computation paths

Notice that the state, the stack, and the remaining input can be different along different paths. As a result, it will not be possible to simulate all paths in parallel, the way we did for NDFSMs.

Example 12.5 Even Length Palindromes

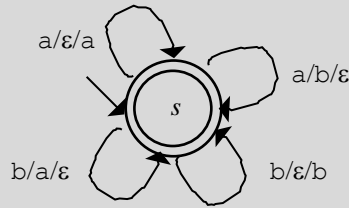
Consider again $\text{PalEven} = \{ww^R : w \in \{a,b\}^*\}$, the language of even-length palindromes of a's and b's. The following nondeterministic PDA M accepts PalEven :



M is nondeterministic because it cannot know when it has reached the middle of its input. Before each character is read, it has two choices: It can guess that it has not yet gotten to the middle. In that case, it stays in state s , where it pushes each symbol it reads. Or it can guess that it has reached the middle. In that case, it takes the ϵ -transition to state f , where it pops one symbol for each symbol that it reads.

Example 12.6 Equal Numbers of a's and b's

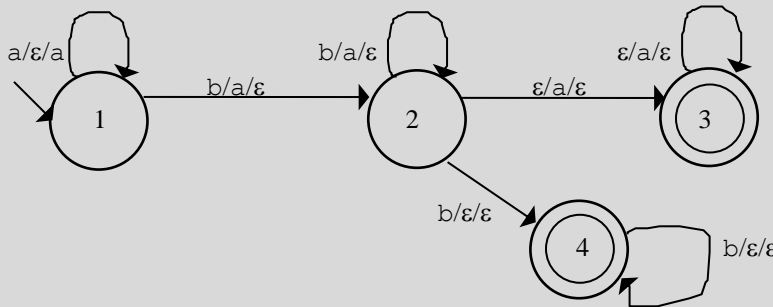
Let $L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$. Now we don't know the order in which the a's and b's will occur. They can be interleaved. So for example, any PDA to accept L must accept $aabbbba$. The only way to count the number of characters that have not yet found their mates is to use the stack. So the stack will sometimes count a's and sometimes count b's. It will count whatever it has seen more of. The following simple PDA accepts L :



This machine is highly nondeterministic. Whenever it sees an a in the input, it can either push it (which is the right thing to do if it should be counting a 's) or attempt to pop a b (which is the right thing to do if it should be counting b 's). All the computations that make the wrong guess will fail to accept since they will not succeed in clearing the stack. But if $\#_a(w) = \#_b(w)$, there will be one computation that will accept.

Example 12.7 The a Region and the b Region are Different

Let $L = \{a^m b^n : m \neq n; m, n > 0\}$. We want to build a PDA M to accept L . It is hard to build a machine that looks for something negative, like \neq . But we can break L into two sublanguages: $\{a^m b^n : 0 < m < n\}$ and $\{a^m b^n : 0 < n < m\}$. Either there are more a 's or more b 's. M must accept any string that is in either of those sublanguages. So M is:



As long as M sees a 's, it stays in state 1 and pushes each a onto the stack. When it sees the first b , it goes to state 2. It will accept nothing but b 's from that point on. So far, its behavior has been deterministic. But, from state 2, it must make choices. Each time it sees another b and there is an a on the stack, it should consume the b and pop the a and stay in state 2. But, in order to accept, it must eventually either read at least one b that does not have a matching a or pop an a that does not have a matching b . It should do the former (and go to state 4) if there is a b in the input stream when the stack is empty. But we have no way to specify that a move can be taken only if the stack is empty. It should do the latter (and go to state 3) if there is an a on the stack but the input stream is empty. But we have no way to specify that the input stream is empty.

As a result, in most of its moves in state 2, M will have a choice of three paths to take. All but the correct one will die out without accepting. But a good deal of computational effort will be wasted first.

In the next section, we present techniques for reducing nondeterminism caused by the two problems we've just presented:

- a transition that should be taken only if the stack is empty, and
- a transition that should be taken only if the input stream is empty.

But first we present one additional example of the power of nondeterminism.

Example 12.8 $\neg A^n B^n C^n$

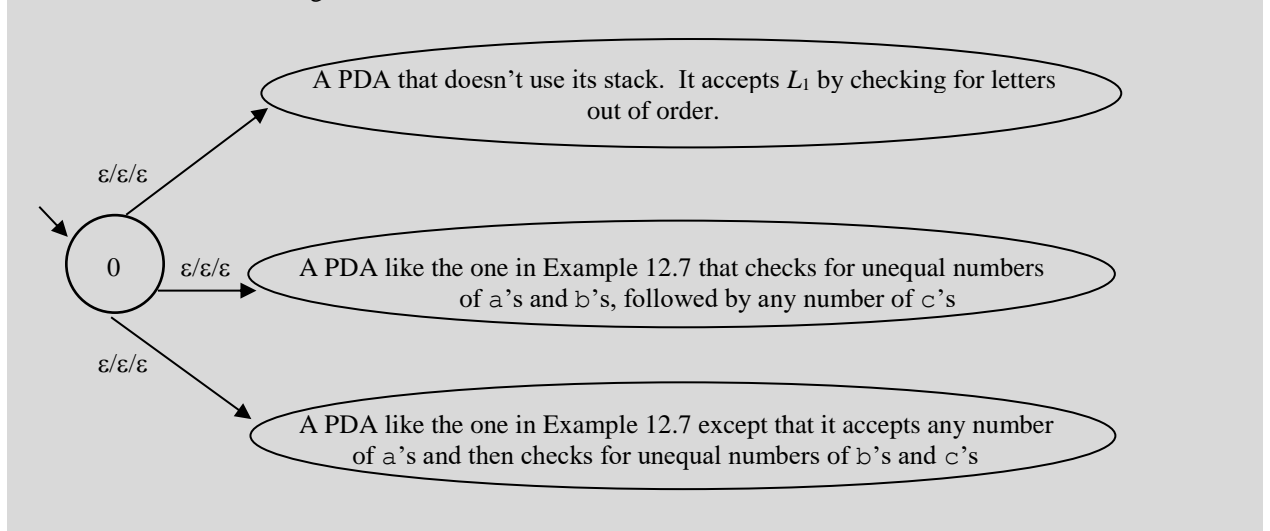
Let's first consider $A^n B^n C^n = \{a^n b^n c^n : n \geq 0\}$. If we try to think about building a PDA to accept $A^n B^n C^n$, we immediately run into trouble. We can use the stack to count a 's and then compare them to the b 's. But then the stack

will be empty and it won't be possible to compare the c 's. We can try to think of something clever to get around this problem, but we will fail. We'll prove in Chapter 13 that no PDA exists to accept this language.

But now let $L = \neg A^n B^n C^n$. There is a PDA that accepts L . $L = L_1 \cup L_2$, where:

- $L_1 = \{w \in \{a, b, c\}^* : \text{the letters are out of order}\}$.
- $L_2 = \{a^i b^j c^k : i, j, k \geq 0 \text{ and } (i \neq j \text{ or } j \neq k)\}$ (in other words, not equal numbers of a 's, b 's, and c 's).

A simple FSM can accept L_1 . So we focus on L_2 . It turns out to be easier to check for a mismatch in the number of a 's, b 's, and c 's than to check for a match because, to detect a mismatch, it is sufficient to find one thing wrong. It is not necessary to compare everything. So a string w is in L_2 iff *either* (or both) the a 's and b 's don't match or the b 's and c 's don't match. We can build PDAs, such as the one we built in Example 12.7, to check each of those conditions. So we can build a straightforward PDA for L . It first guesses which condition to check for. Then submachines do the checking. We sketch a PDA for L here and leave the details as an exercise:



This last example is significant for two reasons:

- It illustrates the power of nondeterminism.
- It proves that the class of languages acceptable by PDAs is not closed under complement. We'll have more to say about that in Section 13.4.

An important fact about the context-free languages, in contrast to the regular ones, is that nondeterminism is more than a convenient design tool. In Section 13.5 we will define the *deterministic context-free languages* to be those that can be accepted by some deterministic PDA that may exploit an end-of-string marker. Then we will prove that there are context-free languages that are not deterministic in this sense. Thus there exists, for the context-free languages, no equivalent of the regular language algorithm *ndfsmtodfsm*. There are, however, some techniques that can be used to reduce nondeterminism in many of the kinds of cases that often occur. We'll sketch two of them in the next section.

12.2.3 Techniques for Reducing Nondeterminism ❖

In Example 12.7, we saw nondeterminism arising from two very specific circumstances:

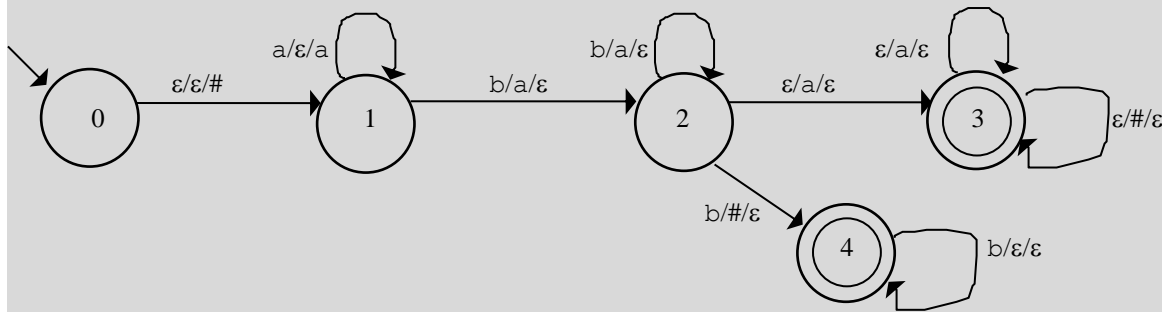
- A transition that should be taken only if the stack is empty competes against one or more moves that require a match of some string on the stack, and
- A transition that should be taken only if the input stream is empty competes against one or more moves that require a match against a specific input character.

Both of these circumstances are common, so we would like to find a way to reduce or eliminate the nondeterminism that they cause.

We first consider the case in which the nondeterminism could be eliminated if it were possible to check for an empty stack. Although our PDA model does not provide a way to do that directly, it is easy to simulate. Any PDA M that would like to be able to check for empty stack can simply, before it does anything else, push a special character onto the stack. The stack is then logically empty iff that special character is at the top of the stack. The only thing we must be careful about is that, before M can accept a string, its stack must be completely empty. So the special character must be popped whenever M reaches an accepting state.

Example 12.9 Using a Bottom of Stack Marker

We can use the special, bottom-of-stack marker technique to reduce the nondeterminism in the PDA that we showed in Example 12.7. We'll use # as the marker. When we do that, we get the following PDA M' :

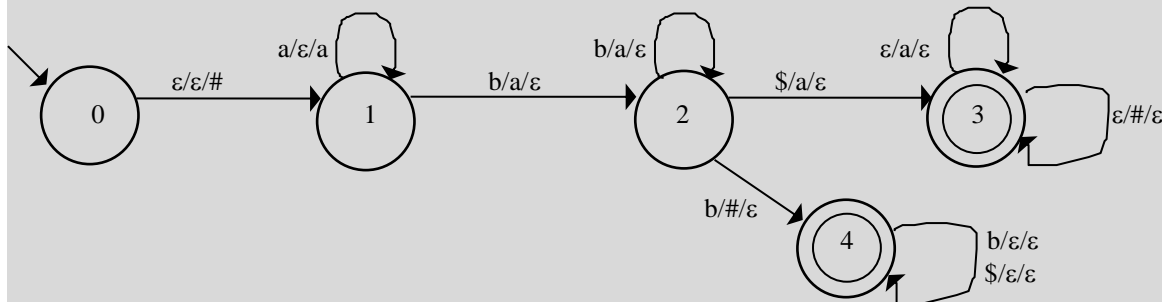


Now the transition back to state 2 no longer competes with the transition to state 4, which can only be taken when the # is the only symbol left on the stack. M' is still nondeterministic though, because the transition back to state 2 competes with the transition to state 3. We still don't have a way to specify that M' should go to state 3 only if it has run out of input.

Next we consider the “out of input” problem. To solve that one, we will make a change to the input language. Instead of building a machine to accept a language L , we'll build one to accept $L\$$, where \$ is a special end-of-string marker. In any practical system, we would probably choose <newline> or <cr> or <enter>, rather than \$, but we'll use \$ here because it is easy to see.

Example 12.10 Using an End-of-String Marker

We can use the end-of-string marker technique to eliminate the remaining nondeterminism in the PDAs that we showed in Example 12.7 and Example 12.9. When we do that, we get the following PDA M'' :



Now the transition back to state 2 no longer competes with the transition to state 3, since the latter can only be taken when the \$ is read. Notice that we must be careful to read the \$ on all paths, not just the one where we needed it.

Adding an end-of-string marker to the language to be accepted is a powerful tool for reducing nondeterminism. In Section 13.5, we'll define the class of deterministic context-free languages to be exactly the set of context-free languages L such that $L\$$ can be accepted by some deterministic PDA. We'll do that because, for practical reasons, we would like the class of deterministic context-free languages to be as large as possible.

12.3 Equivalence of Context-Free Grammars and PDAs

So far, we have shown PDAs to accept several of the context-free languages for which we wrote grammars in Chapter 11. This is no accident. In this section we'll prove, as usual by construction, that context-free grammars and pushdown automata describe exactly the same class of languages.

12.3.1 Building a PDA from a Grammar

Theorem 12.1 For Every CFG There Exists an Equivalent PDA

Theorem: Given a context-free grammar $G = (V, \Sigma, R, S)$, there exists a PDA M such that $L(M) = L(G)$.

Proof: The proof is by construction. There are two equally straightforward ways to do this construction, so we will describe both of them. Either of them can be converted to a practical parser (a recognizer that returns a parse tree if it accepts) by adding simple tree-building operations associated with each stack operation. We'll see how in Chapter 15.

Top-down parsing: A top-down parser answers the question, "Could G generate w ?" by starting with S , applying the rules of R , and seeing whether w can be derived. We can build a PDA that does exactly that. We will define the algorithm $cfgtoPDA_{topdown}(G)$, which, from a grammar G , builds a corresponding PDA M that, on input w , simulates G attempting to produce a leftmost derivation of w . M will have two states. The only purpose of the first state is to push S onto the stack and then go to the second state. M 's stack will actually do all the work by keeping track of what G is trying to derive. Initially, of course, that is S , which is why M begins by pushing S onto the stack. But suppose that R contains a rule of the form $S \rightarrow \gamma_1\gamma_2\dots\gamma_n$. Then M can replace its goal of generating an S by the goal of generating a γ_1 , followed by a γ_2 , and so forth. So M can pop S off the stack and replace it by the sequence of symbols $\gamma_1\gamma_2\dots\gamma_n$ (with γ_1 on top). As long as the symbol on the top of the stack is a nonterminal in G , this process continues, effectively applying the rules of G to the top of the stack (thus producing a left-most derivation).

The appearance of a terminal symbol c on the top of the stack means that G is attempting to generate c . M only wants to pursue paths that generate its input string w . So, at that point, it pops the top symbol off the stack, reads its next input character, and compares the two. If they match, the derivation that M is pursuing is consistent with generating w and the process continues. If they don't match, the path that M is currently following ends without accepting. So, at each step, M either applies a grammar rule, without consuming any input, or it reads an input character and pops one terminal symbol off the stack.

When M has finished generating each of the constituents of the S it pushed initially, its stack will become empty. If that happens at the same time that M has read all the characters of w , G can generate w , so M accepts. It will do so since its second state will be an accepting state. Parsers with a structure like M 's are called top-down parsers. We'll have more to say about them in Section 15.2.

As an example, suppose that R contains the rules $A \rightarrow a$, $B \rightarrow b$ and $S \rightarrow AAB$. Assume that the input to M is aab . Then M first shifts S onto the stack. Next it applies its third rule, pops S off, and replaces it by AAB . Then it applies its first rule, pops off A , and replaces it by a . The stack is then aAB . At that point, it reads the first character of its input, pops a , compares the two characters, sees that they match, and continues. The stack is then AB . Again M applies its first rule, pops off A , and replaces it by a . The stack then is aB . Then it reads the next character of its input, pops a , compares the two characters, sees that they match, and continues. The stack is then B . M applies its second rule, pops off B , and replaces it by b . It reads the last input character, pops off b , compares the two characters, and sees that they match. At that point, M is in an accepting state and both the stack and the input stream are empty, so M accepts. The outline of M is shown in Figure 12.2.



Figure 12.2 A PDA that parses top-down

Formally, $M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$, where Δ contains:

- The start-up transition $((p, \epsilon, \epsilon), (q, S))$, which pushes the start symbol onto the stack and goes to state q .
- For each rule $X \rightarrow \gamma_1\gamma_2\dots\gamma_n$ in R , the transition $((q, \epsilon, X), (q, \gamma_1\gamma_2\dots\gamma_n))$, which replaces X by $\gamma_1\gamma_2\dots\gamma_n$. If $n = 0$ (i.e., the right-hand side of the rule is ϵ), then the transition is $((q, \epsilon, X), (q, \epsilon))$.
- For each character $c \in \Sigma$, the transition $((q, c, c), (q, \epsilon))$, which compares an expected character from the stack against the next input character and continues if they match.

So we can define:

$cfgtoPDA_{topdown}(G: CFG) =$
From G , construct M as defined above.

Bottom-up parsing: A bottom-up parser answers the question, “Could G generate w ?” by starting with w , applying the rules of R backwards, and seeing whether S can be reached. We can build a PDA that does exactly that. We will define the algorithm $cfgtoPDA_{bottomup}(G)$, which, from a grammar G , builds a corresponding PDA M that, on input w , simulates the construction, backwards, of a rightmost derivation of w in G . Again, M will have two states, but this time all the work will happen in the first one. In the top-down approach that we described above, the entries in the stack corresponded to expectations: to constituents that G was trying to derive. In the bottom-up approach that we are describing now, the objects in the stack will correspond to constituents that have actually been found in the input. If M ever finds a complete S that covers its entire input, then it should accept. So if, when M runs out of input, the stack contains a single S , it will accept.

M will be able to perform two kinds of actions:

- M can read an input symbol and **shift** it onto the stack.
- Whenever a sequence of elements at the top of the stack matches, in reverse, the right-hand side of some rule r in R , M can pop that sequence off and replace it by the left-hand side of r . When this happens, we say that M has **reduced** by rule r .

Because of the two actions that it can perform, a parser based on a PDA like M is called a **shift-reduce parser**. We’ll have more to say about how such parsers work in Section 15.3. For now, we just observe that they simulate, backwards, a right-most derivation.

To see how M might work, suppose that R contains the rules $A \rightarrow a$, $B \rightarrow b$ and $S \rightarrow AAB$. Assume that the input to M is aab . Then M first shifts a onto the stack. The top of the stack matches the right-hand side of the first rule. So M can apply the rule, pop off a , and replace it with A . Then it shifts the next a , so the stack is aA . It reduces by the first rule again, so the stack is AA . It shifts the b , applies the second rule, and leaves the stack as BAA . At that point, the top of the stack matches, in reverse, the right-hand side of the third rule. The string is reversed because the leftmost symbol was read first and so is at the bottom of the stack. M will pop off BAA and replace it by S .

To accept, M must pop S off the stack, leave the stack empty, and go to its second state, which will accept. The outline of M is shown in Figure 12.3.

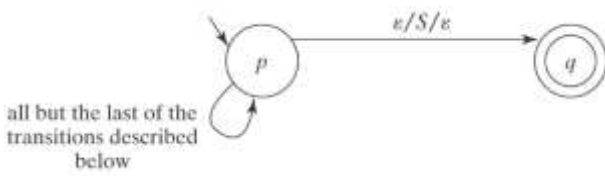


Figure 12.3 A PDA that parses bottom-up

Formally, $M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$, where Δ contains:

- The shift transitions: $((p, c, \epsilon), (p, c))$, for each $c \in \Sigma$.
- The reduce transitions: $((p, \epsilon, (\gamma_1\gamma_2\dots\gamma_n)^R), (p, X))$, for each rule $X \rightarrow \gamma_1\gamma_2\dots\gamma_n$ in R .
- The finish up transition: $((p, \epsilon, S), (q, \epsilon))$.

So we can define:

$cfptoPDAbottomup(G: CFG) =$
From G , construct M as defined above.

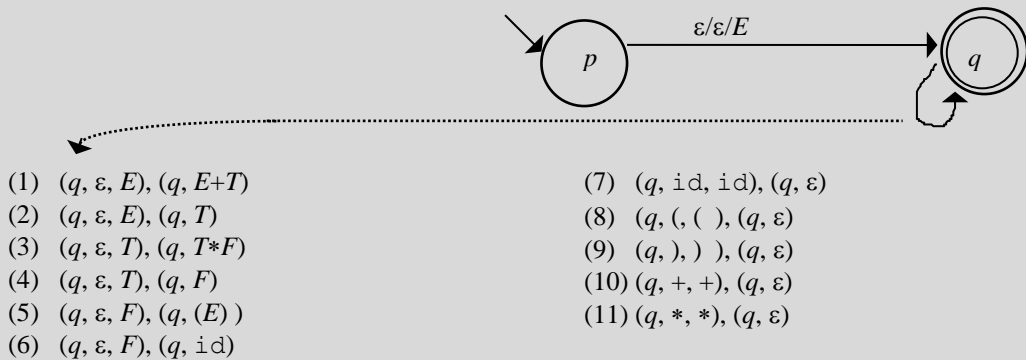


Example 12.11 Using $cfptoPDAtopdown$ and $cfptoPDAbottomup$

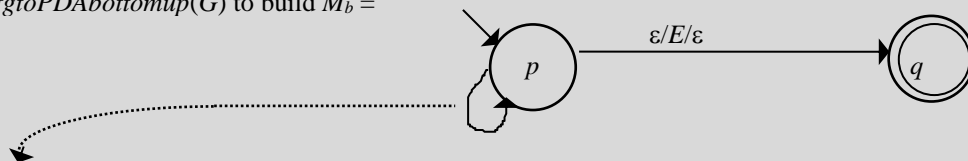
Consider E_{expr} , our simple expression language, defined by $G = \{\{E, T, F, \text{id}, +, *, (,)\}, \{\text{id}, +, *, (,)\}, R, E\}$, where:

- $R =$
- $E \rightarrow E + T$
 - $E \rightarrow T$
 - $T \rightarrow T * F$
 - $T \rightarrow F$
 - $F \rightarrow (E)$
 - $F \rightarrow \text{id}$ }

We show two PDAs, M_a and M_b , that accept E_{expr} . We can use $cfptoPDAtopdown(G)$ to build M_a :



We can use $cfptoPDAbottomup(G)$ to build $M_b =$



(1) $(p, \text{id}, \varepsilon), (p, \text{id})$	(6) $(p, \varepsilon, T + E), (p, E)$
(2) $(p, (, \varepsilon), (p, ($	(7) $(p, \varepsilon, T), (p, E)$
(3) $(p,), \varepsilon), (p,))$	(8) $(p, \varepsilon, F * T), (p, T)$
(4) $(p, +, \varepsilon), (p, +)$	(9) $(p, \varepsilon, F), (p, T)$
(5) $(p, *, \varepsilon), (p, *)$	(10) $(p, \varepsilon,)E(), (p, F)$
	(11) $(p, \varepsilon, \text{id}), (p, F)$

The theorem that we just proved is important for two very different kinds of reasons:

- It is theoretically important because we will use it to prove one direction of the claim that context-free grammars and PDAs describe the same class of languages. For this purpose, all we care about is the truth of the theorem.
- It is of great practical significance. The languages we use to communicate with programs are, in the main, context-free. Before an application can assign meaning to our programs, our queries, and our marked up documents, it must parse the statements that we have written. Consider either of the PDAs that we built in our proof of this theorem. Each stack operation of either of them corresponds to the building of a piece of the parse tree that corresponds to the derivation that the PDA found. So we can go a long way toward building a parser by simply augmenting one of the PDAs that we just built with a mechanism that associates a tree-building operation with each stack action. Because the PDAs follow the structure of the grammar, we can guarantee that we get the parses we want by writing appropriate grammars. In truth, building efficient parsers is more complicated than this. We'll have more to say about the issues in Chapter 15.

12.3.2 Building a Grammar from a PDA ✦

We next show that it is possible to go the other way, from a PDA to a grammar. Unfortunately, the process is not as straightforward as the grammar-to-PDA process. Fortunately, for applications, it is rarely (if ever) necessary to go in this direction.

Restricted Normal Form

The grammar-creation algorithm that we are about to define must make some assumptions about the structure of the PDA to which it is applied. So, before we present that algorithm, we will define what we'll call **restricted normal form** for PDAs. A PDA M is in restricted normal form iff:

- 1) M has a start state s' that does nothing except push a special symbol onto the stack and then transfer to a state s from which the rest of the computation begins. There must be no transitions back to s' . The special symbol must not be an element of Γ . We will use $\#$ to stand for such a symbol.
- 2) M has a single accepting state a . All transitions into a pop $\#$ and read no input.
- 3) Every transition in M , except the one from s' , pops exactly one symbol from the stack.

As with other normal forms, in order for restricted normal form to be useful, we must define an algorithm that converts an arbitrary PDA $M = (K, \Sigma, \Gamma, \Delta, s, A)$ into it. Given M , *convertpdatorestricted* builds a new PDA M' such that $L(M') = L(M)$ and M' is in restricted normal form.

convertpdatorestricted(M : PDA) =

1. Initially, let $M' = M$.

/* Establish property 1:

2. Create a new start state s' .
3. Add the transition $((s', \varepsilon, \varepsilon), (s, \#))$.

/* Establish property 2:

4. Create a new accepting state a .
5. For each accepting state q in M do:
 - 5.1. Create the transition $((q, \varepsilon, \#), (a, \varepsilon))$.

5.2. Remove q from the set of accepting states (making a the only accepting state in M).

/* Establish property 3:

/* Assure that no more than one symbol is popped at each transition:

6. For every transition t that pops k symbols, where $k > 1$ do:

6.1. Replace t with k transitions, each of which pops a single symbol. Create additional states as necessary to do this. Only if the last of the k symbols can be popped should any input be read or any new symbols pushed. Specifically, let $qq_1, qq_2, \dots, qq_{k-1}$ be new state names. Then:

Replace $((q_1, c, \gamma_1 \gamma_2 \dots \gamma_n), (q_2, \gamma_P))$ with:

$$((q_1, \varepsilon, \gamma_1), (qq_1, \varepsilon)), ((qq_1, \varepsilon, \gamma_2), (qq_2, \varepsilon)), \dots, ((qq_{k-1}, c, \gamma_n), (q_2, \gamma_P)).$$

/* Assure that exactly one symbol is popped at each transition. We already know that no more than one will be. But perhaps none were. In that case, what M' needs to do instead is to pop whatever was on the top of the stack and then just push it right back. So we'll need one new transition for every symbol that might be on the top of the stack. Note that, because of existence of the bottom of stack marker $\#$, we are guaranteed that the stack will not be empty so there will always be a symbol that can be popped.

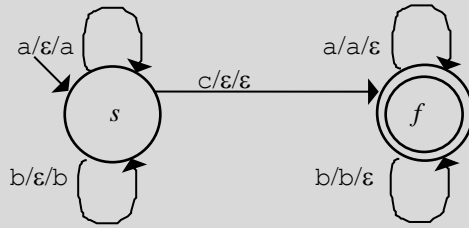
7. For every transition $t = ((q_1, c, \varepsilon), (q_2, \gamma))$ do:

7.1. Replace t with $|\Gamma_M|$ transitions, each of which pops a single symbol and then pushes it back on. Specifically, for each symbol α in $\Gamma_M \cup \{\#\}$, add the transition $((q_1, c, \alpha), (q_2, \gamma\alpha))$.

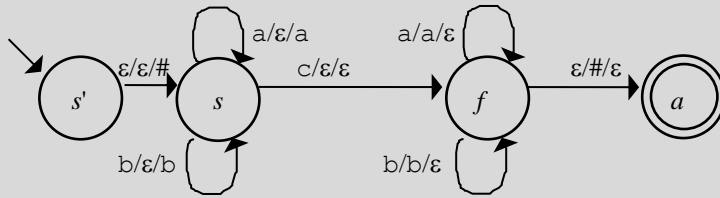
8. Return M' .

Example 12.12 Converting to Restricted Normal Form

Let $WcW^R = \{wCw^R : w \in \{a, b\}^*\}$. A straightforward PDA M that accepts WcW^R is the one we showed in Example 12.3:



M is not in restricted normal form. To create an equivalent PDA M' , we first create new start and accepting states and connect them to M :



M' contains no transitions that pop more than one symbol. And it contains no transitions that push more than one symbol. But it does contain transitions that pop nothing. Since $\Gamma_{M'} = \{a, b, \#\}$, the three transitions from state s must be replaced by the following nine transitions:

$$\begin{aligned} &((s, a, \#), (s, a\#)), ((s, a, a), (s, aa)), ((s, a, b), (s, ab)), \\ &((s, b, \#), (s, b\#)), ((s, b, a), (s, ba)), ((s, b, b), (s, bb)), \\ &((s, c, \#), (f, \#)), ((s, c, a), (f, a)), ((s, c, b), (f, b)). \end{aligned}$$

Building the Grammar

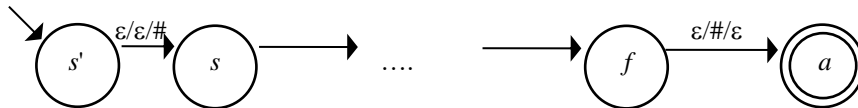
Since we have now shown that any PDA can be converted into an equivalent one in restricted normal form, we can show that, for any PDA M , there exists a context-free grammar that generates $L(M)$ by first converting M to restricted normal form and then constructing a grammar.

Theorem 12.2 For Every PDA There Exists an Equivalent CFG

Theorem: Given a PDA $M = (K, \Sigma, \Gamma, \Delta, s, A)$, there exists a CFG $G = (V, \Sigma, R, S)$ such that $L(G) = L(M)$.

Proof: The proof is by construction. In the proof of Theorem 12.1, we showed how to use a PDA to simulate a grammar. Now we show how to use a grammar to simulate a PDA. The basic idea is simple: the productions of the grammar will simulate the moves of the PDA. Unfortunately, the details get messy.

The first step of the construction of G will be to build from M , using the algorithm *convertpdatorestricted* that we just defined, an equivalent PDA M' , where M' is in restricted normal form. So every machine that the grammar-construction algorithm must deal with will look like this (with the part in the middle that actually does the work indicated with ...):



G , the grammar that we will build, will exploit a collection of nonterminal symbols to which we will give names of the following form:

$$\langle q_i, \gamma, q_j \rangle.$$

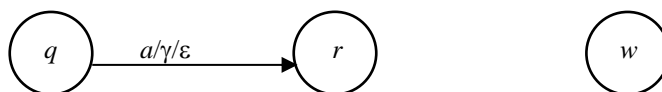
The job of a nonterminal $\langle q_i, \gamma, q_j \rangle$ is to generate all and only the strings that can drive M from state q_i with the symbol γ on the stack to state q_j , having popped off the stack γ and anything else that got pushed on top of it in the process of going from q_i to q_j . So, for example, in the machine M' that we described above in Example 12.12, the job of $\langle s, \#, a \rangle$ is to generate all the strings that could take M' from s with $\#$ on the top of the stack to a , having popped the $\#$ (and anything else that got pushed along the way) off the stack. But notice that that is exactly the set of strings that M' will accept. So G will contain the rule:

$$S \rightarrow \langle s, \#, a \rangle.$$

Now we need to describe the rules that will have $\langle s, \#, a \rangle$ on their left-hand sides. They will make use of additional nonterminals. For example, M' from Example 12.12 must go through state f on its way to a . So there will be the nonterminal $\langle f, \#, a \rangle$, which describes the set of strings that can drive M' from f to a , popping $\#$. That set is, of course, $\{\varepsilon\}$.

How can an arbitrary machine M get from one state to another? Because M is in restricted normal form, we must consider only the following three kinds of transitions, all of which pop exactly one symbol:

- Transitions that push no symbols: Suppose that there is a such a transition $((q, a, \gamma), (r, \varepsilon))$, where $a \in \Sigma \cup \{\varepsilon\}$. We consider how such a transition can participate in a computation of M :



If this transition is taken, then M reads a , pops γ , and then moves to r . After doing that, it may follow any available paths from r to any next state w , where w may be q or r or any other state. So consider the nonterminal $\langle q, \gamma, w \rangle$,

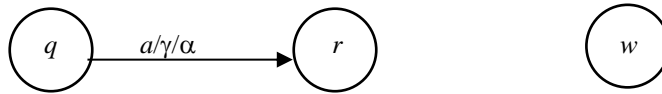
for any state w . Its job is to generate all strings that drive M from q to w while popping off γ . We now know how to describe at least some of those strings: They are the ones that start with a and are followed by any string that could drive M from r to w without popping anything (since the only thing we need to pop, γ , has already been popped). So we can write the rule:

$$\langle q, \gamma, w \rangle \rightarrow a \langle r, \varepsilon, w \rangle.$$

Read this rule to say that M can go from q to w , leaving the stack just as it was except that a γ on the top has been popped, by reading a , popping γ , going to r , and then somehow getting from r to w , leaving the stack just as it was. Since M reads a , G must generate it.

Every transition in M of the form $((q, a, \gamma), (r, \varepsilon))$ generates one grammar rule, like the one above, for every state w in M , except s' .

- Transitions that push one symbol: This situation is similar to the case where M pushes no symbols except that whatever computation follows must pop the symbol that this transition pushes. So, suppose that M contains:



If this transition is taken, then M reads the character a , pops γ , pushes α , and then moves to r . After doing that, it may follow any available paths from r to any next state w , where w may be q or r or any other state. So consider the nonterminal $\langle q, \gamma, w \rangle$, for any state w . Its job is to generate all strings that drive M from q to w while popping off γ . We now know how to describe at least some of those strings: They are the ones that start with a and are followed by any string that could drive M from r to w while popping the α that just got pushed. So we can write the rule:

$$\langle q, \gamma, w \rangle \rightarrow a \langle r, \alpha, w \rangle.$$

Read this rule to say that M can go from q to w , leaving the stack just as it was except that a γ on the top has been popped, by reading a , popping γ , pushing α , going to r , and then somehow getting from r to w , leaving the stack just as it was except that a α on the top has been popped.

Every transition in M of the form $((q, a, \gamma), (r, \alpha))$ generates one grammar rule, like the one above, for every state w in M , except s' .

- Transitions that push two symbols: This situation is a bit more complicated since two symbols are pushed and must then be popped.



If this transition is taken, then M reads a , pops γ , pushes two characters $\alpha\beta$, and then moves to r . Now suppose that we again want to consider strings that drive M from q to w , where the only change to the stack is to pop the γ that gets popped on the way from q to r . This time, two symbols have been pushed, so both must subsequently be popped. Since M is in restricted normal form, it can pop only a single symbol on each transition. So the only way to go from r to w and pop both symbols is to visit another state in between the two. Call it v , as shown in the figure. We now know how to describe at least some of the strings that drive M from q to w , popping γ : They are the ones that start with a and are followed first by any string that could drive M from r to v while popping α and then by any string that could drive M from v to w while popping β . So we can write the rule:

$$\langle q, \gamma, w \rangle \rightarrow a \langle r, \alpha, v \rangle \langle v, \beta, w \rangle$$

Every transition in M of the form $((q, a, X), (r, \alpha\beta))$ generates one grammar rule, like the one above, for every pair of states v and w in M , except s' . Note that v and w may be the same and either or both of them could be q or r .

- Transitions that push more than two symbols: These transitions can be treated by extending the technique for two symbols, adding one additional state for each additional symbol.

The last situation that we need to consider is how to stop. So far, every rule we have created has some nonterminal on its right-hand side. If G is going to generate strings composed solely of terminal symbols, it must have a way to eliminate the final nonterminals once all the terminal symbols have been generated. It can do this with one rule for every state q in M :

$$\langle q, \varepsilon, q \rangle \rightarrow \varepsilon.$$

Read these rules to say that M can start in q , remain in q , having popped nothing, without consuming any input.

We can now define $buildgrammar(M)$, which assumes that M is in restricted normal form:

$buildgrammar(M: \text{PDA in restricted normal form}) =$

1. Set Σ_G to Σ_M .
2. Set the start symbol of G to S .
3. Build R as follows:
 - 3.1. Insert the rule $S \rightarrow \langle s, \# a \rangle$.
 - 3.2. For every transition $((q, a, \gamma), (r, \varepsilon))$ (i.e., every transition that pushes no symbols), except the one from s' ; and every state w , except s' ; in M do:
Insert the rule $\langle q, \gamma, w \rangle \rightarrow a \langle r, \varepsilon, w \rangle$.
 - 3.3. For every transition $((q, a, \gamma), (r, \alpha))$ (i.e., every transition that pushes one symbol), except the one from s' ; and every state w , except s' ; in M do:
Insert the rule $\langle q, \gamma, w \rangle \rightarrow a \langle r, \alpha, w \rangle$.
 - 3.4. For every transition $((q, a, \gamma), (r, \alpha\beta))$ (i.e., every transition that pushes two symbols), except the one from s' ; and every pair of states v and w , except s' ; in M do:
Insert the rule $\langle q, \gamma, w \rangle \rightarrow a \langle r, \alpha, v \rangle \langle v, \beta, w \rangle$.
 - 3.5. In a similar way, create rules for transitions that push more than two symbols.
 - 3.6. For every state q , except s' ; in M do:
Insert the rule $\langle q, \varepsilon, q \rangle \rightarrow \varepsilon$.
4. Set V_G to $\Sigma_M \cup \{\text{nonterminal symbols mentioned in the rules inserted into } R\}$.

The algorithm $buildgrammar$ creates all the nonterminals and all the rules required for G to generate exactly the strings in $L(M)$. We should note, however, that it generally also creates many nonterminals that are useless because they are either unreachable or unproductive (or both). For example, suppose that, in M , there is a transition $((q_6, a, \gamma), (q_7, \alpha))$ from state q_6 to state q_7 , but no path from state q_7 to state q_8 . Nevertheless, in step 3.3, $buildgrammar$ will insert the rule $\langle q_6, \gamma, q_8 \rangle \rightarrow a \langle q_7, \alpha, q_8 \rangle$. $\langle q_7, \alpha, q_8 \rangle$ is unproductive since there are no strings that drive M from q_7 to q_8 .

Finally, for an arbitrary PDA M , we define $PDAtoCFG$:

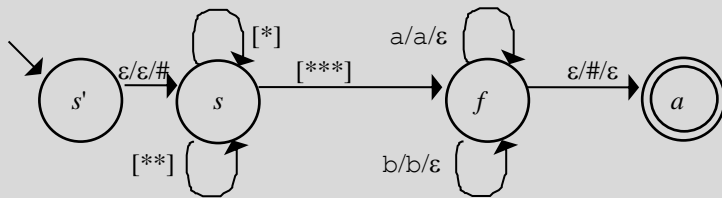
$PDAtoCFG(M: \text{PDA}) =$

1. Return $buildgrammar(convertpdatorestricted(M))$.

■

Example 12.13 Building a Grammar from a PDA

In Example 12.12, we showed a simple PDA for $WcW^R = \{wcw^R; w \in \{a, b\}^*\}$. Then we converted that PDA to restricted normal form and got M' :



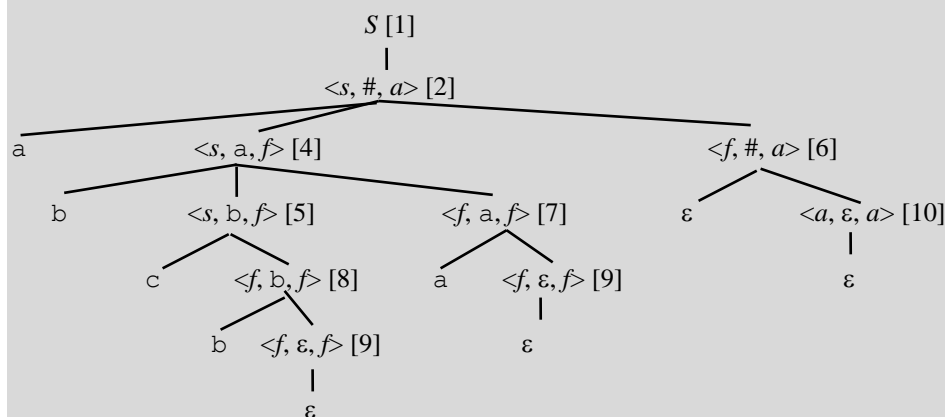
Each of the bracket-labeled arcs corresponds to:

[*] $((s, a, \#), (s, a\#)), ((s, a, a), (s, aa)), ((s, a, b), (s, ab)),$

[**] $((s, b, \#), (s, b\#)), ((s, b, a), (s, ba)), ((s, b, b), (s, bb)),$

[***] $((s, c, \#), (f, \#)), ((s, c, a), (f, a)), ((s, c, b), (f, b)).$

Buildgrammar constructs a grammar G from M' . To see how G works, consider the parse tree that it builds for the input string $abcba$. The numbers in brackets at each node indicate the rule that is applied to the nonterminal at the node.



Here are some of the rules in G . On the left are the transitions of M' . The middle column contains the rules derived from each transition. The ones marked [x] in the right column contain useless nonterminals and so cannot be part of any derivation of a string in $L(G)$. Because there are so many useless rules, we have omitted the ones generated from all transitions after the first.

	$S \rightarrow \langle s, \#, a \rangle$	[1]
	$((s', \varepsilon, \varepsilon), (s, \#))$	no rules based on the transition from s''
[*]	$((s, a, \#), (s, a\#))$	$\langle s, \#, s \rangle \rightarrow a \langle s, a, s \rangle \langle s, \#, s \rangle$ [x]
		$\langle s, \#, s \rangle \rightarrow a \langle s, a, f \rangle \langle f, \#, s \rangle$ [x]
		$\langle s, \#, s \rangle \rightarrow a \langle s, a, a \rangle \langle a, \#, s \rangle$ [x]
		$\langle s, \#, f \rangle \rightarrow a \langle s, a, s \rangle \langle s, \#, f \rangle$ [x]
		$\langle s, \#, f \rangle \rightarrow a \langle s, a, f \rangle \langle f, \#, f \rangle$ [x]
		$\langle s, \#, f \rangle \rightarrow a \langle s, a, a \rangle \langle a, \#, f \rangle$ [x]
		$\langle s, \#, a \rangle \rightarrow a \langle s, a, s \rangle \langle s, \#, a \rangle$ [x]
		$\langle s, \#, a \rangle \rightarrow a \langle s, a, f \rangle \langle f, \#, a \rangle$ [2]
		$\langle s, \#, a \rangle \rightarrow a \langle s, a, a \rangle \langle a, \#, a \rangle$ [x]
	$((s, a, a), (s, aa))$	$\langle s, a, f \rangle \rightarrow a \langle s, a, f \rangle \langle f, a, f \rangle$ [3]
	$((s, a, b), (s, ab))$	$\langle s, b, f \rangle \rightarrow a \langle s, a, f \rangle \langle f, b, f \rangle$ [14]
[**]	$((s, b, \#), (s, b\#))$	$\langle s, \#, f \rangle \rightarrow b \langle s, b, f \rangle \langle f, \#, f \rangle$ [15]
	$((s, b, a), (s, ba))$	$\langle s, a, f \rangle \rightarrow b \langle s, b, f \rangle \langle f, a, f \rangle$ [4]
	$((s, b, b), (s, bb))$	$\langle s, b, f \rangle \rightarrow b \langle s, b, f \rangle \langle f, b, f \rangle$ [16]

[***]	$((s, c, \#), (f, \#))$	$\langle s, \#, f \rangle \rightarrow c \langle f, \#, f \rangle$	[17]
	$((s, c, a), (f, a))$	$\langle s, a, f \rangle \rightarrow c \langle f, a, f \rangle$	[18]
	$((s, c, b), (f, b))$	$\langle s, b, f \rangle \rightarrow c \langle f, b, f \rangle$	[5]
	$((f, \varepsilon, \#), (a, \varepsilon))$	$\langle f, \#, a \rangle \rightarrow \varepsilon \langle a, \varepsilon, a \rangle$	[6]
	$((f, a, a), (f, \varepsilon))$	$\langle f, a, f \rangle \rightarrow a \langle f, \varepsilon, f \rangle$	[7]
	$((f, b, b), (f, \varepsilon))$	$\langle f, b, f \rangle \rightarrow b \langle f, \varepsilon, f \rangle$	[8]
		$\langle s, \varepsilon, s \rangle \rightarrow \varepsilon$	[19]
		$\langle f, \varepsilon, f \rangle \rightarrow \varepsilon$	[9]
		$\langle a, \varepsilon, a \rangle \rightarrow \varepsilon$	[10]

12.3.3 The Equivalence of Context-free Grammars and PDAs

Theorem 12.3 PDAs and CFGs Describe the Same Class of Languages

Theorem: A language is context-free iff it is accepted by some PDA.

Proof: Theorem 12.1 proves the only if part. Theorem 12.2 proves the if part. ■

12.4 Nondeterminism and Halting

Recall that a computation C of a PDA $M = (K, \Sigma, \Gamma, \Delta, s, A)$ on a string w is an accepting computation iff:

$$C = (s, w, \varepsilon) \vdash_{-M}^* (q, \varepsilon, \varepsilon), \text{ for some } q \in A.$$

We'll say that a computation C of M **halts** iff at least one of the following conditions holds:

- C is an accepting computation, or
- C ends in a configuration from which there is no transition in Δ that can be taken.

We'll say that M **halts** on w iff every computation of M on w halts. If M halts on w and does not accept, then we say that M rejects w .

For every context-free language L , we've proven that there exists a PDA M such that $L(M) = L$. Suppose that we would like to be able to:

- Examine a string and decide whether or not it is in L .
- Examine a string that is in L and create a parse tree for it.
- Examine a string that is in L and create a parse tree for it in time that is linear in the length of the string.
- Examine a string and decide whether or not it is in the complement of L .

Do PDAs provide the tools we need to do those things? When we were at a similar point in our discussion of regular languages, the answer to that question was yes. For every regular language L , there exists a minimal deterministic FSM that accepts it. That minimal DFSM halts on all inputs, accepts all strings that are in L , and rejects all strings that are not in L .

Unfortunately, the facts about context-free languages and PDAs are different from the facts about regular languages and FSMs. Now we must face the following:

- 1) There are context-free languages for which no deterministic PDA exists. We'll prove this as Theorem 13.13.
- 2) It is possible that a PDA may
 - not halt, or
 - not ever finish reading its input.

So, let M be a PDA that accepts some language L . Then, on input w , if $w \in L$ then M will halt and accept. But if $w \notin L$, while M will not accept w , it is possible that it will not reject it either. To see how this could happen, let $\Sigma = \{a\}$ and consider the PDA M , shown in Figure 12.4. $L(M) = \{a\}$. The computation $(1, a, \epsilon) \vdash (2, a, a) \vdash (3, \epsilon, \epsilon)$ will cause M to accept a . But consider any other input except a . Observe that:

- M will never halt. There is no accepting configuration, but there is always at least one computation that has not yet halted. For example, on input aa , one computation is:
 $(1, aa, \epsilon) \vdash (2, aa, a) \vdash (1, aa, aa) \vdash (2, aa, aaa) \vdash (1, aa, aaaa) \vdash (2, aa, aaaaa) \vdash \dots$
 - M will never finish reading its input unless its input is ϵ . On input aa , for example, there is no computation that will read the second a .
- 3) There exists no algorithm to minimize a PDA. In fact, it is undecidable whether a PDA is already minimal.

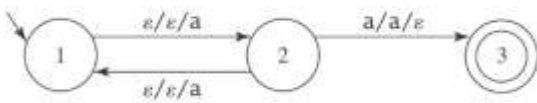


Figure 12.4 A PDA that might neither accept nor reject

Problem 2 is especially critical. Of course, the same problem also arose with NDFSMs. But there we had a choice of two solutions:

- Use *ndfsmtodfsm* to convert the NDFSM to an equivalent deterministic one. A DFSM halts on input w in $|w|$ steps.
- Simulate the NDFSM using *ndfsmsimulate*, which ran all computational paths in parallel and handled ϵ -transitions in a way that guaranteed that the simulation of an NDFSM M on input w halted in $|w|$ steps.

Neither of those approaches works for PDAs. There may not be an equivalent deterministic PDA. And it is not possible to simulate all paths in parallel on a single PDA because each path would need its own stack. So what can we do? Solutions to these problems fall into two classes:

- formal ones that do not restrict the class of languages that are being considered. Unfortunately, these approaches generally do restrict the *form* of the grammars and PDAs that can be used. For example, they may require that grammars be in Chomsky or Greibach normal form. As a result, parse trees may not make much sense. We'll see some of these techniques in Chapter 14.
- practical ones that work only on a subclass of the context-free languages. But the subset is large enough to be useful and the techniques can use grammars in their natural forms. We'll see some of these techniques in Chapters 13 and 15.

12.5 Alternative Equivalent Definitions of a PDA ♦

We could have defined a PDA somewhat differently. We list here a few reasonable alternative definitions. In all of them a PDA M is a sextuple $(K, \Sigma, \Gamma, \Delta, s, A)$:

- We allow M to pop and to push any string in Γ^* . In some definitions, M may pop only a single symbol but it may push any number of them. In some definitions, M may pop and push only a single symbol.
- In our definition, M accepts its input w only if, when it finishes reading w , it is in an accepting state and its stack is empty. There are two alternatives to this:
 - Accept if, when the input has been consumed, M lands in an accepting state, regardless of the contents of the stack.
 - Accept if, when the input has been consumed, the stack is empty, regardless of the state M is in.

All of these definitions are equivalent in the sense that, if some language L is accepted by a PDA using one definition, it can be accepted by some PDA using each of the other definitions.

We can prove this claim for any pair of definitions by construction. To do so, we show an algorithm that transforms a PDA of one sort into an equivalent PDA of the other sort.

Example 12.14 Accepting by Final State Alone

Define a PDA $M = (K, \Sigma, \Gamma, \Delta, s, A)$ in exactly the way we have except that it will accept iff it lands in an accepting state, regardless of the contents of the stack. In other words, if $(s, w, \varepsilon) \vdash_{M^*} (q, \varepsilon, \gamma)$ and $q \in A$, then M accepts.

To show that this model is equivalent to ours, we must show two things: for each of our machines, there exists an equivalent one of these, and, for each of these, there exists an equivalent one of ours. We'll do the first part to show how such a construction can be done. We leave the second as an exercise.

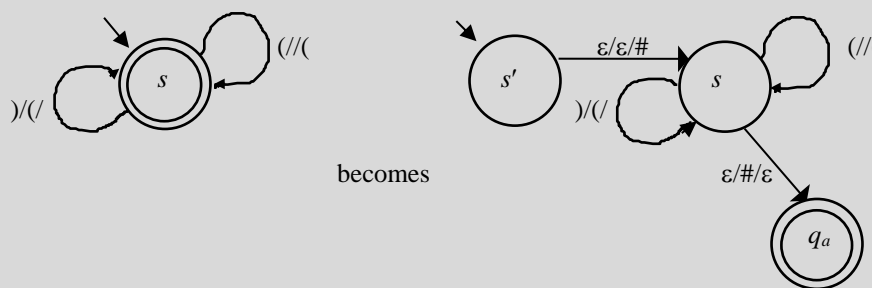
Given a PDA M that accepts by accepting state and empty stack, construct a new PDA M' that accepts by accepting state alone, where $L(M') = L(M)$. M' will have a single accepting state q_a . The only way for M' to get to q_a will be to land in an accepting state of M when the stack is logically empty. But there is no way to check that the stack is empty. So M' will begin by pushing a bottom-of-stack marker $\#$, onto the stack. Whenever $\#$ is the top symbol on the stack, the stack is logically empty.

So the construction proceeds as follows:

1. Initially, let $M' = M$.
2. Create a new start state s' . Add the transition $((s', \varepsilon, \varepsilon), (s, \#))$.
3. Create a new accepting state q_a .
4. For each accepting state a in M do:
Add the transition $((a, \varepsilon, \#), (q_a, \varepsilon))$.
5. Make q_a the only accepting state in M' .

It is easy to see that M' lands in its only accepting state (q_a) iff M lands in some accepting state with an empty stack. Thus M' and M accept the same strings.

As an example, we apply this algorithm to the PDA we built for the balanced parentheses language Bal:



Notice, by the way, that while M is deterministic, M' is not.

12.6 Alternatives that are Not Equivalent to the PDA

We defined a PDA to be a finite state machine to which we add a single stack. We mention here two variants of that definition, each of which turns out to define a more powerful class of machine. In both cases, we'll still start with an FSM.

For the first variation, we add a first-in, first-out (FIFO) queue in place of the stack. Such machines are called tag systems or Post machines. As we'll see in Section 18.2.3, tag systems are equivalent to Turing machines in computational power.

For the second variation, we add two stacks instead of one. Again, the resulting machines are equivalent in computational power to Turing machines, as we'll see in Section 17.5.2.

12.7 Exercises

- 1) Build a PDA to accept each of the following languages L :
 - a) $\text{BalDelim} = \{w : \text{where } w \text{ is a string of delimiters: } (,), [,], \{, \}, \text{ that are properly balanced}\}$.
 - b) $\{a^i b^j : 2i = 3j + 1\}$.
 - c) $\{w \in \{a, b\}^* : \#_a(w) = 2 \cdot \#_b(w)\}$.
 - d) $\{a^m b^n : m \leq n \leq 2m\}$.
 - e) $\{w \in \{a, b\}^* : w = w^R\}$.
 - f) $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } (i \neq j \text{ or } j \neq k)\}$.
 - g) $\{w \in \{a, b\}^* : \text{every prefix of } w \text{ has at least as many } a\text{'s as } b\text{'s}\}$.
 - h) $\{a^m b^n a^n : n, m \geq 0 \text{ and } m \text{ is even}\}$.
 - i) $\{x c^n : x \in \{a, b\}^*, \#_a(x) = n \text{ or } \#_b(x) = n\}$.
 - j) $\{a^m b^n : m \geq n, m - n \text{ is even}\}$.
 - k) $\{a^m b^n c^p d^q : m, n, p, q \geq 0 \text{ and } m + n = p + q\}$.
 - l) $\{b_i \# b_{i+1}^R : b_i \text{ is the binary representation of some integer } i, i \geq 0, \text{ without leading zeros}\}$. (For example $101\#011 \in L$.)
 - m) $\{x^R \# y : x, y \in \{0, 1\}^* \text{ and } x \text{ is a substring of } y\}$.
 - n) L_1^* , where $L_1 = \{x x^R : x \in \{a, b\}^*\}$.
- 2) Complete the PDA that we sketched, in Example 12.8, for $\neg A^n B^n C^n$, where $A^n B^n C^n = \{a^n b^n c^n : n \geq 0\}$.
- 3) Let $L = \{b a^{m_1} b a^{m_2} b a^{m_3} \dots b a^{m_n} : n \geq 2, m_1, m_2, \dots, m_n \geq 0, \text{ and } m_i \neq m_j \text{ for at least one } i, j \text{ pair}\}$.
 - a) Show a PDA that accepts L .
 - b) Show a context-free grammar that generates L .
 - c) Prove that L is not regular.
- 4) Consider the language $L = L_1 \cap L_2$, where $L_1 = \{w w^R : w \in \{a, b\}^*\}$ and $L_2 = \{a^n b^* a^n : n \geq 0\}$.
 - a) List the first four strings in the lexicographic enumeration of L ?
 - b) Write a context-free grammar to generate L .
 - c) Show a natural PDA for L . (In other words, don't just build it from the grammar using one of the two-state constructions presented in this chapter.)
 - d) Prove that L is not regular.
- 5) Build a deterministic PDA to accept each of the following languages:
 - a) $L\$$, where $L = \{w\$ \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$.
 - b) $L\$$, where $L = \{a^n b^+ a^m : n \geq 0 \text{ and } \exists k \geq 0 (m = 2k + n)\}$.
- 6) Complete the proof that we started in Example 12.14. Specifically, show that if M is a PDA that accepts by accepting state alone, then there exists a PDA M' that accepts by accepting state and empty stack (our definition) where $L(M') = L(M)$.

13 Context-Free and Noncontext-Free Languages

The language $A^nB^n = \{a^n b^n : n \geq 0\}$ is context-free. The language $A^nB^nC^n = \{a^n b^n c^n : n \geq 0\}$ is not context free (intuitively because a PDA's stack cannot count all three of the letter regions and compare them). $\text{PalEven} = \{ww^R : w \in \{a, b\}^*\}$ is context-free. The similar language $WW = \{ww : w \in \{a, b\}^*\}$ is not context-free (again, intuitively, because a stack cannot pop the characters of w off in the same order in which they were pushed).

Given a new language L , how can we know whether or not it is context-free? In this chapter, we present a collection of techniques that can be used to answer that question.

13.1 Where Do the Context-Free Languages Fit in the Big Picture?

First, we consider the relationship between the regular languages and the context-free languages.

Theorem 13.1 The Context-free Languages Properly Contain the Regular Languages

Theorem: The regular languages are a proper subset of the context-free languages.

Proof: We first show that every regular language is context-free. We then show that there exists at least one context-free language that is not regular.

We show that every regular language is context-free by construction. If L is regular, then it is accepted by some DFMSM $M = (K, \Sigma, \delta, s, A)$. From M we construct a PDA $M' = (K', \Sigma', \Gamma', \Delta', s', A')$ to accept L . In essence, M' will simply be M and will ignore the stack. Let M' be $(K, \Sigma, \emptyset, \Delta', s, A)$, where Δ' is constructed as follows: For every transition (q_i, c, q_j) in δ , add to Δ' the transition $((q_i, c, \varepsilon), (q_j, \varepsilon))$. M' behaves identically to M , so $L(M) = L(M')$. So the regular languages are a subset of the context-free languages.

The regular languages are a *proper* subset of the context-free languages because there exists at least one language, A^nB^n , that is context-free but not regular. ■

Next, we observe that there are *many* more noncontext-free languages than there are context-free ones:

Theorem 13.2 How Many Context-free Languages are There?

Theorem: There is a countably infinite number of context-free languages.

Proof: Every context-free language is generated by some context-free grammar $G = (V, \Sigma, R, S)$. We can encode the elements of V as binary strings, so we can lexicographically enumerate all the syntactically legal context-free grammars. There cannot be more context-free languages than there are context-free grammars, so there are at most a countably infinite number of context-free languages. There is not a one-to-one relationship between context-free languages and context-free grammars since there is an infinite number of grammars that generate any given language. But, by Theorem 13.1, every regular language is context-free. And, by Theorem 8.1, there is a countably infinite number of regular languages. So there is at least and at most a countably infinite number of context-free languages. ■

But, by Theorem 2.3, there is an uncountably infinite number of languages over any nonempty alphabet Σ . So there are many more noncontext-free languages than there are regular ones.

13.2 Showing That a Language is Context-Free

We have so far seen two techniques that can be used to show that a language L is context-free:

- Exhibit a context-free grammar for it.
- Exhibit a (possibly nondeterministic) PDA for it.

There are also closure theorems for context-free languages and they can be used to show that a language is context-free if it can be described in terms of other languages whose status is already known. Unfortunately, there are fewer closure theorems for the context-free languages than there are for the regular languages. In order to be able to discuss both the closure theorems that exist, as well as the ones we'd like but don't have, we will wait and consider the issue of closure theorems in Section 13.4, after we have developed a technique for showing that a language is not context-free.

13.3 The Pumping Theorem for Context-Free Languages

Suppose we are given a language and we want to prove that it is not context-free. Just as with regular languages, it is not sufficient simply to claim that we tried to build a grammar or a PDA and we failed. That doesn't show that there isn't some other way to approach the problem.

Instead, we will again approach this problem from the other direction. We will articulate a property that is provably true of all context-free languages. Then, if we can show that a language L does not possess this property, then we know that L is not context-free. So, just as we did when we used the Pumping Theorem for regular languages, we will construct *proofs by contradiction*. We will say, "If L were context-free, then it would possess certain properties. But it does not possess those properties. Therefore, it is not context-free."

This time we exploit the fact that every context-free language is generated by some context-free grammar. The argument we are about to make is based on the structure of parse trees. Recall that a parse tree, derived by a grammar $G = (V, \Sigma, R, S)$, is a rooted, ordered tree in which:

- Every leaf node is labeled with an element of $\Sigma \cup \{\epsilon\}$,
- The root node is labeled S ,
- Every other node is labeled with some element of $V - \Sigma$, and
- If m is a nonleaf node labeled X and the children of m are labeled x_1, x_2, \dots, x_n , then the rule $X \rightarrow x_1 x_2 \dots, x_n$ is in R .

Consider an arbitrary parse tree, as shown in Figure 13.1. The *height* of a tree is the length of the longest path from the root to any leaf. The *branching factor* of a tree is the largest number of daughters of any node in the tree. The *yield* of a tree is the ordered sequence of its leaf nodes.

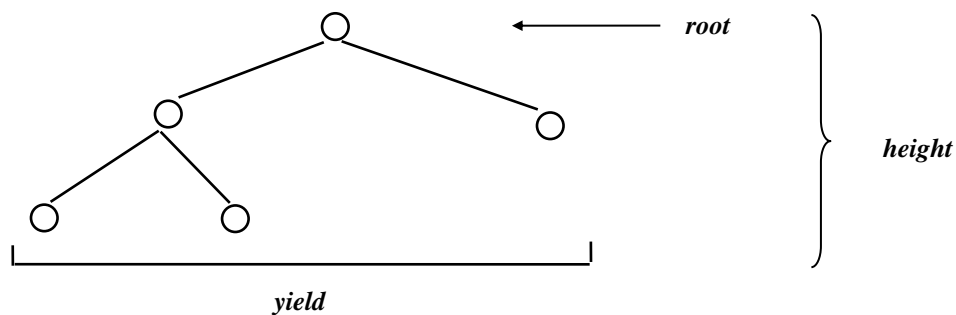


Figure 13.1 The structure of a parse tree

Theorem 13.3 The Height of a Tree and its Branching Factor Put a Bound on its Yield

Theorem: The length of the yield of any tree T with height h and branching factor b is $\leq b^h$.

Proof: The proof is by induction on h . If h is 1, then just a single rule applies. So the longest yield is of length less than or equal to b . Assume the claim is true for $h = n$. We show that it is true for $h = n + 1$. Consider any tree with $h = n + 1$. It consists of a root, and some number of subtrees, each of which is of height $\leq n$. By the induction hypothesis, the length of the yield of each of those subtrees is $\leq b^n$. The number of subtrees of the root is $\leq b$. So the length of the yield must be $\leq b(b^n) = b^{n+1} = b^h$. ■

Let $G = (V, \Sigma, R, S)$ be context-free grammar. Let $n = |V - \Sigma|$ be the number of nonterminal symbols in G . Let b be the branching factor of G , defined to be the length of the longest right-hand side of any rule in R .

Now consider any parse tree T generated by G . Suppose that no nonterminal appears more than once on any path from the root of T to a nonterminal. Then the height of T is $\leq n$. So the longest string that could correspond to the yield of T has length $\leq b^n$.

Now suppose that w is a string in $L(G)$ and $|w| > b^n$. Then any parse tree that G generates for w must contain at least one path that contains at least one repeated nonterminal. Another way to think of this is that, to derive w , G must have used at least one recursive rule. So any parse tree for w must look like the one shown in Figure 13.2, where X is some repeated nonterminal. We use dotted lines to make it clear that the derivation may not be direct but may, instead, require several steps. So, for example, it is possible that the tree shown here was derived using a grammar that contained the rules $X \rightarrow aYb$, $Y \rightarrow bXa$, and $X \rightarrow ab$.

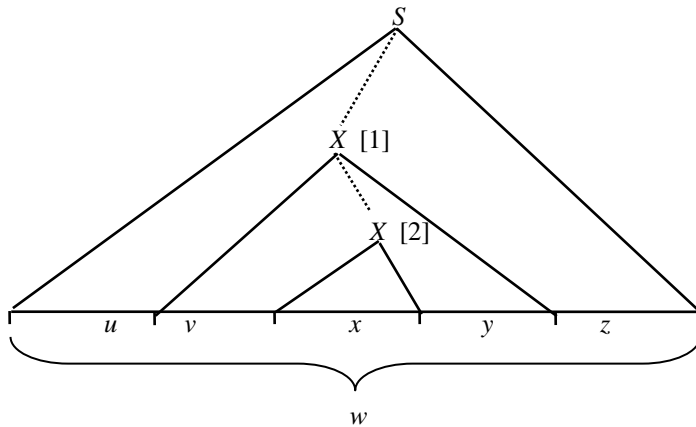


Figure 13.2 A parse tree whose height is greater than n

Of course, it is possible that w has more than one parse tree. For the rest of this discussion we will pick some tree such that G generates no other parse tree for w that has fewer nodes. Within that tree it is possible that there are many repeated nonterminals and that some of them are repeated more than once. We will assume only that we have chosen point [1] in the tree such that X is the first repeated nonterminal on any path, coming up from the bottom, in the subtree rooted at [1]. We'll call the rule that was applied at [1] $rule_1$ and the rule that was applied at [2] $rule_2$.

We can sketch the derivation that produced this tree as:

$$S \Rightarrow^* uXz \Rightarrow^* uvXyz \Rightarrow^* uvxyz.$$

In the diagram, we have carved w up into five pieces: u , v , x , y , and z . We observe that:

- There is another derivation in G , $S \Rightarrow^* uXz \Rightarrow^* uxz$, in which, at the point labeled [1], the nonrecursive $rule_2$ is used. So uxz is also in $L(G)$.

- There are infinitely many derivations in G , such as $S \Rightarrow^* uXz \Rightarrow^* uvXyz \Rightarrow^* uvvXyyz \Rightarrow^* uvvxyyz$, in which the recursive $rule_1$ is applied one or more additional times before the nonrecursive $rule_2$ is used. Those derivations produce the strings, uv^2xy^2z , uv^3xy^3z , etc. So all of those strings are also in $L(G)$.
- It is possible that $v = \varepsilon$, as it would be, for example if $rule_1$ were $X \rightarrow Xa$. It is also possible that $y = \varepsilon$, as it would be, for example if $rule_1$ were $X \rightarrow aX$. But it is not possible that both v and y are ε . If they were, then the derivation $S \Rightarrow^* uXz \Rightarrow^* uxz$ would also yield w and it would create a parse tree with fewer nodes. But that contradicts the assumption that we started with a tree with the smallest possible number of nodes.
- The height of the subtree rooted at [1] is at most $n + 1$ (since there is one repeated nonterminal and every other nonterminal can occur no more than once). So $|vxy| \leq b^{n+1}$.

These observations are the basis for the context-free Pumping Theorem, which we state next.

Theorem 13.4 The Pumping Theorem for Context-Free Languages

Theorem: If L is a context-free language, then:

$$\exists k \geq 1 (\forall \text{ strings } w \in L, \text{ where } |w| \geq k (\exists u, v, x, y, z (w = uvxyz, \\ vy \neq \varepsilon, \\ |vxy| \leq k, \text{ and} \\ \forall q \geq 0 (uv^qxy^qz \text{ is in } L))))).$$

Proof: The proof is the argument that we gave above: If L is context-free, then it is generated by some context-free grammar $G = (V, \Sigma, R, S)$ with n nonterminal symbols and branching factor b . Let k be b^{n+1} . Any string that can be generated by G and whose parse tree contains no paths with repeated nonterminals must have length less than or equal to b^n . Assuming that $b \geq 2$, it must be the case that $b^{n+1} > b^n$. So let w be any string in $L(G)$ where $|w| \geq k$. Let T be any smallest parse tree for w (i.e., a parse tree such that no other parse tree for w has fewer nodes). T must have height at least $n + 1$. Choose some path in T of length at least $n + 1$. Let X be the bottom-most repeated nonterminal along that path. Then w can be rewritten as $uvxyz$ as shown in the tree diagram above. The tree rooted at [1] has height at most $n + 1$. Thus its yield, vxy , has length less than or equal to b^{n+1} , which is k . Further, $vy \neq \varepsilon$ since if vy were ε then there would be a smaller parse tree for w and we chose T so that that wasn't so. Finally, v and y can be pumped: uxz must be in L because $rule_2$ could have been used immediately at [1]. And, for any $q \geq 1$, uv^qxy^qz must be in L because $rule_1$ could have been used q times before finally using $rule_2$. ■

So, if L is a context-free language, every “long” string in L must be pumpable. Just as with the Pumping Theorem for regular languages, the pumped region can be pumped out once or pumped in any number of times, in all cases resulting in another string that is also in L . So, if there is even one “long” string in L that is not pumpable, then L is not context-free.

Note that the value k plays two roles in the Pumping Theorem. It defines what we mean by a “long” string and it imposes an upper bound on $|vxy|$. When we set k to b^{n+1} , we guaranteed that it was large enough so that we could prove that it served both of those purposes. But we should point out that a smaller value would have sufficed as the definition for a “long” string, since any string of length greater than b^n must be pumpable.

There are a few important ways in which the context-free Pumping Theorem differs from the regular one:

- The most obvious is that two regions, v and y , must be pumped in tandem.
- We don't know anything about where the strings v and y will fall. All we know is that they are reasonably “close together”, i.e., $|vxy| \leq k$.
- Either v or y could be empty, although not both.

Example 13.1 $A^nB^nC^n$ is Not Context-Free

Let $L = A^nB^nC^n = \{a^n b^n c^n : n \geq 0\}$. We can use the Pumping Theorem to show that L is not context-free. If it were, then there would exist some k such that any string w , where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let $w = a^k b^k c^k$, where k is the constant from the Pumping Theorem. For w to satisfy the conditions of the Pumping Theorem, there must be some u, v, x, y , and z such that $w = uvxyz$, $vy \neq \epsilon$, $|vxy| \leq k$, and $\forall q \geq 0 (uv^qxy^qz \text{ is in } L)$. We show that no such u, v, x, y , and z exist. If either v or y contains two or more different characters, then set q to 2 (i.e., pump in once) and the resulting string will have letters out of order and thus not be in $A^nB^nC^n$. (For example, if v is $aabb$ and y is cc , then the string that results from pumping will look like $aaa...aaabbaabbcccc...ccc$.) If both v and y each contain at most one distinct character then set q to 2. Additional copies of at most two different characters are added, leaving the third unchanged. There are no longer equal numbers of the three letters, so the resulting string is not in $A^nB^nC^n$. There is no way to divide w into $uvxyz$ such that all the conditions of the Pumping Theorem are met. So $A^nB^nC^n$ is not context-free.

As with the Pumping Theorem for regular languages, it requires some skill to design simple and effective proofs using the context-free Pumping Theorem. As before, the choices that we can make, when trying to show that a language L is not context-free are:

- We choose w , the string to be pumped. It is important to choose w so that it is in the part of L that captures the essence of why L is not context-free.
- We choose a value for q that shows that w isn't pumpable.
- We may apply closure theorems before we start, so that we show that L is not context-free by showing that some other language L' isn't. We'll have more to say about this technique later.

Example 13.2 The Language of Strings with n^2 a's is Not Context-Free

Let $L = \{a^{n^2}, n \geq 0\}$. We can use the Pumping Theorem to show that L is not context-free. If it were, then there would exist some k such that any string w , where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let n (in the definition of L) be k^2 . So $n^2 = k^4$ and $w = a^{k^4}$. For w to satisfy the conditions of the Pumping Theorem, there must be some u, v, x, y , and z , such that $w = uvxyz$, $vy \neq \epsilon$, $|vxy| \leq k$, and $\forall q \geq 0 (uv^qxy^qz \text{ is in } L)$. We show that no such u, v, x, y , and z exist. Since w contains only a's, $vy = a^p$, for some nonzero p . Set q to 2. The resulting string, which we'll call s , is a^{k^4+p} , which must be in L . But it isn't because it is too short. If a^{k^4} , which contains $(k^2)^2$ a's, is in L , then the next longer element of L contains $(k^2 + 1)^2$ a's. That's $k^4 + 2k^2 + 1$ a's. So there are no strings in L with length between k^4 and $k^4 + 2k^2 + 1$. But $|s| = k^4 + p$. So, for s to be in L , $p = |vy|$ would have to be at least $2k^2 + 1$. But $|vxy| \leq k$, so p can't be that large. Thus s is not in L . There is no way to divide w into $uvxyz$ such that all the conditions of the Pumping Theorem are met. So L is not context-free.

When using the Pumping Theorem, we focus on v and y . Once they are specified, so are u, x , and z .

To show that there exists no v, y pair that satisfies all of the conditions of the Pumping Theorem, it is sometimes necessary to enumerate a set of cases and rule them out one at a time. Sometimes the easiest way to do this is to imagine the string to be pumped as divided into a set of regions. Then we can consider all the ways in which v and y can fall across those regions.

Example 13.3 Dividing the String w Into Regions

Let $L = \{a^n b^m a^n : n, m \geq 0 \text{ and } n \geq m\}$. We can use the Pumping Theorem to show that L is not context-free. If it were, then there would exist some k such that any string w , where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let $w = a^k b^k a^k$, where k is the constant from the Pumping Theorem. For w to satisfy the conditions of the Pumping Theorem, there must be some u, v, x, y , and z , such that $w = uvxyz$, $vy \neq \epsilon$, $|vxy| \leq k$, and $\forall q \geq 0 (uv^qxy^qz \text{ is in } L)$. We show that no such u, v, x, y , and z exist. Imagine w divided into three regions as follows:

aaa ... aaabbb ... bbbaaa ... aaa
 | 1 | 2 | 3 |

We consider all the cases for where v and y could fall and show that in none of them are all the conditions of the theorem met:

- If either v or y crosses regions, then set q to 2 (thus pumping in once). The resulting string will have letters out of order and so not be in L . So in all the remaining cases we assume that v and y each falls within a single region.
- (1, 1): both v and y fall in region 1. Set q to 2. In the resulting string, the first group of a's is longer than the second group of a's. So the string is not in L .
- (2, 2): both v and y fall in region 2. Set q to 2. In the resulting string, the b region is longer than either of the a regions. So the string is not in L .
- (3, 3): both v and y fall in region 3. Set q to 0. The same argument as for (1, 1).
- (1, 2): nonempty v falls in region 1 and nonempty y falls in region 2. (If either v or y is empty, it does not matter where it falls. So we can treat it as though it falls in the same region as the nonempty one. We have already considered all of those cases.) Set q to 2. In the resulting string, the first group of a's is longer than the second group of a's. So the string is not in L .
- (2, 3): nonempty v falls in region 2 and nonempty y falls in region 3. Set q to 2. In the resulting string the second group of a's is longer than the first group of a's. So the string is not in L .
- (1, 3): nonempty v falls in region 1 and nonempty y falls in region 3. If this were allowed by the other conditions of the Pumping Theorem, we could pump in a's and still produce strings in L . But if we pumped out, we would violate the requirement that the a regions be at least as long as the b region. More importantly, this case violates the requirement that $|vxy| \leq k$. So it need not be considered.

There is no way to divide w into $uvxyz$ such that all the conditions of the Pumping Theorem are met. So L is not context-free.

Consider the language $\text{PalEven} = \{ww^R : w \in \{a, b\}^*\}$, the language of even-length palindromes of a's and b's, which we introduced in Example 11.3. Let w be any string in PalEven . Then substrings of w are related to each other in a perfectly nested way, as shown in Figure 13.3 (a). Nested relationships of this sort can naturally be described with a context-free grammar, so languages whose strings are structured in this way are typically context-free.

But now consider the case in which the relationships are not properly nested but instead cross. For example, consider the language $\text{WcW} = \{w_cw : w \in \{a, b\}^*\}$. Now let w be any string in WcW . Then substrings of w are related to each other as shown in Figure 13.3 (b). We call such dependencies, where lines cross each other, **cross-serial dependencies**. Languages whose strings are characterized by cross serial dependencies are typically not context-free.

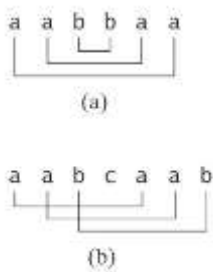


Figure 13.3 Nested versus cross-serial dependencies

Example 13.4 WcW is Not Context-Free

Let $\text{WcW} = \{w_cw : w \in \{a, b\}^*\}$. WcW is not context-free because its strings contain cross-serial dependencies.

We can use the Pumping Theorem to show that. If it were, then there would exist some k such that any string w , where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let $w = a^k b^k c a^k b^k$, where k is the constant from the Pumping Theorem. For w to satisfy the conditions of the Pumping Theorem, there must be

some $u, v, x, y,$ and $z,$ such that $w = uvxyz, v \neq \epsilon, |vxy| \leq k,$ and $\forall q \geq 0 (uv^qxy^qz \text{ is in } W \subset W).$ We show that no such $u, v, x, y,$ and z exist. Imagine w divided into five regions as follows:

```

aaa ... aaabbb ... bbbcaaa ... aaabbb ... bbb
|   1   |   2   |3|   4   |   5   |

```

Call the part before the c the left side and the part after the c the right side. We consider all the cases for where v and y could fall and show that in none of them are all the conditions of the theorem met:

- If either v or y overlaps region 3, set q to 0. The resulting string will no longer contain a c and so is not in $W \subset W$.
- If both v and y occur before region 3 or they both occur after region 3, then set q to 2. One side will be longer than the other and so the resulting string is not in $W \subset W$.
- If either v or y overlaps region 1, then set q to 2. In order to make the right side match, something would have to be pumped into region 4. But any v, y pair that did that would violate the requirement that $|vxy| \leq k$.
- If either v or y overlaps region 2, then set q to 2. In order to make the right side match, something would have to be pumped into region 5. But any v, y pair that did that would violate the requirement that $|vxy| \leq k$.

There is no way to divide w into $uvxyz$ such that all the conditions of the Pumping Theorem are met. So $W \subset W$ is not context-free.

Are programming languages like C++ and Java context-free? $\text{C } 666.$

The language $W \subset W$, which we just showed is not context-free, is important because of its similarity to the structure of many common programming languages. Consider a programming language that requires that variables be declared before they are used. If we consider just a single variable w , then a program that declares w and then uses it has a structure very similar to the strings in the language $W \subset W$, since the string w must occur in exactly the same form in both the declaration section and the body of the program.

13.4 Some Important Closure Properties of Context-Free Languages

It helps to be able to analyze a complex language by decomposing it into simpler pieces. Closure theorems, when they exist, enable us to do that. We'll see in this section that, while the context-free languages are closed under some common operations, we cannot prove as strong a set of closure theorems as we were able to prove for the regular languages.

13.4.1 The Closure Theorems

Theorem 13.5 Closure under Union, Concatenation, Kleene Star, Reverse, and Letter Substitution

Theorem: The context-free languages are closed under union, concatenation, Kleene star, reverse, and letter substitution.

Proof: We prove each of the claims separately by construction:

- The context-free languages are closed under union: if L_1 and L_2 are context-free languages, then there exist context-free grammars $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$ such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$. If necessary, rename the nonterminals of G_1 and G_2 so that the two sets are disjoint and so that neither includes the symbol S . We will build a new grammar G such that $L(G) = L(G_1) \cup L(G_2)$. G will contain all the rules of both G_1 and G_2 . We add to G a new start symbol, S , and two new rules, $S \rightarrow S_1$ and $S \rightarrow S_2$. The two new rules allow G to generate a string iff at least one of G_1 or G_2 generates it. So $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}, S)$.
- The context-free languages are closed under concatenation: if L_1 and L_2 are context-free languages, then there exist context-free grammars $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$ such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$. If necessary, rename the nonterminals of G_1 and G_2 so that the two sets are disjoint and so that neither includes the symbol S . We will build a new grammar G such that $L(G) = L(G_1) L(G_2)$. G will contain all the rules of both G_1

and G_2 . We add to G a new start symbol, S , and one new rule, $S \rightarrow S_1 S_2$. So $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2\}, S)$.

- The context-free languages are closed under Kleene star: if L_1 is a context-free language, then there exists a context-free grammar $G_1 = (V_1, \Sigma_1, R_1, S_1)$ such that $L_1 = L(G_1)$. If necessary, rename the nonterminals of G_1 so that V_1 does not include the symbol S . We will build a new grammar G such that $L(G) = L(G_1)^*$. G will contain all the rules of G_1 . We add to G a new start symbol, S , and two new rules, $S \rightarrow \varepsilon$ and $S \rightarrow S S_1$. So $G = (V_1 \cup \{S\}, \Sigma_1, R_1 \cup \{S \rightarrow \varepsilon, S \rightarrow S S_1\}, S)$.
- The context-free languages are closed under reverse: recall that $L^R = \{w \in \Sigma^* : w = x^R \text{ for some } x \in L\}$. If L is a context-free language, then it is generated by some Chomsky normal form grammar $G = (V, \Sigma, R, S)$. Every rule in G is of the form $X \rightarrow BC$ or $X \rightarrow a$, where X, B , and C are elements of $V - \Sigma$ and $a \in \Sigma$. In the latter case $L(X) = \{a\}$. $\{a\}^R = \{a\}$. In the former case, $L(X) = L(B)L(C)$. By Theorem 2.4, $(L(B)L(C))^R = L(C)^R L(B)^R$. So we construct, from G , a new grammar G' , such that $L(G') = L^R$. $G' = (V_G, \Sigma_G, R', S_G)$, where R' is constructed as follows:
 - For every rule in G of the form $X \rightarrow BC$, add to R' the rule $X \rightarrow CB$.
 - For every rule in G of the form $X \rightarrow a$, add to R' the rule $X \rightarrow a$.
- The context-free languages are closed under letter substitution, defined as follows: consider any two alphabets, Σ_1 and Σ_2 . Let sub be any function from Σ_1 to Σ_2^* . Then $letsub$ is a letter substitution function from L_1 to L_2 iff $letsub(L_1) = \{w \in \Sigma_2^* : \exists y \in L_1 (w = y \text{ except that every character } c \text{ of } y \text{ has been replaced by } sub(c))\}$. We leave the proof of this as an exercise. ■

As with regular languages, we can use these closure theorems as a way to prove that a more complex language is context-free if it can be shown to be built from simpler ones using operations under which the context-free languages are closed.

Theorem 13.6 Nonclosure under Intersection, Complement, and Difference

Theorem: The context-free languages are not closed under intersection, complement, or difference.

Proof:

- The context-free languages are not closed under intersection: the proof is by counterexample. Let:

$$L_1 = \{a^n b^n c^m : n, m \geq 0\}. \quad /* \text{ equal } a\text{'s and } b\text{'s.}$$

$$L_2 = \{a^m b^n c^n : n, m \geq 0\}. \quad /* \text{ equal } b\text{'s and } c\text{'s.}$$

Both L_1 and L_2 are context-free, since there exist straightforward context-free grammars for them.

But now consider:

$$L = L_1 \cap L_2$$

$$= \{a^n b^n c^n : n \geq 0\}.$$

If the context-free languages were closed under intersection, L would have to be context-free. But we proved, in Example 13.1, that it isn't.

- The context-free languages are not closed under complement: given any sets L_1 and L_2 ,

$$L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2).$$

The context-free languages are closed under union. So, if they were also closed under complement, they would necessarily be closed under intersection. But we just showed that they are not. Thus they are not closed under complement either.

We've also seen an example that proves this claim directly. $\neg A^n B^n C^n$ is context-free. We showed a PDA that accepts it in Example 12.8. But $\neg(\neg A^n B^n C^n) = A^n B^n C^n$ is not context-free.

- The context-free languages are not closed under difference (subtraction): given any language L ,

$$\neg L = \Sigma^* - L.$$

Σ^* is context-free. So, if the context-free languages were closed under difference, the complement of any context-free language would necessarily be context-free. But we just showed that that is not so. ■

Recall that, in using the regular Pumping Theorem to show that some language L was not regular, we sometimes found it useful to begin by intersecting L with another regular language to create a new language L' . Since the regular languages are closed under intersection, L' would necessarily be regular if L were. We then showed that L' , designed to be simpler to work with, was not regular. And so neither was L .

It would be very useful to be able to exploit this technique when using the context-free Pumping Theorem. Unfortunately, as we have just shown, the context-free languages are not closed under intersection. Fortunately, however, they are closed under intersection with the regular languages. We'll prove this result next and then, in Section 13.4.2, we'll show how it can be exploited in a proof that a language is not context-free.

Theorem 13.7 Closure under Intersection with the Regular Languages

Theorem: The context-free languages are closed under intersection with the regular languages.

Proof: The proof is by construction. If L_1 is context-free, then there exists some PDA $M_1 = (K_1, \Sigma, \Gamma_1, \Delta_1, s_1, A_1)$ that accepts it. If L_2 is regular then there exists a DFSM $M_2 = (K_2, \Sigma, \delta, s_2, A_2)$ that accepts it. We construct a new PDA, M_3 that accepts $L_1 \cap L_2$. M_3 will work by simulating the parallel execution of M_1 and M_2 . The states of M_3 will be ordered pairs of states of M_1 and M_2 . As each input character is read, M_3 will simulate both M_1 and M_2 moving appropriately to a new state. M_3 will have a single stack, which will be controlled by M_1 . The only slightly tricky thing is that M_1 may contain ϵ -transitions. So M_3 will have to allow M_1 to follow them while M_2 just stays in the same state and waits until the next input symbol is read.

$M_3 = (K_1 \times K_2, \Sigma, \Gamma_1, \Delta_3, (s_1, s_2), A_1 \times A_2)$, where Δ_3 is built as follows:

- For each transition $((q_1, a, \beta), (p_1, \gamma))$ in Δ_1 ,
and each transition (q_2, a, p_2) in δ , add to Δ_3 the transition:
 $((q_1, q_2), a, \beta), ((p_1, p_2), \gamma)$.
- For each transition $((q_1, \epsilon, \beta), (p_1, \gamma))$ in Δ_1 ,
and each state q_2 in K_2 , add to Δ_3 the transition:
 $((q_1, q_2), \epsilon, \beta), ((p_1, q_2), \gamma)$.

We define *intersectPDAandFSM* as follows:

intersectPDAandFSM(M_1 : PDA, M_2 : FSM) =
Build M_3 as defined in the proof of Theorem 13.7.

Theorem 13.8 Closure under Difference with the Regular Languages

Theorem: The difference ($L_1 - L_2$) between a context-free language L_1 and a regular language L_2 is context-free.

Proof: $L_1 - L_2 = L_1 \cap \neg L_2$. If L_2 is regular, then, since the regular languages are closed under complement, $\neg L_2$ is also regular. Since L_1 is context-free, by Theorem 13.7, $L_1 \cap \neg L_2$ is context-free. ■

The last two theorems are important tools, both for showing that a language is context-free and for showing that a language is not context-free.

Example 13.5 Using Closure Theorems to Prove a Language Context-Free

Consider the perhaps contrived language $L = \{a^n b^n : n \geq 0 \text{ and } n \neq 1776\}$. Another way to describe L is that it is $\{a^n b^n : n \geq 0\} - \{a^{1776} b^{1776}\}$. $A^n B^n = \{a^n b^n : n \geq 0\}$ is context-free. We have shown both a simple grammar that generates it and a simple PDA that accepts it. $\{a^{1776} b^{1776}\}$ is finite and thus regular. So, by Theorem 13.8, L is context free.

Generalizing that example a bit, from Theorem 13.8 it follows that any language that can be described as the result of subtracting a finite number of elements from some language known to be context-free must also be context-free.

13.4.2 Using the Pumping Theorem in Conjunction with the Closure Properties

Languages that impose no specific order constraints on the symbols contained in their strings are not always context-free. But it may be hard to prove that one isn't just by using the Pumping Theorem. In such a case, it is often useful to exploit Theorem 13.7, which tells us that the context-free languages are closed under intersection with the regular languages.

Recall our notational convention from Section 13.3: (n, n) means that all nonempty substrings of vy occur in region n . This may happen either because v and y are both nonempty and they both occur in region n . Or it may happen because one or the other is empty and the nonempty one occurs in region n .

Are natural languages like English or Chinese or German context-free? © 747.

Example 13.6 WW is Not Context-Free

Let $WW = \{ww : w \in \{a, b\}^*\}$. WW is similar to $WcW = \{wcw : w \in \{a, b\}^*\}$, except that there is no longer a middle marker. Because, like WcW , it contains cross-serial dependencies, it is not context-free. We could try proving that by using the Pumping Theorem alone. Here are some attempts, using various choices for w :

- Let $w = (ab)^{2k}$. If $v = \varepsilon$ and $y = ab$, pumping works fine.
- Let $w = a^k b a^k b$. If $v = a$ and is in the first group of a 's and $y = a$ and is in the second group of a 's, pumping works fine.
- Let $w = a^k b^k a^k b^k$. Now the constraint that $|vxy| \leq k$ prevents v and y from both being in the two a regions or the two b regions. This choice of w will lead to a successful Pumping Theorem proof. But there are four regions in w and we must consider all the ways in which v and y could overlap those regions, including all those in which either or both of v and y occur on a region boundary. While it is possible to write out all those possibilities and show, one at a time, that every one of them violates at least one condition of the Pumping Theorem, there is an easier way.

If WW were context-free, then $L' = WW \cap a^* b^* a^* b^*$ would also be context-free. But it isn't, which we can show using the Pumping Theorem. If it were, then there would exist some k such that any string w , where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let $w = a^k b^k a^k b^k$, where k is the constant from the Pumping Theorem. For w to satisfy the conditions of the Pumping Theorem, there must be some u, v, x, y , and z , such that $w = uvxyz$, $vy \neq \varepsilon$, $|vxy| \leq k$, and $\forall q \geq 0 (uv^q xy^q z \text{ is in } L')$. We show that no such u, v, x, y , and z exist. Imagine w divided into four regions as follows:

aaa ... aaabbb ... bbbaaa ... aaabbb ... bbb
 | 1 | 2 | 3 | 4 |

We consider all the cases for where v and y could fall and show that in none of them are all the conditions of the theorem met:

- If either v or y overlaps more than one region, set q to 2. The resulting string will not be in $a^*b^*a^*b^*$ and so is not in L' .
- If $|vy|$ is not even then set q to 2. The resulting string will have odd length and so not be in L' . We assume in all the other cases that $|vy|$ is even.
- (1, 1), (2, 2), (1, 2): set q to 2. The boundary between the first half and the second half will shift into the first b region. So the second half will start with a b , while the first half still starts with an a . So the resulting string is not in L' .
- (3, 3), (4, 4), (3, 4): set q to 2. This time the boundary shifts into the second a region. The first half will end with an a while the second half still ends with a b . So the resulting string is not in L' .
- (2, 3): set q to 2. If $|v| \neq |y|$ then the boundary moves and, as argued above, the resulting string is not in L' . If $|v| = |y|$ then the first half contains more b 's and the second half contains more a 's. Since they are no longer the same, the resulting string is not in L' .
- (1, 3), (1, 4), and (2, 4) violate the requirement that $|vxy| \leq k$.

There is no way to divide w into $uvxyz$ such that all the conditions of the Pumping Theorem are met. So L' is not context-free. So neither is WW .

One reason that context-free grammars are typically too weak to describe musical structures is that they cannot describe constraints such as the one that defines WW . $\text{C } 778$.

Example 13.7 A Simple Arithmetic Language is Not Context-Free

Let $L = \{x \# y = z : x, y, z \in \{0, 1\}^* \text{ and, if } x, y \text{ and } z \text{ are viewed as positive binary numbers without leading zeros, then } xy = z^R\}$. For example, $100\#111=00111 \in L$. (We do this example instead of the more natural one in which we require that $xy = z$ because it seems as though it might be more likely to be context-free. As we'll see, however, even this simpler variant is not.)

If L were context-free, then $L' = L \cap 10^* \# 1^* = 0^* 1^*$ would also be context-free. But it isn't, which we can show using the Pumping Theorem. If it were, then there would exist some k such that any string w , where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let $w = 10^k \# 1^k = 0^k 1^k$, where k is the constant from the Pumping Theorem. Note that $w \in L$ because $10^k \cdot 1^k = 1^k 0^k$. For w to satisfy the conditions of the Pumping Theorem, there must be some u, v, x, y , and z , such that $w = uvxyz$, $vy \neq \epsilon$, $|vxy| \leq M$, and $\forall q \geq 0$ (uv^qxy^qz is in L). We show that no such u, v, x, y , and z exist. Imagine w divided into seven regions as follows:

1 000 ... 000 # 111 ... 111 = 000 ... 000111 ... 111
 |1| 2 |3| 4 |5| 6 | 7 |

We consider all the cases for where v and y could fall and show that in none of them are all the conditions of the theorem met:

- If either v or y overlaps region 1, 3, or 5 then set q to 0. The resulting string will not be in $10^* \# 1^* = 0^* 1^*$ and so is not in L' .
- If either v or y contains the boundary between 6 and 7, set q to 2. The resulting string will not be in $10^* \# 1^* = 0^* 1^*$ and so is not in L' . So the only cases left to consider are those where v and y each occur within a single region.
- (2, 2), (4,4), (2, 4): set q to 2. Because there are no leading zeros, changing the left side of the string changes its value. But the right side doesn't change to match. So the resulting string is not in L' .

- (6, 6), (7, 7), (6, 7): set q to 2. The right side of the equality statement changes value but the left side doesn't. So the resulting string is not in L' .
- (4, 6): note that, because of the first argument to the multiplication, the number of 1's in the second argument must equal the number of 1's after the $=$. Set q to 2. The number of 1's in the second argument changed but the number of 1's in the result did not. So the resulting string is not in L' .
- (2, 6), (2, 7), and (4, 7) violate the requirement that $|vxy| \leq k$.

There is no way to divide w into $uvxyz$ such that all the conditions of the Pumping Theorem are met. So L is not context-free.

Sometimes the closure theorems can be used to reduce the proof that a new language L is not context-free to the proof that some other language L' is not context-free, where we have already proven the case for L' .

Example 13.8 Using Intersection to Force Order Constraints

Let $L = \{w \in \{a, b, c\}^* : w \text{ has an equal number of } a\text{'s, } b\text{'s, and } c\text{'s}\}$. If L were context-free, then $L' = L \cap a^*b^*c^*$ would also be context-free. But $L' = A^nB^nC^n$, which is not context-free, so neither is L .

13.5 Deterministic Context-Free Languages

The regular languages are closed under complement, intersection, and difference. Why are the context-free languages different? In a nutshell, because the machines that accept them may necessarily be nondeterministic. Recall the technique that we used, in the proof of Theorem 8.4, to show that the regular languages are closed under complement: Given a (possibly nondeterministic) FSM M_1 , we used the following procedure to construct a new FSM M_2 such that $L(M_2) = \neg L(M_1)$:

1. From M_1 , construct an equivalent DFSM M' , using the algorithm *ndfsmtodfsm*, presented in the proof of Theorem 5.3. (If M_1 is already deterministic, $M' = M_1$.)
2. M' must be stated completely, so if it is described with an implied dead state, add the dead state and all required transitions to it.
3. Begin building M_2 by setting it equal to M' . Then swap the accepting and the nonaccepting states. So $M_2 = (K_{M'}, \Sigma, \delta_{M'}, s_{M'}, K_{M'} - A_{M'})$.

We have no PDA equivalent of *ndfsmtodfsm*, so we cannot simply adapt this construction for PDAs. Our proofs that the regular languages are closed under intersection and difference relied on the fact that they were closed under complement, so we cannot adapt those proofs here either.

We have no PDA equivalent of *ndfsmtodfsm* because there probably isn't one, as we will show shortly. Recall that, in Section 12.2, we defined a PDA M to be *deterministic* iff:

- Δ_M contains no pairs of transitions that compete with each other, and
- if q is an accepting state of M , then there is no transition $((q, \varepsilon, \varepsilon), (p, a))$ for any p or a .

In other words, M never has a choice between two or more moves, nor does it have a choice between moving and accepting. There exist context-free languages that cannot be accepted by any deterministic PDA. But suppose that we restrict our attention to the ones that can.

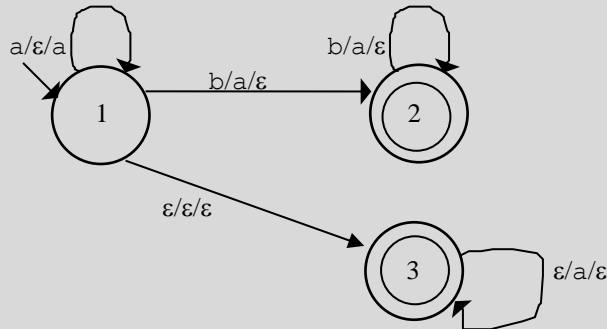
What is a Deterministic Context-Free Language?

We are about to define the class of deterministic context-free languages. Because this class is useful, we would like it to be as large as possible. So let $\$$ be an end-of-string marker. We could use any symbol that is not in Σ_L (for example $\langle \text{line feed} \rangle$ or $\langle \text{cr} \rangle$), but $\$$ is easier to read. A language L is *deterministic context-free* iff $L\$$ can be accepted by some deterministic PDA.

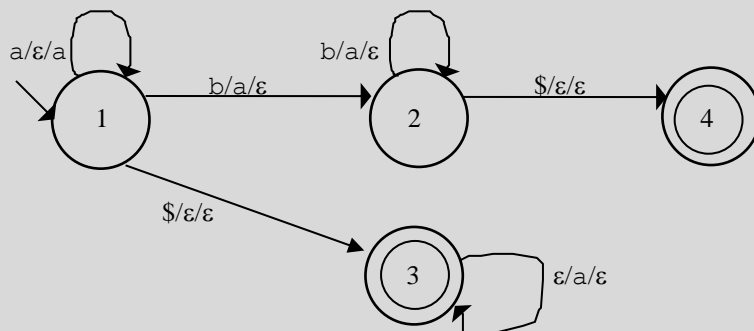
To see why we have defined the deterministic context-free languages to exploit an end-of-string marker, consider the following example of a straightforward language for which no deterministic PDA exists unless an end-of-string marker is used.

Example 13.9 Why an End-of-String Marker is Useful

Let $L = a^* \cup \{a^n b^n : n > 0\}$. Consider any PDA M that accepts L . When it begins reading a 's, M must push them onto the stack in case there are going to be b 's. But, if it runs out of input without seeing b 's, it needs a way to pop those a 's from the stack before it can accept. Without an end-of-string marker, there is no way to allow that popping to happen *only* when all the input has been read. So, for example, the following PDA accepts L , but it is nondeterministic because the transition to state 3 (where the a 's will be popped) can compete with both of the other transitions from state 1.



With an end-of-string marker, we can build the following deterministic PDA, which can only take the transition to state 3, the a -popping state, when it sees the $\$$:



Before we go any farther, we have to be sure of one thing. We introduced the end-of-string marker to make it easier to build PDAs that are deterministic. We need to make sure that it doesn't make it possible to build a PDA for a language L that was not already context-free. In other words, adding the end-of-string marker cannot convert a language that was not context-free into one that is. We do that next.

Theorem 13.9 CFLs and Deterministic CFLs

Theorem: Every deterministic context-free language (as just defined) is context-free.

Proof: If L is deterministic context-free, then $L\$$ is accepted by some deterministic PDA $M = (K, \Sigma, \Gamma, \Delta, s, A)$. From M , we construct M' such that $L(M') = L$. The idea is that, whatever M can do on reading $\$, M'$ can do on reading ϵ (i.e., by simply guessing that it is at the end of the input). But, as soon as M' makes that guess, it cannot read any more input. It may perform the rest of its computation (such as popping its stack), but any path that pretends it has seen the $\$$ before it has read all of its input will fail to accept. To enable M' to perform whatever stack operations M could have performed, but not to read any input, M' will be composed of two copies of M : the first copy will be identical to M , and M' will operate in that part of itself until it guesses that it is at the end of the input; the second copy

will be identical to M except that it contains only the transitions that do not consume any input. The states in the first copy will be labeled as in M . Those in the second copy will have the prime symbol appended to their names. So, if M contains the transition $((q, \varepsilon, \gamma_1), (p, \gamma_2))$, M' will contain the transition $((q', \varepsilon, \gamma_1), (p', \gamma_2))$. The two copies will be connected by finding, in the first copy of M , every $\$$ -transition from some state q to some state p . We replace each such transition with an ε -transition into the second copy. So the new transition goes from q to p' .

We can define the following procedure to construct M' :

without\$(M: \text{PDA}) =

1. Initially, set M' to M .
- /* Make the copy that does not read any input.
2. For every state q in M , add to M' a new state q' .
3. For every transition $((q, \varepsilon, \gamma_1), (p, \gamma_2))$ in Δ_M do:
 - 3.1 Add to $\Delta_{M'}$ the transition $((q', \varepsilon, \gamma_1), (p', \gamma_2))$.
 - /* Link up the two copies.
 4. For every transition $((q, \$, \gamma_1), (p, \gamma_2))$ in Δ_M do:
 - 4.1 Add to $\Delta_{M'}$ the transition $((q, \varepsilon, \gamma_1), (p', \gamma_2))$.
 - 4.2 Remove $((q, \$, \gamma_1), (p, \gamma_2))$ from $\Delta_{M'}$.
 - /* Set the accepting states of M' .
 5. $A_{M'} = \{q' : q \in A\}$.

■

Closure Properties of the Deterministic Context-Free Languages

The deterministic context-free languages are practically very significant because it is possible to build deterministic, linear time parsers for them. They also possess additional formal properties that are important, among other reasons, because they enable us to prove that not all context-free languages are deterministic context-free. The most important of these is that the deterministic context-free languages, unlike the larger class of context-free languages, are closed under complement.

Theorem 13.10 Closure under Complement

Theorem: The deterministic context-free languages are closed under complement.

Proof: The proof is by construction. If L is a deterministic context-free language over the alphabet Σ , then $L\$$ is accepted by some deterministic PDA $M = (K, \Sigma \cup \{\$\}, \Gamma, \Delta, s, A)$. We need to describe an algorithm that constructs a new deterministic PDA that accepts $(\neg L)\$$. To prove Theorem 8.4 (that the regular languages are closed under complement), we defined a construction that proceeded in two steps: given an arbitrary FSM, convert it to an equivalent DFSM, and then swap accepting and nonaccepting states. We can skip the first step here, but we must solve a new problem. A deterministic PDA may fail to accept an input string w for any one of several reasons:

1. Its computation ends before it finishes reading w .
2. Its computation ends in an accepting state but the stack is not empty.
3. Its computation loops forever, following ε -transitions, without ever halting in either an accepting or a nonaccepting state.
4. Its computation ends in a nonaccepting state.

If we simply swap accepting and nonaccepting states we will correctly fail to accept every string that M would have accepted (i.e., every string in $L\$$). But we will not necessarily accept every string in $(\neg L)\$$. To do that, we must also address issues 1 – 3 above.

An additional problem is that we don't want to accept $\neg L(M)$. That includes strings that do not end in $\$$. We must accept only strings that do end in $\$$ and that are in $(\neg L)\$$.

A construction that solves these problems is given in [§ 635](#).

■

What else can we say about the deterministic context-free languages? We know that they are closed under complement. What about union and intersection? We observe that $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$. So, if the deterministic context-free languages were closed under union, they would necessarily be closed under intersection also. But they are not closed under union. The context-free languages are closed under union, so the union of two deterministic context-free languages must be context-free. It may, however not be deterministic. The deterministic context-free languages are also not closed under intersection. In fact, when two deterministic context-free languages are intersected, the result may not even be context-free.

Theorem 13.11 Nonclosure under Union

Theorem: The deterministic context-free languages are not closed under union.

Proof: We show a counterexample to the claim that they are closed:

$$\text{Let } L_1 = \{a^i b^j c^k : i, j, k \geq 0 \text{ and } i \neq j\}.$$

$$\text{Let } L_2 = \{a^i b^j c^k : i, j, k \geq 0 \text{ and } j \neq k\}.$$

$$\begin{aligned} \text{Let } L' &= L_1 \cup L_2. \\ &= \{a^i b^j c^k : i, j, k \geq 0 \text{ and } (i \neq j) \text{ or } (j \neq k)\}. \end{aligned}$$

$$\begin{aligned} \text{Let } L'' &= \neg L'. \\ &= \{a^i b^j c^k : i, j, k \geq 0 \text{ and } i = j = k\} \cup \{w \in \{a, b, c\}^* : \text{the letters are out of order}\}. \end{aligned}$$

$$\begin{aligned} \text{Let } L''' &= L'' \cap a^* b^* c^*. \\ &= \{a^n b^n c^n : n \geq 0\}. \end{aligned}$$

L_1 and L_2 are deterministic context-free. Deterministic PDAs that accept L_1 and L_2 can be constructed using the same approach we used to build a deterministic PDA for $L = \{a^m b^n : m \neq n; m, n > 0\}$ in Example 12.7. Their union L' is context-free but it cannot be deterministic context-free. If it were, then its complement L'' would also be deterministic context-free and thus context-free. But it isn't. If it were context-free, then L''' , the intersection of L'' with $a^* b^* c^*$, would also be context-free since the context-free languages are closed under intersection with the regular languages. But L''' is $A^n B^n C^n = \{a^n b^n c^n : n \geq 0\}$, which we have shown is not context-free. ■

Theorem 13.12 Nonclosure under Intersection

Theorem: The deterministic context-free languages are not closed under intersection.

Proof: We show a counterexample:

$$\text{Let } L_1 = \{a^i b^j c^k : i, j, k \geq 0 \text{ and } i = j\}.$$

$$\text{Let } L_2 = \{a^i b^j c^k : i, j, k \geq 0 \text{ and } j = k\}.$$

$$\begin{aligned} \text{Let } L' &= L_1 \cap L_2. \\ &= \{a^n b^n c^n : n \geq 0\}. \end{aligned}$$

L_1 and L_2 are deterministic context-free. The deterministic PDA shown in Figure 13.4 accepts L_1 . A similar one accepts L_2 . But we have shown that their intersection L' is not context-free, much less deterministic context-free.

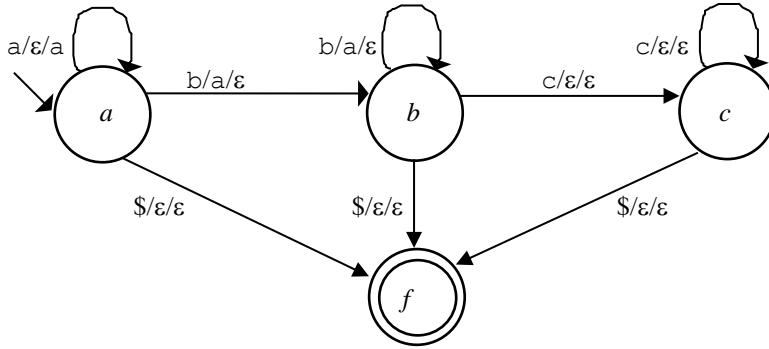


Figure 13.4 A deterministic PDA that accepts $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } i = j\}$

A Hierarchy within the Class of Context-Free Languages

The most important result of this section is the following theorem. There are context-free languages that are not deterministic context-free. Since there are context-free languages for which no deterministic PDA exists, there can exist no equivalent of *ndfsm* for PDAs. Nondeterminism is a fact of life when working with PDAs unless we are willing to work only with languages that have been designed to be deterministic.

The fact that there are context-free languages that are not deterministic poses a problem for the design of efficient parsing algorithms. The best parsing algorithms we have sacrifice either generality (i.e., they cannot correctly parse all context-free languages) or efficiency (i.e., they do not run in time that is linear in the length of the input). In Chapter 15, we will describe some of these algorithms.

Theorem 13.13 Some CFLs are not Deterministic

Theorem: The class of deterministic context-free languages is a *proper* subset of the class of context-free languages. Thus there exist nondeterministic PDAs for which no equivalent deterministic PDA exists.

Proof: By Theorem 13.9, every deterministic context-free language is context-free. So all that remains is to show that there exists at least one context-free language that is not deterministic context-free.

Consider $L = \{a^i b^j c^k : i, j, k \geq 0 \text{ and } (i \neq j) \text{ or } (j \neq k)\}$. L is context-free. The construction of a grammar for it was an exercise in Chapter 11. But we can show that L is not deterministic context-free by the same argument that we used in the proof of Theorem 13.11. If L were deterministic context-free, then, by Theorem 13.10, its complement $L' = \{a^i b^j c^k : i, j, k \geq 0 \text{ and } i = j = k\} \cup \{w \in \{a, b, c\}^* : \text{the letters are out of order}\}$ would also be deterministic context-free and thus context-free. If L' were context-free, then $L'' = L' \cap a^* b^* c^*$ would also be context-free (since the context-free languages are closed under intersection with the regular languages). But $L'' = A^n B^n C^n = \{a^n b^n c^n : n \geq 0\}$, which is not context-free. So L is context-free but not deterministic context-free.

Since L is context-free, it is accepted by some (nondeterministic) PDA M . M is an example of a nondeterministic PDA for which no equivalent deterministic PDA exists. If such a deterministic PDA did exist and accept L , it could be converted into a deterministic PDA that accepted $L\$$. But, if that machine existed, L would be deterministic context-free and we just showed that it is not.

We get the class of deterministic context-free languages when we think about the context-free languages from the perspective of PDAs that accept them. Recall from Section 11.7.3 that, when we think about the context-free languages from the perspective of the grammars that generate them, we also get a subclass of languages that are, in some sense, “easier” than others: there are context-free languages for which unambiguous grammars exist and there are others that are inherently ambiguous, by which we mean that every corresponding grammar is ambiguous.

Example 13.10 Inherent Ambiguity versus Nondeterminism

Recall the language $L_1 = \{a^i b^j c^k : i, j, k \geq 0 \text{ and } (i = j) \text{ or } (j = k)\}$, which can also be described as $\{a^n b^n c^m : n, m \geq 0\} \cup \{a^n b^m c^m : n, m \geq 0\}$. L_1 is inherently ambiguous because every string that is also in $A^n B^n C^n = \{a^n b^n c^n : n \geq 0\}$ is an element of both sublanguages and so has at least two derivations in any grammar for L_1 .

Now consider the slightly different language $L_2 = \{a^n b^n c^m d : n, m \geq 0\} \cup \{a^n b^m c^m e : n, m \geq 0\}$. L_2 is not inherently ambiguous. It is straightforward to write an unambiguous grammar for each of the two sublanguages and any string in L_2 is an element of only one of the sublanguages (since each such string must end in d or e but not both). L_2 is not, however, deterministic. There exists no PDA that can decide which of the two sublanguages a particular string is in until it has consumed the entire string.

What is the relationship between the deterministic context-free languages and the languages that are not inherently ambiguous? The answer is shown in Figure 13.5.

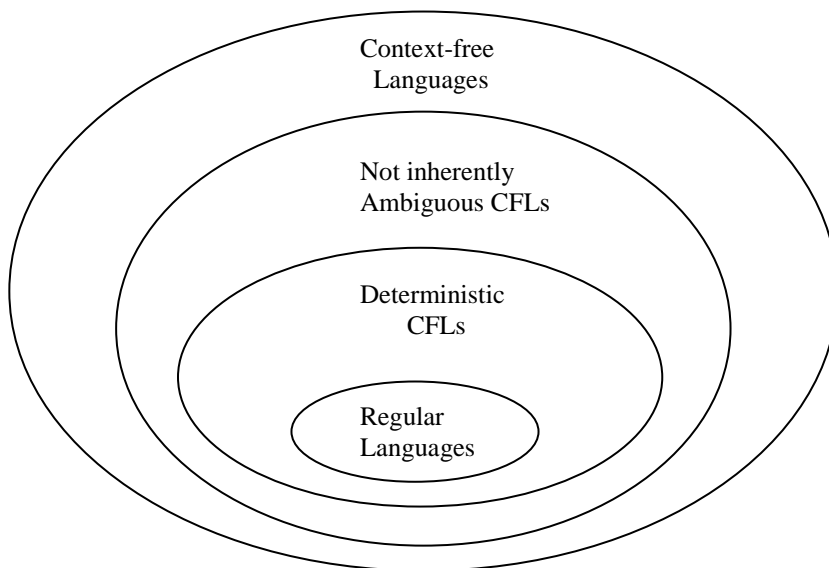


Figure 13.5 A hierarchy within the class of context-free languages

The subset relations shown in the figure are proper:

- There exist deterministic context-free languages that are not regular. These languages are in the innermost donut in the figure. One example is $A^n B^n = \{a^n b^n : n \geq 0\}$.
- There exist languages that are not in the inner donut, i.e., they are not deterministic. But they are context-free and not inherently ambiguous. Two examples of languages in this second donut are:
 - $\text{PalEven} = \{ww^R : w \in \{a,b\}^*\}$. The grammar we showed for it in Example 11.3 is unambiguous.
 - $\{a^n b^n c^m d : n, m \geq 0\} \cup \{a^n b^m c^m e : n, m \geq 0\}$.
- There exist languages that are in the outer donut because they are inherently ambiguous. Two examples are:
 - $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } (i = j) \text{ or } (j = k)\}$.
 - $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } (i \neq j) \text{ or } (j \neq k)\}$.

To prove that the figure is properly drawn requires two additional results:

Theorem 13.14 Every Regular Language is Deterministic Context-Free

Theorem: Every regular language is deterministic context-free.

Proof: The proof is by construction. $\{\$\}$ is regular. So, if L is regular, then so is $L\$$ (since the regular languages are closed under concatenation). So there is a DFSM M that accepts it. Using the construction that we used in the proof of Theorem 13.1 to show that every regular language is context-free, construct, from M a PDA P that accepts $L\$$. P will be deterministic. ■

Theorem 13.15 Every Deterministic CFL has an Unambiguous Grammar

Theorem: For every deterministic context-free language there exists an unambiguous grammar.

Proof: If a language L is deterministic context-free, then there exists a deterministic PDA M that accepts $L\$$. We prove the theorem by construction of an unambiguous grammar G such that $L(M) = L(G)$. We construct G using approximately the same technique that we used to build a grammar from a PDA in the proof of Theorem 12.2. The algorithm *PDAtoCFG* that we presented there proceeded in two steps:

1. Invoke *convertpdatorestricted*(M) to build M' , an equivalent PDA in restricted normal form.
2. Invoke *buildgrammar*(M'), to build an equivalent grammar G .

It is straightforward to show that, if M' is deterministic, then the grammar G that *buildgrammar* constructs will be unambiguous: G produces derivations that mimic the operation of M' . Since M' is deterministic, on any input w it can follow only one path. So G will be able to produce only one leftmost derivation for w . Thus w has only one parse tree. If every string in $L(G)$ has a single parse tree, then G is unambiguous. Since M' accepts $L\$$, G will generate $L\$$. But we can build, from G , a grammar G' that generates L by substituting ϵ for $\$$ in each rule in which $\$$ occurs.

So it remains to show that, from any deterministic PDA M , it is possible to build an equivalent PDA M' that is in restricted normal form and is still deterministic. This can be done using the algorithm *convertPDAtoetnormalform*, which is described in the proof, presented on \mathfrak{B} 635, of Theorem 13.10 (that the deterministic context-free languages are closed under complement). If M is deterministic, then the PDA that is returned by *convertPDAtoetnormalform*(M) will be both deterministic and in restricted normal form.

So the construction that proves the theorem is:

buildunambiggrammar(M : deterministic PDA) =

1. Let $G = \text{buildgrammar}(\text{convertPDAtoetnormalform}(M))$.
 2. Let G' be the result of substituting ϵ for $\$$ in each rule in which $\$$ occurs.
 3. Return G' .
-

13.6 Ogden's Lemma ✦

The context-free Pumping Theorem is a useful tool for showing that a language is not context-free. However, there are many languages that are not context-free but that cannot be proven so just with the Pumping Theorem. In this section we consider a more powerful technique that may be useful in those cases.

Recall that the Pumping Theorem for regular languages imposed the constraint that the pumpable region y had to fall within the first k characters of any “long” string w . We exploited that fact in many of our proofs. But notice that the Pumping Theorem for context-free languages imposes no similar constraint. The two pumpable regions, v and y must be reasonably close together, but, as a group, they can fall anywhere in w . Sometimes there is a region that is pumpable, even though other regions aren't, and this can happen even in the case of long strings drawn from languages that are not context-free.

Example 13.11 Sometimes Pumping Isn't Strong Enough

Let $L = \{a^i b^j c^k : i, j \geq 0, i \neq j\}$. We could attempt to use the context-free Pumping Theorem to show that L is not context-free. Let $w = a^k b^k c^{k+k!}$. (The reason for this choice will be clear soon.) Divide w into three regions, the a 's, the b 's, and the c 's, which we'll call regions 1, 2, and 3, respectively. If either v or y contains two or more distinct symbols, then set q to 2. The resulting string will have letters out of order and thus not be in L . We consider the remaining possibilities:

- (1, 1), (2, 2), (1, 3), (2, 3): set q to 2. The number of a 's will no longer equal the number of b 's, so the resulting string is not in L .
- (1, 2): if $|v| \neq |y|$ then set q to 2. The number of a 's will no longer equal the number of b 's, so the resulting string is not in L . If $|v| = |y|$ then set q to $(k!/|v|) + 1$. Note that $(k!/|v|)$ must be an integer since $|v| \leq k$. The string that results from pumping is $a^X b^X c^{k+k!}$, where $X = k + (q - 1) \cdot |v| = k + (k!/|v|) \cdot |v| = k + k!$. So the number of a 's and of b 's equals the number of c 's. This string is not in L . So far, the proof is going well. But now we must consider:
- (3, 3): pumping in will result in even more c 's than a 's and b 's, so it will produce a string that is still in L . And, while pumping out can reduce the number of c 's, it can't reduce it all the way down to k because $|v \cdot xy| \leq k$. So the maximum number of c 's that can be pumped out is k , which would result in a string with $k!$ c 's. But, as long as $k \geq 3$, $k! > k$. So the resulting string is in L and we have failed to show that L is not context-free.

What we need is a way to prevent v and y from falling in the c region of w .

Ogden's Lemma is a generalization of the Pumping Theorem. It lets us mark some number of symbols in our chosen string w as *distinguished*. Then at least one of v and y must contain at least one distinguished symbol. So, for example, we could complete the proof that we started in Example 13.11 if we could force at least one of v or y to contain at least one a .

Theorem 13.16 Ogden's Lemma

Theorem: If L is a context-free language, then:

$$\exists k \geq 1 (\forall \text{ strings } w \in L, \text{ where } |w| \geq k, \text{ if we mark at least } k \text{ symbols of } w \text{ as distinguished then:} \\ (\exists u, v, x, y, z (w = uvxyz, \\ \begin{array}{l} v \cdot y \text{ contains at least one distinguished symbol,} \\ v \cdot xy \text{ contains at most } k \text{ distinguished symbols, and} \\ \forall q \geq 0 (uv^qxy^qz \text{ is in } L))))).$$

Proof: The proof is analogous to the one we did for the context-free Pumping Theorem except that we consider only paths that generate the distinguished symbols. If L is context-free, then it is generated by some context-free grammar $G = (V, \Sigma, R, S)$ with n nonterminal symbols and branching factor b . Let k be b^{n+1} . Let w be any string in $L(G)$ such that $|w| \geq k$. A parse tree T for w might look like the one shown in Figure 13.6.

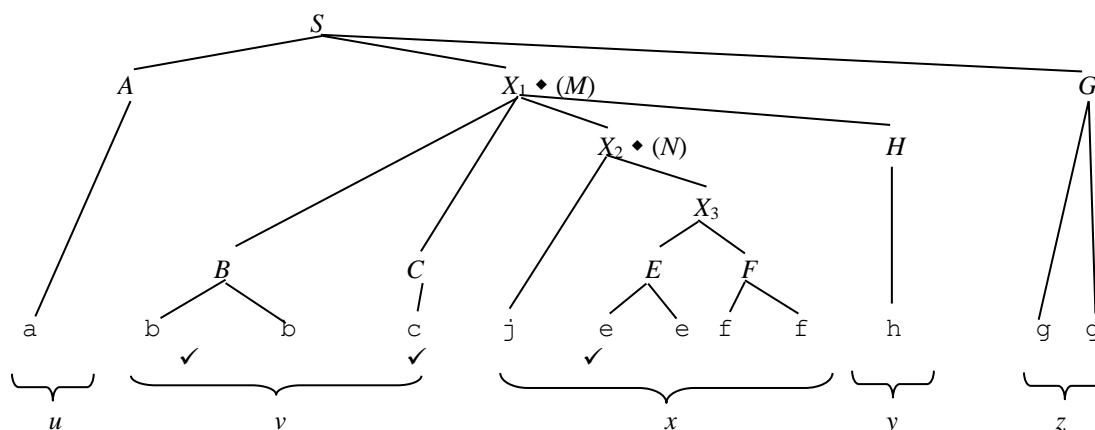


Figure 13.6 A parse tree with some symbols marked as distinguished

Suppose that we mark at least b^{n+1} symbols as distinguished. The distinguished symbols are marked with a \checkmark . (Ignore the fact that there aren't enough of them in the picture. Its only role is to make it easier to visualize the process.) Call the sequence of distinguished nodes the *distinguished subsequence* of w . In this example, that is bje . Note that the distinguished subsequence is not necessarily a substring. The characters in it need not be contiguous. The length of the distinguished subsequence is at least b^{n+1} . We can now mark the nonleaf nodes that branched in a way that enabled the distinguished subsequence to grow to at least length b^{n+1} . Mark every nonleaf node that has at least two daughters that contain a distinguished leaf. In this example, we mark X_2 , and X_1 , as indicated by the symbol \blacklozenge . It is straightforward to prove by induction that T must contain at least one path that contains at least $n + 1$ marked nonleaf nodes since its yield contains b^{n+1} distinguished symbols. Choose one such path such that there is no longer one. That path must contain at least two nodes labeled with the same nonterminal symbol. Choose the two nodes that are labeled with the bottom-most pair of repeated marked nonterminals. Call the lower one N and the higher one M . In the example, M is X_1 and N is X_2 . As shown in the diagram, divide w into $uvxyz$, such that x is the yield of N and vxy is the yield of M . Now observe that:

- vy contains at least one distinguished symbol because the root of the subtree with yield vxy has at least two daughters that contain distinguished symbols. One of them may be in the subtree whose yield is x , but that leaves at least one that must be in either v or y . There may be distinguished symbols in both, although, as in our example T , that is not necessary.
- vxy contains at most $k(b^{n+1})$ distinguished symbols because there are at most $n + 1$ marked internal nodes on a longest path in the subtree that dominates it. Only marked internal nodes create branches that lead to more than one distinguished symbol, and no internal node can create more than b branches.
- $\forall q \geq 0$ (uv^qxy^qz is in L), by the same argument that we used in the proof of the context-free Pumping Theorem. ■

Notice that the context-free Pumping Theorem describes the special case in which all symbols of the string w are marked.

Ogden's Lemma is the tool that we need to complete the proof that we started in Example 13.11:

Example 13.12 Ogden's Lemma May Work When Pumping Doesn't

Now we can use Ogden's Lemma to complete the proof that $L = \{a^i b^j c^k : i, j \geq 0, i \neq j\}$ is not context-free. Let $w = a^k b^k c^{k+k!}$. Mark all the a 's in w as distinguished. If either v or y contains two or more distinct symbols, then set q to 2. The resulting string will have letters out of order and thus not be in L . We consider the remaining possibilities:

- (1, 1), (1, 3): set q to 2. The number of a's will no longer equal the number of b's, so the resulting string is not in L .
- (1, 2): if $|v| \neq |y|$ then set q to 2. The number of a's will no longer equal the number of b's, so the resulting string is not in L . If $|v| = |y|$ then set q to $(k!/|v|) + 1$. Note that $(k!/|v|)$ must be an integer since $|v| \leq k$. The string that results from pumping is $a^{k+(q-1)|v|} b^{k+(q-1)|v|} c^{k+k!} = a^{k+(k!/|v|)|v|} b^{k+(k!/|v|)|v|} c^{k+k!} = a^{k+k!} b^{k+k!} c^{k+k!}$. So the number of a's and of b's equals the number of c's. This string is not in L .
- (2, 2), (2, 3), (3, 3): fails to satisfy the requirement that at least one symbol in vy be marked as distinguished.

There is no way to divide w into vxy such that all the conditions of Ogden's Lemma are met. So L is not context-free.

13.7 Parikh's Theorem ♦

Suppose that we consider a language L not from the point of view of the exact strings it contains but instead by simply counting, for each string w in L , how many instances of each character in Σ w contains. So, from this perspective, the strings $aaabbbba$ and $abababa$ are the same. If Σ is $\{a, b\}$, then both strings can be described with the pair $(4, 3)$ since they contain 4 a's and 3 b's. We can build such descriptions by defining a family of functions ψ_Σ , with domain Σ^* and range $\{(i_1, i_2, \dots, i_k)\}$, where $k = |\Sigma|$:

$$\psi_\Sigma(w) = (i_1, i_2, \dots, i_k) \text{ where, for all } j, i_j = \text{the number of occurrences in } w \text{ of the } j^{\text{th}} \text{ element of } \Sigma.$$

So, if $\Sigma = \{a, b, c, d\}$, then $\psi_\Sigma(aabbbbddd) = (2, 4, 0, 3)$.

Now consider some language L , which is a set of strings over some alphabet Σ . Instead of considering L as a set of strings, we can consider it as the set of vectors that are produced by applying ψ_Σ to the strings it contains. To do this, we define another family of functions Ψ_Σ , with domain $\mathcal{P}(\Sigma^*)$ and range $\mathcal{P}\{(i_1, i_2, \dots, i_k)\}$:

$$\Psi_\Sigma(L) = \{(i_1, i_2, \dots, i_k) : \exists w \in L (\psi_\Sigma(w) = (i_1, i_2, \dots, i_k))\}.$$

If Σ is fixed, then there is a single function ψ and a single function Ψ . In that case, we will omit Σ and refer to the functions just as ψ and Ψ .

We will say that two languages L_1 and L_2 , over the alphabet Σ^* , are **letter-equivalent** iff $\Psi_\Sigma(L_1) = \Psi_\Sigma(L_2)$. In other words, L_1 and L_2 contain the same strings if we disregard the order in which the symbols occur in the strings.

Example 13.13 Letter Equivalence

Let $\Sigma = \{a, b\}$. So, for example, $\psi(a) = (1, 0)$. $\psi(b) = (0, 1)$. $\psi(ab) = (1, 1)$. $\psi(aaabbbb) = (3, 4)$.

Now consider Ψ :

- | | |
|--|---|
| • Let $L_1 = A^n B^n = \{a^n b^n : n \geq 0\}$. | Then $\Psi(L_1) = \{(i, i) : 0 \leq i\}$. |
| • Let $L_2 = (ab)^*$. | Then $\Psi(L_2) = \{(i, i) : 0 \leq i\}$. |
| • Let $L_3 = \{a^n b^n a^n : n \geq 0\}$. | Then $\Psi(L_3) = \{(2i, i) : 0 \leq i\}$. |
| • Let $L_4 = \{a^{2^n} b^n : n \geq 0\}$. | Then $\Psi(L_4) = \{(2i, i) : 0 \leq i\}$. |
| • Let $L_5 = (aba)^*$. | Then $\Psi(L_5) = \{(2i, i) : 0 \leq i\}$. |

L_1 and L_2 are letter-equivalent. So are L_3, L_4 and L_5 .

Just looking at the five languages we considered in Example 13.13, we can observe that it is possible for two languages with different formal properties (for example a regular language and a context-free but not regular one) to be letter equivalent to each other. L_3 is not context-free. L_4 is context-free but not regular. L_5 is regular. But the three of them are letter equivalent to each other.

Parikh's Theorem, which we are about to state formally and then prove, tells us that that example is far from unique. In fact, given any context-free language L , there exists some regular language L' such that L and L' are letter-equivalent to each other. So A^nB^n is letter equivalent to $(ab)^*$. The language $\{a^{2^n}b^n : n \geq 0\}$ is letter equivalent to $(aba)^*$ and to $(aab)^*$. And $\text{PalEven} = \{ww^R : w \in \{a, b\}^*\}$ is letter equivalent to $(aa \cup bb)^*$ since $\Psi(\text{PalEven}) = \Psi((aa \cup bb)^*) = \{(2i, 2j) : 0 \leq i \wedge 0 \leq j\}$. The proof of Parikh's Theorem is similar to the proofs we have already given for the Context-free Pumping Theorem and for Ogden's Lemma. It is based on the fact that, if L is context-free, then all the strings in L can be formed by starting with one of a finite set of "short" strings in L and then pumping in some finite number of strings (v, y pairs), all of which are chosen from a finite library of possible values for v and y .

An interesting application of Parikh's Theorem is in the proof of a corollary that tells us that every context-free language over a single character alphabet must also be regular. We will add that corollary to our kit of tools for proving that a language is not context-free (by showing that, if it were, then it would also be regular but we know that it isn't).

Notice, by the way, that while we are about to prove that if L is context-free then it is letter-equivalent to some regular language, the converse of that claim is false. A language can be letter-equivalent to some regular language and not be context-free. We prove this by considering two of the languages from Example 13.13: $L_3 = \{a^n b^n a^n : n \geq 0\}$ is not context-free, but it is letter-equivalent to $L_5 = (aba)^*$, which is regular.

Theorem 13.17 Parikh's Theorem

Theorem: Every context-free language is letter-equivalent to some regular language.

Proof: The proof follows an argument similar to the one we used to prove the context-free Pumping Theorem. It is given in § 639. ■

An algebraic approach to thinking about what ψ and Ψ are doing is the following: we can describe the standard way of looking at strings as starting with a set S of primitive strings (ϵ and all the one-character strings drawn from Σ) and the single operation of concatenation, which is associative and has ϵ as an identity. Σ^* is then the closure of S under concatenation. ψ_Σ maps elements of Σ^* to elements of $\{(i_1, i_2, \dots, i_k)\}$, on which is defined the operation of pair wise addition, which is associative and has $(0, 0, \dots, 0)$ as an identity. But addition is also commutative, while concatenation is not. So, while, if we concatenate strings, it matters what order we do it in, if we consider the images of strings under ψ , the order in which we combine them doesn't matter. Parikh's Theorem can be described as a special case of more general properties of commutative systems.

When Σ contains just a single character, the order of the characters in a string is irrelevant. So we have the following result:

Theorem 13.18 Every CFL Over a Single-Character Alphabet is Regular

Theorem: Any context-free language over a single-character alphabet is regular.

Proof: By Parikh's Theorem, if L is context-free then L is letter-equivalent to some regular language L' . Since the order of characters has no effect on strings when all characters are the same, $L = L'$. Since L' is regular, so is L . ■

Example 13.14 A^nA^n is Regular

Let $\Sigma = \{a, b\}$ and consider $L = A^nB^n = \{a^n b^n : n \geq 0\}$. A^nB^n is context-free but not regular.

Now let: $\Sigma = \{a\}$ and $L' = \{a^n a^n, n \geq 0\}$.
 $= \{a^{2^n} : n \geq 0\}$.
 $= \{w \in \{a\}^* : |w| \text{ is even}\}$. L' is regular.

Example 13.15 PalEven is Regular if $\Sigma = \{a\}$

Let $\Sigma = \{a, b\}$ and consider $L = \text{PalEven} = \{ww^R : w \in \{a, b\}^*\}$. PalEven is context-free but not regular.

Now let: $\Sigma = \{a\}$ and $L' = \{ww^R : w \in \{a\}^*\}$
 $= \{w \in \{a\}^* : |w| \text{ is even}\}$. L' is regular.

When we are considering only a single letter alphabet, we can use Theorem 13.18 to show that a language that we already know not to be regular cannot be context-free either.

Example 13.16 The Prime Number of a's Language is Not Context-Free

Consider again $\text{Prime}_a = \{a^n : n \text{ is prime}\}$. Prime_a is not context-free. If it were, then, by Theorem 13.18, it would also be regular. But we showed in Example 8.13 that it is not regular. So it is not context-free either.

13.8 Functions on Context-Free Languages

In Section 13.4, we saw that the context-free languages are closed under some important functions, including concatenation, union, and Kleene star. But their closure properties are substantially weaker than are the closure properties of the regular languages. In this section, we consider some other functions that can be applied to languages and we ask whether the context-free languages are closed under them. The proof strategies we will use are the same as the ones we used for the regular languages and for the results we have already obtained for the context-free languages:

- To show that the context-free languages are closed under some function f , we will show an algorithm that constructs, given any context-free language L , either a grammar or a PDA that describes $f(L)$.
- To show that the context-free languages are not closed under some function f , we will exhibit a counterexample, i.e., a language L where L is context-free but $f(L)$ is not.

Example 13.17 Firstchars

Consider again the function $\text{firstchars}(L) = \{w : \exists y \in L (y = cx \wedge c \in \Sigma_L \wedge x \in \Sigma_L^* \wedge w \in c^*)\}$. The context-free languages are closed under $\text{firstchars}(L)$. In fact, if L is context-free then $\text{firstchars}(L)$ is regular. We know that this must be true by an argument similar to the one we used in Example 8.20 to show that the regular languages are closed under firstchars . There must be some finite set of characters $\{c_1, c_2, \dots, c_n\}$ that can begin strings in L (since Σ_L is finite). So there exists some regular expression of the following form that describes $\text{firstchars}(L)$:

$$c_1^* \cup c_2^* \cup \dots \cup c_n^*.$$

We can also show a constructive proof that $\text{firstchars}(L)$ is context-free if L is. If L is a context-free language, then there is some context-free grammar $G = (V, \Sigma, R, S)$ that generates it. We construct a context-free grammar $G' = (V', \Sigma', R', S')$ that generates $\text{firstchars}(L)$:

1. Convert G to Greibach normal form using the procedure *converttoGreibach*, defined in \mathfrak{B} 630.
2. Remove from G all unreachable nonterminals and all rules that mention them.
3. Remove from G all unproductive nonterminals and all rules that mention them.
4. Initialize V' to $\{S'\}$, Σ' to $\{\}$, and R' to $\{\}$.
5. For each remaining rule of the form $S \rightarrow c \gamma$ do:
 - 5.1. Add to R' the rules $S' \rightarrow C_c$, $C_c \rightarrow c C_c$ and $C_c \rightarrow \epsilon$.
 - 5.2. Add to Σ' the symbol c .
 - 5.3. Add to V' the symbol C_c .
6. Return G' .

The idea behind this construction is that, if G is in Greibach normal form, then, each time a rule is applied, the next terminal symbol is generated. So, if we look at G 's start symbol S and ask what terminals any of its rules can generate, we'll know exactly what terminals strings in $L(G)$ can start with.

Example 13.18 Maxstring

Consider again the function $maxstring(L) = \{w : w \in L \text{ and } \forall z \in \Sigma^* (z \neq \varepsilon \rightarrow wz \notin L)\}$. The context-free languages are not closed under $maxstring(L)$. The proof is by counterexample. Consider the language $L = \{a^i b^j c^k : k \leq i \text{ or } k \leq j\}$. L is context-free but $maxstring(L)$ is not. We leave the proof of this as an exercise.

13.9 Exercises

- 1) For each of the following languages L , state whether L is regular, context-free but not regular, or not context-free and prove your answer.
 - a) $\{xy : x, y \in \{a, b\}^* \text{ and } |x| = |y|\}$.
 - b) $\{(ab)^n a^n b^n : n > 0\}$.
 - c) $\{x\#y : x, y \in \{0, 1\}^* \text{ and } x \neq y\}$.
 - d) $\{a^i b^n : i, n > 0 \text{ and } i = n \text{ or } i = 2n\}$.
 - e) $\{wx : |w| = 2 \cdot |x| \text{ and } w \in a^+ b^+ \text{ and } x \in a^+ b^+\}$.
 - f) $\{a^n b^m c^k : m \leq \min(n, k)\}$.
 - g) $\{xyx^R : x \in \{0, 1\}^+ \text{ and } y \in \{0, 1\}^*\}$.
 - h) $\{xwx^R : x, w \in \{a, b\}^+ \text{ and } |x| = |w|\}$.
 - i) $\{ww^R w : w \in \{a, b\}^*\}$.
 - j) $\{xwx : |w| = 2 \cdot |x| \text{ and } w \in \{a, b\}^* \text{ and } x \in \{c\}^*\}$.
 - k) $\{a^i : i \geq 0\} \{b^i : i \geq 0\} \{a^i : i \geq 0\}$.
 - l) $\{x \in \{a, b\}^* : |x| \text{ is even and the first half of } x \text{ has one more } a \text{ than does the second half}\}$.
 - m) $\{w \in \{a, b\}^* : \#_a(w) = \#_b(w) \text{ and } w \text{ does not contain either the substring } aaa \text{ or } abab\}$.
 - n) $\{a^m b^{2n} c^m\} \cap \{a^m b^m c^{2m}\}$.
 - o) $\{x \subset y : x, y \in \{0, 1\}^* \text{ and } y \text{ is a prefix of } x\}$.
 - p) $\{w : w = uu^R \text{ or } w = ua^n : n = |u|, u \in \{a, b\}^*\}$.
 - q) $L(G)$, where $G = S \rightarrow aSa$
 $S \rightarrow SS$
 $S \rightarrow \varepsilon$
 - r) $\{w \in (A-Z, a-z, \dots, \text{blank})^+ : \text{there exists at least one duplicated, capitalized word in } w\}$. For example, the sentence, "The history of China can be viewed from the perspective of an outsider or of someone living in China," $\in L$.
 - s) $\neg L_0$, where $L_0 = \{ww : w \in \{a, b\}^*\}$.
 - t) L^* , where $L = \{0^* 1^i 0^* 1^i 0^*\}$.
 - u) $\neg A^n B^n$.
 - v) $\{ba^j b : j = n^2 \text{ for some } n \geq 0\}$. For example, $baaaab \in L$.
 - w) $\{w \in \{a, b, c, d\}^* : \#_b(w) \geq \#_c(w) \geq \#_d(w) \geq 0\}$.
- 2) Let $L = \{w \in \{a, b\}^* : \text{the first, middle, and last characters of } w \text{ are identical}\}$.
 - a) Show a context-free grammar for L .
 - b) Show a natural PDA that accepts L .
 - c) Prove that L is not regular.
- 3) Let $L = \{a^n b^m c^n d^m : n, m \geq 1\}$. L is interesting because of its similarity to a useful fragment of a typical programming language in which one must declare procedures before they can be invoked. The procedure declarations include a list of the formal parameters. So now imagine that the characters in a^n correspond to the formal parameter list in the declaration of procedure 1. The characters in b^m correspond to the formal parameter list in the declaration of procedure 2. Then the characters in c^n and d^m correspond to the parameter lists in an

invocation of procedure 1 and procedure 2 respectively, with the requirement that the number of parameters in the invocations match the number of parameters in the declarations. Show that L is not context-free.

- 4) Without using the Pumping Theorem, prove that $L = \{w \in \{a, b, c\}^* : \#_a(w) = \#_b(w) = \#_c(w) \text{ and } \#_a(w) > 50\}$ is not context-free.
- 5) Give an example of a context-free language $L (\neq \Sigma^*)$ that contains a subset L_1 that is not context-free. Prove that L is context free. Describe L_1 and prove that it is not context-free.
- 6) Let $L_1 = L_2 \cap L_3$.
 - a) Show values for $L_1, L_2,$ and $L_3,$ such that L_1 is context-free but neither L_2 nor L_3 is.
 - b) Show values for $L_1, L_2,$ and $L_3,$ such that L_2 is context-free but neither L_1 nor L_3 is.
- 7) Give an example of a context-free language $L,$ other than one of the ones in the book, where $\neg L$ is not context-free.
- 8) Theorem 13.7 tells us that the context-free languages are closed under intersection with the regular languages. Prove that the context-free languages are also closed under union with the regular languages.
- 9) Complete the proof that the context-free languages are not closed under *maxstring* by showing that $L = \{a^i b^j c^k : k \leq i \text{ or } k \leq j\}$ is context-free but *maxstring*(L) is not context-free.
- 10) Use the Pumping Theorem to complete the proof, started in \mathbb{C} 748, that English is not context-free if we make the assumption that subjects and verbs must match in a “respectively” construction.
- 11) In \mathbb{C} 778, we give an example of a simple musical structure that cannot be described with a context-free grammar. Describe another one, based on some musical genre with which you are familiar. Define a sublanguage that captures exactly that phenomenon. In other words, ignore everything else about the music you are considering and describe a set of strings that meets the one requirement you are studying. Prove that your language is not context-free.
- 12) Define the leftmost maximal P subsequence m of a string w as follows:
 - P must be a nonempty set of characters.
 - A string s is a P subsequence of w iff s is a substring of w and s is composed entirely of characters in P . For example 1, 0, 10, 01, 11, 011, 101, 111, 1111, and 1011 are $\{0, 1\}$ subsequences of 2312101121111.
 - Let S be the set of all P subsequences of w such that, for each element t of $S,$ there is no P subsequence of w longer than t . In the example above, $S = \{1111, 1011\}$.
 - Then m is the leftmost (within w) element of S . In the example above, $m = 1011$.
 - a) Let $L = \{w \in \{0-9\}^* : \text{if } y \text{ is the leftmost maximal } \{0, 1\} \text{ subsequence of } w \text{ then } |y| \text{ is even}\}$. Is L regular, context free (but not regular) or neither? Prove your answer.
 - b) Let $L = \{w \in \{a, b, c\}^* : \text{the leftmost maximal } \{a, b\} \text{ subsequence of } w \text{ starts with } a\}$. Is L regular (but not context free), context free or neither? Prove your answer.
- 13) Are the context-free languages closed under each of the following functions? Prove your answer.
 - a) $\text{chop}(L) = \{w : \exists x \in L (x = x_1 c x_2 \wedge x_1 \in \Sigma_L^* \wedge x_2 \in \Sigma_L^* \wedge c \in \Sigma_L \wedge |x_1| = |x_2| \wedge w = x_1 x_2)\}$.
 - b) $\text{mix}(L) = \{w : x, y, z : (x \in L, x = yz, |y| = |z|, w = yz^R)\}$.
 - c) $\text{pref}(L) = \{w : \exists x \in \Sigma^* (wx \in L)\}$.
 - d) $\text{middle}(L) = \{x : \exists y, z \in \Sigma^* (yxz \in L)\}$.
 - e) letter substitution.
 - f) $\text{shuffle}(L) = \{w : \exists x \in L (w \text{ is some permutation of } x)\}$.
 - g) $\text{copyreverse}(L) = \{w : \exists x \in L (w = xx^R)\}$.

- 14) Let $alt(L) = \{x: \exists y, n (y \in L, |y| = n, n > 0, y = a_1 \dots a_n, \forall i \leq n (a_i \in \Sigma), \text{ and } x = a_1 a_3 a_5 \dots a_k, \text{ where } k = (\text{if } n \text{ is even then } n-1 \text{ else } n))\}$.
- Consider $L = a^n b^n$. Clearly describe $L_1 = alt(L)$.
 - Are the context free languages closed under the function alt ? Prove your answer.
- 15) Let $L_1 = \{a^n b^m : n \geq m\}$. Let $R_1 = \{(a \cup b)^* : \text{there is an odd number of } a\text{'s and an even number of } b\text{'s}\}$. Use the construction that is described in the proof of Theorem 13.7 to build a PDA that accepts $L_1 \cap R_1$.
- 16) Let T be a set of languages defined as follows:
- $$T = \{L : L \text{ is a context-free language over the alphabet } \{a, b, c\} \text{ and, if } x \in L, \text{ then } |x| \equiv_3 0\}.$$
- Let P be the following function on languages:
- $$P(L) = \{w : \exists x \in \{a, b, c\}^* \text{ and } \exists y \in L \text{ and } y = xw\}.$$
- Is the set T closed under P ? Prove your answer.
- 17) Show that the following languages are deterministic context-free:
- $\{w : w \in \{a, b\}^* \text{ and each prefix of } w \text{ has at least as many } a\text{'s as } b\text{'s}\}$.
 - $\{a^n b^n, n \geq 0\} \cup \{a^n c^n, n \geq 0\}$.
- 18) Show that $L = \{a^n b^n, n \geq 0\} \cup \{a^n b^{2n}, n \geq 0\}$ is not deterministic context-free.
- 19) Are the deterministic context-free languages closed under reverse? Prove your answer.
- 20) Prove that each of the following languages is not context-free. (Hint: use Ogden's Lemma.)
- $\{a^i b^j c^k : i \geq 0, j \geq 0, k \geq 0, \text{ and } i \neq j \neq k\}$.
 - $\{a^i b^j c^k d^n : i \geq 0, j \geq 0, k \geq 0, n \geq 0, \text{ and } (i = 0 \text{ or } j = k = n)\}$.
- 21) Let $\Psi(L)$ be as defined in Section 13.7, in our discussion of Parikh's Theorem. For each of the following languages L , first state what $\Psi(L)$ is. Then give a regular language that is letter-equivalent to L .
- $Bal = \{w \in \{(), ()^*\} : \text{the parentheses are balanced}\}$.
 - $Pal = \{w \in \{a, b\}^* : w \text{ is a palindrome}\}$.
 - $\{x^R \# y : x, y \in \{0, 1\}^* \text{ and } x \text{ is a substring of } y\}$.
- 22) For each of the following claims, state whether it is *True* or *False*. Prove your answer.
- If L_1 and L_2 are two context-free languages, $L_1 - L_2$ must also be context-free.
 - If L_1 and L_2 are two context-free languages and $L_1 = L_2 L_3$, then L_3 must also be context-free.
 - If L is context free and R is regular, $R - L$ must be context-free.
 - If L_1 and L_2 are context-free languages and $L_1 \subseteq L \subseteq L_2$, then L must be context-free.
 - If L_1 is a context-free language and $L_2 \subseteq L_1$, then L_2 must be context-free.
 - If L_1 is a context-free language and $L_2 \subseteq L_1$, it is possible that L_2 is regular.
 - A context-free grammar in Chomsky normal form is always unambiguous.

14 Algorithms and Decision Procedures for Context-Free Languages

Many questions that we could answer when asked about regular languages are unanswerable for context-free ones. But a few important questions can be answered and we have already presented a useful collection of algorithms that can operate on context-free grammars and PDAs. We'll present a few more here.

14.1 The Decidable Questions

Fortunately, the most important questions (i.e., the ones that must be answerable if context-free grammars are to be of any practical use) are decidable.

14.1.1 Membership

We begin with the most fundamental question, "Given a language L and a string w , is w in L ?" Fortunately this question can be answered for every context-free language. By Theorem 12.1, for every context-free language L , there exists a PDA M such that M accepts L . But we must be careful. As we showed in Section 12.4, PDAs are not guaranteed to halt. So the mere existence of a PDA that accepts L does not guarantee the existence of a procedure that decides it (i.e., always halts and says yes or no appropriately).

It turns out that there are two alternative approaches to solving this problem, both of which work:

- Use a grammar: using facts about every derivation that is produced by a grammar in Chomsky normal form, we can construct an algorithm that explores a finite number of derivation paths and finds one that derives a particular string w iff such a path exists.
- Use a PDA: while not all PDAs halt, it is possible, for any context-free language L , to craft a PDA M that is guaranteed to halt on all inputs and that accepts all strings in L and rejects all strings that are not in L .

Using a Grammar to Decide

We begin by considering the first alternative. We show a straightforward, algorithm for deciding whether a string w is in a language L :

decideCFLusingGrammar(L : CFL, w : string) =

1. If L is specified as a PDA, use *PDAtoCFG*, presented in the proof of Theorem 12.2, to construct a grammar G such that $L(G) = L(M)$.
2. If L is specified as a grammar G , simply use G .
3. If $w = \epsilon$ then if S_G is nullable (as defined in the description of *removeEps* in Section 11.7.4) then accept, otherwise reject.
4. If $w \neq \epsilon$ then:
 - 4.1. From G , construct G' such that $L(G') = L(G) - \{\epsilon\}$ and G' is in Chomsky normal form.
 - 4.2. If G derives w , it does so in $2 \cdot |w| - 1$ steps. Try all derivations in G of that number of steps. If one of them derives w , accept. Otherwise reject.

The running time of *decideCFLusingGrammar* can be analyzed as follows: We assume that the time required to build G' is constant, since it does not depend on w . Let $n = |w|$. Let g be the search-branching factor of G' , defined to be the maximum number of rules that share a left-hand side. Then the number of derivations of length $2n-1$ is bounded by g^{2n-1} , and it takes at most $2n-1$ steps to check each one. So the worst-case running time of *decideCFLusingGrammar* is $\mathcal{O}(ng^{2n-1})$. In Section 15.3.1, we will present techniques that are substantially more efficient. We will describe the CKY algorithm, which, given a grammar G in Chomsky normal form, decides the membership question for G in time that is $\mathcal{O}(n^3)$. We will then describe an algorithm that can decide the question in time that is linear in n if the grammar that is provided meets certain requirements.

Theorem 14.1 Decidability of Context-Free Languages

Theorem: Given a context-free language L (represented as either a context-free grammar or a PDA) and a string w , there exists a decision procedure that answers the question, “Is $w \in L$?”

Proof: The following algorithm, *decideCFL*, uses *decideCFLusingGrammar* to answer the question:

decideCFL(L : CFL, w : string) =

1. If *decideCFLusingGrammar*(L , w) accepts, return *True* else return *False*. ■

Using a PDA to Decide ✪

It is also possible to solve the membership problem using PDAs. We take a two-step approach. We first show that, for every context-free language L , it is possible to build a PDA that accepts $L - \{\varepsilon\}$ and that has no ε -transitions. Then we show that every PDA with no ε -transitions is guaranteed to halt.

Theorem 14.2 Elimination of ε -Transitions

Theorem: Given any context-free grammar $G = (V, \Sigma, R, S)$, there exists a PDA M such that $L(M) = L(G) - \{\varepsilon\}$ and M contains no transitions of the form $((q_1, \varepsilon, \alpha), (q_2, \beta))$. In other words, every transition reads exactly one input character.

Proof: The proof is by a construction that begins by converting G to Greibach normal form. Recall that, in any grammar in Greibach normal form, all rules are of the form $X \rightarrow aA$, where $a \in \Sigma$ and $A \in (V - \Sigma)^*$. Now consider again the algorithm *cfgtoPDAtopdown*, which builds, from any context-free grammar G , a PDA M that, on input w , simulates G deriving w , starting from S . $M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$, where Δ contains:

1. The start-up transition $((p, \varepsilon, \varepsilon), (q, S))$, which pushes the start symbol onto the stack and goes to state q .
2. For each rule $X \rightarrow s_1s_2\dots s_n$ in R , the transition $((q, \varepsilon, X), (q, s_1s_2\dots s_n))$, which replaces X by $s_1s_2\dots s_n$. If $n = 0$ (i.e., the right-hand side of the rule is ε), then the transition $((q, \varepsilon, X), (q, \varepsilon))$.
3. For each character $c \in \Sigma$, the transition $((q, c, c), (q, \varepsilon))$, which compares an expected character from the stack against the next input character and continues if they match.

The start-up transition, plus all the transitions generated in step 2, are ε -transitions. But now suppose that G is in Greibach normal form. If G contains the rule $X \rightarrow cs_2\dots s_n$ (where $c \in \Sigma$ and s_2 through s_n are elements of $V - \Sigma$), it is not necessary to push c onto the stack, only to pop it with a rule from step 3. Instead, we collapse the push and the pop into a single transition. So we create a transition that can be taken only if the next input character is c . In that case, the string $s_2\dots s_n$ is pushed onto the stack.

Now we need only find a way to get rid of the start-up transition, whose job is to push S onto the stack so that the derivation process can begin. Since G is in Greibach normal form, any rules with S on the left-hand side must have the form $S \rightarrow cs_2\dots s_n$. So instead of reading no input and just pushing S , M will skip pushing S and instead, if the first input character is c , read it and push the string $s_2\dots s_n$.

Since terminal symbols are no longer pushed onto the stack, we no longer need the transitions created in step 3 of the original algorithm.

So $M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$, where Δ contains:

1. The start-up transitions: For each rule $S \rightarrow cs_2\dots s_n$, the transition $((p, c, \varepsilon), (q, s_2\dots s_n))$.
2. For each rule $X \rightarrow cs_2\dots s_n$ (where $c \in \Sigma$ and s_2 through s_n are elements of $V - \Sigma$), the transition $((q, c, X), (q, s_2\dots s_n))$.
3. For each character $c \in \Sigma$, the transition $((q, c, c), (q, \varepsilon))$, which compares an expected character from the stack against the next input character and continues if they match.

The start-up transition, plus all the transitions generated in step 2, are ϵ -transitions. But now suppose that G is in Greibach normal form. If G contains the rule $X \rightarrow cs_2 \dots s_n$ (where $c \in \Sigma$ and s_2 through s_n are elements of $V - \Sigma$), it is not necessary to push c onto the stack, only to pop it with a rule from step 3. Instead, we collapse the push and the pop into a single transition. So we create a transition that can be taken only if the next input character is c . In that case, the string $s_2 \dots s_n$ is pushed onto the stack.

Now we need only find a way to get rid of the start-up transition, whose job is to push S onto the stack so that the derivation process can begin. Since G is in Greibach normal form, any rules with S on the left-hand side must have the form $S \rightarrow cs_2 \dots s_n$. So instead of reading no input and just pushing S , M will skip pushing S and instead, if the first input character is c , read it and push the string $s_2 \dots s_n$.

Since terminal symbols are no longer pushed onto the stack, we no longer need the transitions created in step 3 of the original algorithm.

So $M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$, where Δ contains:

1. The start-up transitions: For each rule $S \rightarrow cs_2 \dots s_n$, the transition $((p, c, \epsilon), (q, s_2 \dots s_n))$.
2. For each rule $X \rightarrow cs_2 \dots s_n$ (where $c \in \Sigma$ and s_2 through s_n are elements of $V - \Sigma$), the transition $((q, c, X), (q, s_2 \dots s_n))$.

The following algorithm builds the required PDA:

*cfgtoPDA*noeps(G : context-free grammar) =

1. Convert G to Greibach normal form, producing G' .
2. From G' build the PDA M described above.

■

Theorem 14.3 Halting Behavior of PDAs without ϵ -Transitions

Theorem: Let M be a PDA that contains no transitions of the form $((q_1, \epsilon, s_1), (q_2, s_2))$, i.e., no ϵ -transitions. Consider the operation of M on input $w \in \Sigma^*$. M must halt and either accept or reject w . Let $n = |w|$. We make three additional claims:

- a) Each individual computation of M must halt within n steps.
- b) The total number of computations pursued by M must be less than or equal to b^n , where b is the maximum number of competing transitions from any state in M .
- c) The total number of steps that will be executed by all computations of M is bounded by nb^n .

Proof:

- a) Since each computation of M must consume one character of w at each step and M will halt when it runs out of input, each computation must halt within n steps.
- b) M may split into at most b branches at each step in a computation. The number of steps in a computation is less than or equal to n . So the total number of computations must be less than or equal to b^n .
- c) Since the maximum number of computations is b^n and the maximum length of each is n , the maximum number of steps that can be executed before all computations of M halt is nb^n .

■

So a second way to answer the question, “Given a context-free language L and a string w , is w in L ?” is to execute the following algorithm:

*decideCFL*usingPDA(L : CFL, w : string) =

1. If L is specified as a PDA, use *PDAtoCFG*, as presented in the proof of Theorem 12.2, to construct a grammar G such that $L(G) = L(M)$.
2. If L is specified as a grammar G , simply use G .
3. If $w = \epsilon$ then if S_G is nullable (as defined in the description of *removeEps* in Section 11.7.4) then accept, otherwise reject.
4. If $w \neq \epsilon$ then:
 - 4.1. From G , construct G' such that $L(G') = L(G) - \{\epsilon\}$ and G' is in Greibach normal form.

- 4.2. From G' construct, using *cfgtoPDAnoeps*, the algorithm described in the proof of Theorem 14.2, a PDA M' such that $L(M') = L(G')$ and M' has no ϵ -transitions.
- 4.3. By Theorem 14.3, all paths of M' are guaranteed to halt within a finite number of steps. So run M' on w . Accept if M' accepts and reject otherwise.

The running time of *decideCFLusingPDA* can be analyzed as follows: We will take as a constant the time required to build M' , since that can be done once. It need not be repeated for each string that is to be analyzed. Given M' , the time required to analyze a string w is then the time required to simulate all paths of M' on w . Let $n = |w|$. From Theorem 14.3, we know that the total number of steps that will be executed by all paths of M is bounded by nb^n , where b is the maximum number of competing transitions from any state in M' . But is that number of steps required? If one state has a large number of competing transitions but the others do not, then the average branching factor will be less than b , so fewer steps will be necessary. But if b is greater than 1, the number of steps still grows exponentially with n . The exact number of steps also depends on how the simulation is done. A straightforward depth-first search of the tree of possibilities will explore b^n steps, which is less than nb^n because it does not start each path over at the beginning. But it still requires time that is $\mathcal{O}(b^n)$. In Section 15.2.3, we present an alternative approach to top-down parsing that runs in time that is linear in n if the grammar that is provided meets certain requirements.

14.1.2 Emptiness and Finiteness

While many interesting questions are not decidable for context-free languages, two others, in addition to membership are: emptiness and finiteness.

Theorem 14.4 Decidability of Emptiness and Finiteness

Theorem: Given a context-free language L , there exists a decision procedure that answers each of the following questions:

1. Given a context-free language L , is $L = \emptyset$?
2. Given a context-free language L , is L infinite?

Since we have proven that there exists a grammar that generates L iff there exists a PDA that accepts it, these questions will have the same answers whether we ask them about grammars or about PDAs.

Proof:

- (1) Let $G = (V, \Sigma, R, S)$ be a context-free grammar that generates L . $L(G) = \emptyset$ iff S is unproductive (i.e., not able to generate any terminal strings). The following algorithm exploits the procedure *removeunproductive*, defined in Section 11.4, to remove all unproductive nonterminals from G . It answers the question, “Given a context-free language L , is $L = \emptyset$?”.

decideCFLempty(G : context-free grammar) =

1. Let $G' = \text{removeunproductive}(G)$.
 2. If S is not present in G' then return *True* else return *False*.
- (2) Let $G = (V, \Sigma, R, S)$ be a context-free grammar that generates L . We use an argument similar to the one that we used to prove the context-free Pumping Theorem. Let n be the number of nonterminals in G . Let b be the branching factor of G . The longest string that G can generate without creating a parse tree with repeated nonterminals along some path is of length b^n . If G generates no strings of length greater than b^n , then $L(G)$ is finite. If G generates even one string w of length greater than b^n , then, by the same argument we used to prove the Pumping Theorem, it generates an infinite number of strings since $w = uvxyz$, $|vy| > 0$, and $\forall q \geq 0$ (uv^qxy^qz is in L). So we could try to test to see whether L is infinite by invoking *decideCFL*(L, w) on all strings in Σ^* of length greater than b^n . If it returns *True* for any such string, then L is infinite. If it returns *False* on all such strings, then L is finite.

But, assuming Σ is not empty, there is an infinite number of such strings. Fortunately, it is necessary to try only a finite number of them. Suppose that G generates even one string of length greater than $b^{n+1} + b^n$. Let t be the shortest such string. By the Pumping Theorem, $t = uvxyz$, $|vy| > 0$, and uxz (the result of pumping vy out once) \in

L . Note that $|uxz| < |t|$ since some nonempty vy was pumped out of t to create it. Since, by assumption, t is the shortest string in L of length greater than $b^{n+1} + b^n$, $|uxz|$ must be less than or equal to $b^{n+1} + b^n$. But the Pumping Theorem also tells us that $|vxy| \leq k$ (i.e., b^{n+1}), so no more than b^{n+1} strings could have been pumped out of t . Thus we have that $b^n < |uxz| \leq b^{n+1} + b^n$. So, if L contains any strings of length greater than b^n , it must contain at least one string of length less than or equal to $b^{n+1} + b^n$. We can now define *decideCFLinfinite* to answer the question, “Given a context-free language L , is L infinite?”:

decideCFLinfinite(G : context-free grammar) =

1. Lexicographically enumerate all strings in Σ^* of length greater than b^n and less than or equal to $b^{n+1} + b^n$.
2. If, for any such string w , *decideCFL*(L , w) returns *True* then return *True*. L is infinite.
3. If, for all such strings w , *decideCFL*(L , w) returns *False* then return *False*. L is not infinite. ■

14.1.3 Equality of Deterministic Context-Free languages

Theorem 14.5 Decidability of Equivalence for Deterministic Context-Free Languages

Theorem: Given two *deterministic* context-free languages L_1 and L_2 , there exists a decision procedure to determine whether $L_1 = L_2$.

Proof: This claim was not proved until 1997 and the proof [Sénizergues 2001] is beyond the scope of this book, but see [1]. ■

14.2 The Undecidable Questions

Unfortunately, we will prove in Chapter 18 that there exists no decision procedure for many other questions that we might like to be able to ask about context-free languages, including:

- Given a context-free language L , is $L = \Sigma^*$?
- Given a context-free language L , is the complement of L context-free?
- Given a context-free language L , is L regular?
- Given two context-free languages L_1 and L_2 , is $L_1 = L_2$? (Theorem 14.5 tells us that this question is decidable for the restricted case of two deterministic context-free languages. But it is undecidable in the more general case.)
- Given two context-free languages L_1 and L_2 , is $L_1 \subseteq L_2$?
- Given two context-free languages L_1 and L_2 , is $L_1 \cap L_2 = \emptyset$?
- Given a context-free language L , is L inherently ambiguous?
- Given a context-free grammar G , is G ambiguous?

14.3 Summary of Algorithms and Decision Procedures for Context-Free Languages

Although we have presented fewer algorithms and decision procedures for context-free languages than we did for regular languages, there are many important ones, which we summarize here:

- Algorithms that transform grammars:
 - *removeunproductive*(G : context-free grammar): Construct a grammar G' that contains no unproductive nonterminals and such that $L(G') = L(G)$.
 - *removeunreachable*(G : context-free grammar): Construct a grammar G' that contains no unreachable nonterminals and such that $L(G') = L(G)$.
 - *removeEps*(G : context-free grammar): Construct a grammar G' that contains no rules of the form $X \rightarrow \varepsilon$ and such that $L(G') = L(G) - \{\varepsilon\}$.

- *atmostoneEps*(G : context-free grammar): Construct a grammar G' that contains no rules of the form $X \rightarrow \varepsilon$ except possibly $S^* \rightarrow \varepsilon$, in which case there are no rules whose right-hand side contains S^* , and such that $L(G') = L(G)$.
- *converttoChomsky*(G : context-free grammar): Construct a grammar G' in Chomsky normal form, where $L(G') = L(G) - \{\varepsilon\}$.
- *removeUnits*(G : context-free grammar): Construct a grammar G' that contains no unit productions, where $L(G') = L(G) - \{\varepsilon\}$.
- Algorithms that convert between context-free grammars and PDAs:
 - *cfgtoPDAtopdown*(G : context-free grammar): Construct a PDA M such that $L(M) = L(G)$ and M operates top-down to simulate a left-most derivation in G .
 - *cfgtoPDAbottomup*(G : context-free grammar): Construct a PDA M such that $L(M) = L(G)$ and M operates bottom up to simulate, backwards, a right-most derivation in G .
 - *cfgtoPDAnoeps*(G : context-free grammar): Construct a PDA M such that M contains no transitions of the form $((q_1, \varepsilon, s_1), (q_2, s_2))$ and $L(M) = L(G) - \{\varepsilon\}$.
- Algorithms that transform PDAs:
 - *convertpdatorestricted*(M : PDA): Construct a PDA M' in restricted normal form where $L(M') = L(M)$.
- Algorithms that compute functions of languages defined as context-free grammars:
 - Given two grammars G_1 and G_2 , construct a new grammar G_3 such that $L(G_3) = L(G_1) \cup L(G_2)$.
 - Given two grammars G_1 and G_2 , construct a new grammar G_3 such that $L(G_3) = L(G_1) L(G_2)$.
 - Given a grammar G , construct a new grammar G' such that $L(G') = (L(G))^*$.
 - Given a grammar G , construct a new grammar G' such that $L(G') = (L(G))^R$.
 - Given a grammar G , construct a new grammar G' that accepts *letsub*($L(G)$), where *letsub* is a letter substitution function.
- Miscellaneous algorithms for PDAs:
 - *intersectPDAandFSM*(M_1 : PDA, M_2 : FSM): Construct a PDA M_3 such that $L(M_3) = L(M_1) \cap L(M_2)$.
 - *without\$(M: PDA): If M accepts $L\$$, construct a PDA M' such that $L(M') = L$.*
 - *complementdetPDA*(M : DPDA): If M accepts $L\$$, construct a PDA M' such that $L(M') = (\neg L)\$$.
- Decision procedures that answer questions about context-free languages:
 - *decideCFLusingPDA*(L : CFL, w : string): Decide whether w is in L .
 - *decideCFLusingGrammar*(L : CFL, w : string): Decide whether w is in L .
 - *decideCFL*(L : CFL, w : string): Decide whether w is in L .
 - *decideCFLempty*(G : context-free grammar, w : string): Decide whether $L(G) = \emptyset$.
 - *decideCFLinfinite*(G : context-free grammar, w : string): Decide whether $L(G)$ is infinite.

14.4 Exercises

- 1) Give a decision procedure to answer each of the following questions:
 - a) Given a regular expression α and a PDA M , is the language accepted by M a subset of the language generated by α ?
 - b) Given a context-free grammar G and two strings s_1 and s_2 , does G generate $s_1 s_2$?
 - c) Given a context-free grammar G , does G generate at least three strings?
 - d) Given a context-free grammar G , does G generate any even length strings?
 - e) Given a regular grammar G , is $L(G)$ context-free?

15 Context-Free Parsing ♦

Programming languages are (mostly) context-free. Query languages are usually context-free. English can, in large part, be considered context-free. Strings in these languages need to be analyzed and interpreted by compilers, query engines, and various other kinds of application programs. So we need an algorithm that can, given a context-free grammar G :

1. Examine a string and decide whether or not it is a syntactically well-formed member of $L(G)$, and
2. If it is, assign to it a parse tree that describes its structure and thus can be used as the basis for further interpretation.

Are programming languages really context-free? ☹ 666.

In Section 14.1.1, we described two techniques that can be used to construct, from a grammar G , a decision procedure that answers the question, “Given a string w , is w in $L(G)$?” But we aren’t done. We must still deal with the following issues:

- The first procedure, *decideCFLusingGrammar*, requires a grammar that is in Chomsky normal form. The second procedure, *decideCFLusingPDA*, requires a grammar that is in Greibach normal form. We would like to use a natural grammar so that the parsing process can produce a natural parse tree.
- Both procedures require search and take time that grows exponentially in the length of the input string. But we need efficient parsers, preferably ones that run in time that is linear in the length of the input string.
- All either procedure does is to determine membership in $L(G)$. It does not produce parse trees.

Query languages are context-free. ☹ 805.

In this chapter we will sketch solutions to all of these problems. The discussion will be organized as follows:

- Easy issues:
 - Actually building parse trees: All of the parsers we will discuss work by applying grammar rules. So, to build a parse tree, it suffices to augment the parser with a function that builds a chunk of tree every time a rule is applied.
 - Using lookahead to reduce nondeterminism: It is often possible to reduce (or even eliminate) nondeterminism by allowing the parser to look ahead at the next one or more input symbols before it makes a decision about what to do.
- Lexical analysis: a preprocessing step in which strings of individual input characters are divided into strings of larger units, called tokens, that can be input to a parser.
- Top-down parsers:
 - A simple but inefficient recursive descent parser.
 - Modifying a grammar for top-down parsing.
 - LL parsing.
- Bottom-up parsers:
 - The simple but not efficient enough Cocke-Kasami-Younger (CKY) algorithm.
 - LR parsing.
- Parsers for English and other natural languages.

As we’ll see, the bottom line on the efficiency of context-free parsing is the following. Let n be the length of the string to be parsed. Then:

- There exists a straightforward algorithm (CKY) that can parse any context-free language in $\mathcal{O}(n^3)$ time. While this is substantially better than the exponential time required to simulate the kind of nondeterministic PDAs that we built in Section 12.3, it isn't good enough for many practical applications. In addition, CKY requires its grammar to be in Chomsky normal form. There exists a much less straightforward version of CKY that can parse any context-free language in close to $\mathcal{O}(n^2)$ time.
- There exist algorithms that can parse large subclasses of context-free languages (including many of the ones we care about, like most programming languages and query languages) in $\mathcal{O}(n)$ time. There are reasonably straightforward top-down algorithms that can be built by hand. There are more efficient, more complicated bottom-up ones. But there exist tools that make building practical bottom-up parsers very easy.
- Parsing English, or any other natural language, is harder than parsing most artificial languages, which can be designed with parsing efficiency in mind.

15.1 Lexical Analysis

Consider the input string shown in Figure 15.1 (a). It contains 32 characters, including blanks. The job of lexical analysis is to convert it into a sequence of symbols like the one shown in Figure 15.1 (b).

level = observation - 17.5; (a)

id = id - id (b)

Figure 15.1 Lexical analysis

We call each of the symbols that the lexical analyzer produces a *token*. So, in this simple example, there are 5 tokens. In addition to creating a token stream, the lexical analyzer must be able to associate, with each token, some information about how it was formed. That information will matter when it comes time to assign meaning to the input string (for example by generating code).

In principle, we could skip lexical analysis. We could instead extend every grammar to include the rules by which simple constituents like identifiers and numbers are formed.

Example 15.1 Specifying *id* with a Grammar

We could change our arithmetic expression grammar (from Example 11.19) so that *id* is a nonterminal rather than a terminal. We'd then have to add rules such as:

```

id → identifier | integer | float
identifier → letter alphanum          /* a letter followed by zero or more alphanumerics.
alphanum → letter alphanum | digit alphanum | ε
integer → - unsignedint | unsignedint /* an optional minus sign followed by an unsigned integer.
unsignedint → digit | digit unsignedint
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
...

```

But there is an easier way to handle this early part of the parsing problem. We can write regular expressions that define legal identifiers and numbers. Those regular expressions can then be compiled into deterministic finite state machines, which can run in time that is linear in the length of the input.

Useful tools for building lexical analyzers, also called *lexers*, are widely available. Lex is a good example of such a tool. The input to Lex is a set of rules. The left-hand side of each rule is a regular expression that describes the input strings to which the rule should apply. The right-hand side of each rule (enclosed in curly brackets) describes the output that should be created whenever the rule matches. The output of Lex is a lexical analyzer. When the analyzer runs, it matches its rules against an input stream. Any text that is not matched by any rule is simply echoed back into the output stream. Any text that is matched is replaced in the output stream by the right-hand side of the matching rule. The analyzer assumes a specific pattern of run-time communication between itself and a context-free parser to which it will be streaming tokens. In particular, it assumes the existence of a few shared variables, including one called *yylval*, into which the value that corresponds to the current token can be placed.

Example 15.2 Some Simple Lex Rules

Here are some simple Lex rules:

```
(1)    [ \t]+;                               /* get rid of blanks and tabs.
(2)    [A-Za-z][A-Za-z0-9]* {return (ID); }    /* find identifiers.
(3)    [0-9]+                                {scanf(yytext, "%d", &yylval);
                                             return (INTEGER); } /* return INTEGER and put the
                                             value in yyval.
```

- Rule 1 has just a left-hand side, which matches any string composed of just blanks and tabs. Since it has an empty right-hand side, the string it matches will be replaced by the empty string. So it could be used to get rid of blanks and tabs in the input if their only role is as delimiters. In this case, they will not correspond to any symbols in the grammar that the parser will use.
- Rule 2 has a left-hand side that can match any alphanumeric string that starts with a letter. Any substring it matches will be replaced by the value of its right-hand side, namely the token *id*. So this rule could be used to find identifiers. But since no information about what identifier was found is recorded, this rule is too simple for most applications.
- Rule 3 could be used to find integers. It returns the token `INTEGER`. But it also places the specific value that it matched into the shared variable *yylval*.

If two Lex rules match against a single piece of input text, the analyzer chooses between them as follows:

- A longer match is preferred over a shorter one.
- Among rules that match the same number of input characters, the one that was written first in the input to Lex is preferred.

Example 15.3 How Lex Chooses Which Rule to Apply

Suppose that Lex has been given the following two rules:

```
(1)    integer                                {action 1}
(2)    [a-z]+                                {action 2}
```

Now consider what the analyzer it builds will do on the following input sequences:

<code>integers</code>	take action 2 because rule (2) matches the entire string <code>integers</code> , while rule (1) matches only the first 7 characters.
<code>integer</code>	take action 1 because both patterns match all 7 characters and rule (1) comes first.

Lex was specifically designed as a tool for building lexical analyzers to work with parsers generated with the parser-building tool Yacc, which we will describe in Section 15.3.

15.2 Top-Down Parsing

A top-down parser for a language defined by a grammar G works by creating a parse tree with a root labeled S_G . It then builds the rest of the tree, working downward from the root, using the rules in R_G . Whenever it creates a node that is labeled with a terminal symbol, it checks to see that it matches the next input symbol. If it does, the parser continues until it has built a tree that spans the entire input string. If the match fails, the parser terminates that path and tries an alternative way of applying the rules. If it runs out of alternatives, it reports failure. For some languages, described with certain kinds of grammars, it is possible to do all of this without ever having to consider more than one path, generally by looking one character ahead in the input stream before a decision about what to do next is made. We'll begin by describing a very general parser that conducts a depth-first search and typically requires backtracking. Then we'll consider grammar restrictions that may make deterministic top-down parsing possible.

15.2.1 Depth-First Search

We begin by describing a simple top-down parser that works in essentially the same way that the top-down PDAs that we built in Section 12.3.1 did. It attempts to reconstruct a left-most derivation of its input string. The only real difference is that it is now necessary to describe how nondeterminism will be handled. We'll use depth-first search with backtracking. The algorithm that we are about to present is similar to *decideCFLusingPDA*. They are both nondeterministic, top-down algorithms. But the one we present here, in contrast to *decideCFLusingPDA*, does not require a grammar in any particular form.

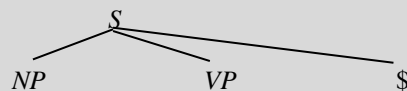
Example 15.4 Top-Down, Depth-First Parsing

To see how a how a depth-first, top-down parser works, let's consider an English grammar that is even simpler than the one we used in Example 11.6. This time, we will require that every sentence end with the end-of-string marker \$:

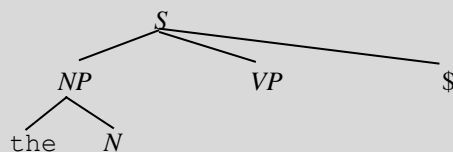
```
 $S \rightarrow NP VP \$$   
 $NP \rightarrow \text{the } N \mid N \mid \text{ProperNoun}$   
 $N \rightarrow \text{cat} \mid \text{dogs} \mid \text{bear} \mid \text{girl} \mid \text{chocolate} \mid \text{rifle}$   
 $\text{ProperNoun} \rightarrow \text{Chris} \mid \text{Fluffy}$   
 $VP \rightarrow V \mid VNP$   
 $V \rightarrow \text{like} \mid \text{likes} \mid \text{thinks} \mid \text{shot} \mid \text{smells}$ 
```

On input the cat likes chocolate \$, the parser, given these rules, will behave as follows:

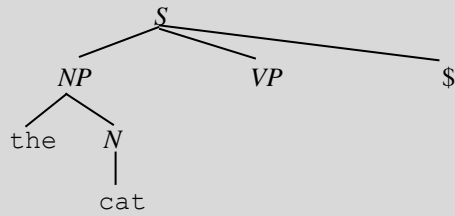
- Build an S using the only rule available



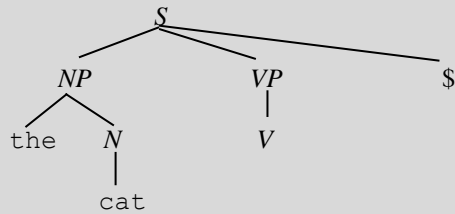
- Build an NP . Start with the first alternative, which successfully matches the first input symbol:



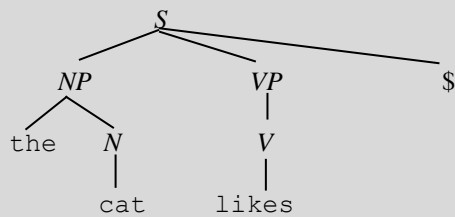
- Build an *N*. Start with the first alternative, which successfully matches the next input symbol:



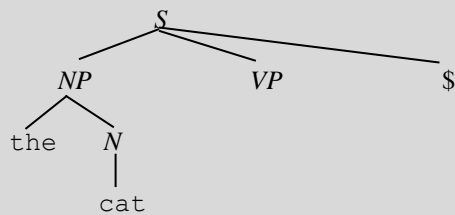
- Build a *VP*. Start with the first alternative:



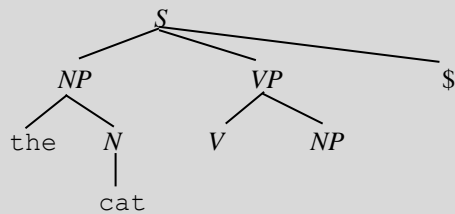
- Build a *V*. The first alternative, *like*, fails to match the input. The second, *likes*, matches:



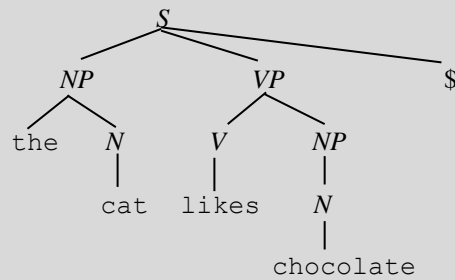
- Match *\$*. This fails, since the word *chocolate* remains in the input. So the process undoes decisions, in order, until it has backtracked to:



- Build a *VP*. This time, try the second alternative:



- Continue until a tree that spans the entire input has been built:



While parsers such as this are simple to define, there are two problems with using them in practical situations:

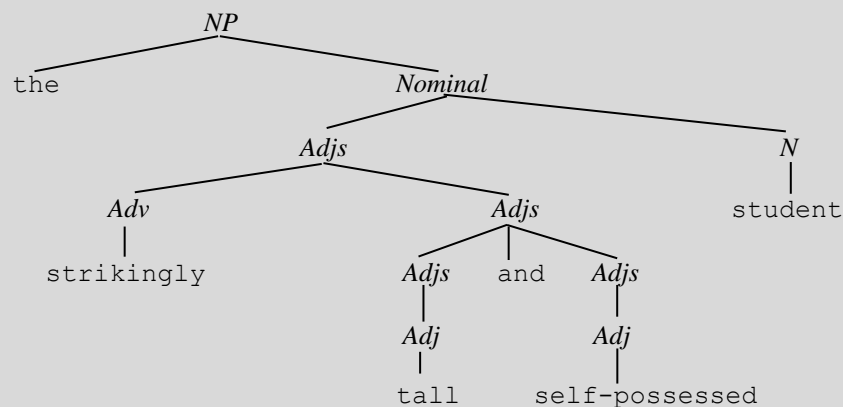
- It is possible to get into an infinite loop, even when there is a correct parse for the input.
- Backtracking is expensive. Some constituents may be built and unbuilt many times. For example, the constituent *V – likes* was built twice in the simple sentence shown in Example 15.4. We'll illustrate this problem on a larger scale in the next example.

Example 15.5 Subtrees May Be Built and Discarded Many Times

Suppose we have the following rules for noun phrases:

NP → the *Nominal* | *Nominal* | *ProperNoun* | *NP PP*
Nominal → *N* | *Adjs N*
Adjs → *Adv Adjs* | *Adjs and Adjs* | *Adj Adjs* | *Adj*
N → student | raincoat
Adj → tall | self-possessed | green
Adv → strikingly
PP → *Prep NP*
Prep → with

Now consider the noun phrase `the strikingly tall and self-possessed student with the green raincoat`. In an attempt to parse this phrase as an *NP*, a depth-first, top-down parser will first try to use the rule *NP* → the *Nominal*. In doing so, it will build the tree:



Then it will notice that four symbols, with the `green raincoat`, remain. At that point, it will have to back all the way up to the top *NP* and start over, this time using the rule *NP* → *NP PP*. It will eventually build an *NP* that

spans the entire phrase⁶. But that *NP* will have, as a constituent, the one we just built and threw away. So the entire tree we showed above will have to be rebuilt from scratch.

Because constituents may be built and rebuilt many times, the depth-first algorithm that we just sketched may take time that is $\mathcal{O}(g^n)$, where g is the maximum number of alternatives for rewriting any nonterminal in the grammar and n is the number of input symbols.

Both the problem of infinite loops and the problem of inefficient rebuilding of constituents during backtracking can sometimes be fixed by rearranging the grammar and/or by looking ahead one or more characters before making a decision. In the next two sections we'll see how this may be done.

15.2.2 Modifying a Grammar for Top-Down Parsing

Some grammars are better than others for top-down parsing. In this section we consider two issues: preventing the parser from getting into an infinite loop and using lookahead to reduce nondeterminism.

Left-Recursive Rules and Infinite Loops

A top-down parser can get into an infinite loop and fail to find a complete parse tree, even when there is one.

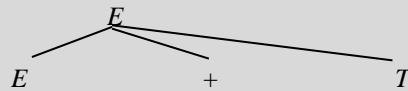
Example 15.6 Backtracking Gets Stuck on Left-Recursive Rules

We consider again the term/factor grammar for arithmetic expressions of Example 11.19:

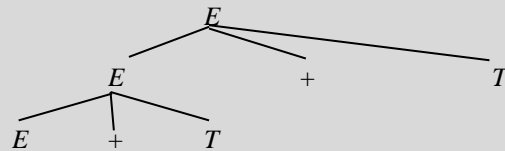
$$\begin{aligned} E &\rightarrow E + T \\ E &\rightarrow T \\ T &\rightarrow T * F \\ T &\rightarrow F \\ F &\rightarrow (E) \\ F &\rightarrow \text{id} \end{aligned}$$

On input `id + id + id`, a top-down parser will behave as follows:

- Build an E , using the first alternative:



- Build an E , using the first alternative:



- Build an E , using the first alternative, and so forth, forever, expanding the leftmost E as $E + T$.

The problem is the existence in the grammar of left-recursive rules like $E \rightarrow E + T$ and $T \rightarrow T * F$. Paralleling the definition we gave in Section 11.2 for a recursive rule, we say that a grammar rule is *left-recursive* iff it is of the form $X \rightarrow Y w_2$ and $Y \Rightarrow_G^* X w_4$, where w_2 and w_4 may be any element of V^* . If the rules were rewritten so that the recursive

⁶ A separate issue is that this phrase is ambiguous. We've shown the parse that corresponds to the bracketing strikingly (tall and self-possessed). An alternative parse corresponds to the bracketing (strikingly tall) and self-possessed.

symbols were on the right of the right-hand side rather than on the left of it, the parser would be able to make progress and consume the input symbols.

We first consider direct recursion, i.e., rules of the form $X \rightarrow X w_2$. This case includes the rules $E \rightarrow E + T$ and $T \rightarrow T * F$. Suppose that such a rule is used to derive a string in $L(G)$. For example, let's use the rule $E \rightarrow E + T$. Then there is a derivation that looks like:

$$E \Rightarrow E + T \Rightarrow E + T + T \Rightarrow \dots \Rightarrow T + T \dots + T \Rightarrow \dots$$

In other words, the left-recursive rule is applied some number of times but then the recursion stops and some nonrecursive rule with the same left-hand side is applied. In this example, it was the rule $E \rightarrow T$. The left-most symbol in the string we just derived came from the nonrecursive rule. So an alternative way to generate that string would be to generate that leftmost T first, by applying once a new rule $E \rightarrow T E'$. Then we can generate the rest by applying, as many times as necessary, a new recursive (but not left-recursive) rule $E' \rightarrow + T E'$, followed by a clean-up rule $E' \rightarrow \varepsilon$, which will stop the recursion.

Applying this idea to our arithmetic expression grammar, we get the new grammar:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \\ E' &\rightarrow \varepsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F T' \\ T' &\rightarrow \varepsilon \\ F &\rightarrow (E) \\ F &\rightarrow \text{id} \end{aligned}$$

We can describe what we just did more generally as follows: Given any context-free grammar G , if G contains any left-recursive rule with left-hand side A , then consider all rules in G with left-hand side A . Divide them into two groups, the left-recursive ones and the others. Replace all of them with new rules, as shown in Table 15.1.

Original left-recursive rules:	Replace with:
$A \rightarrow A\alpha_1$	$A' \rightarrow \alpha_1 A'$
$A \rightarrow A\alpha_2$	$A' \rightarrow \alpha_2 A'$
...	...
$A \rightarrow A\alpha_n$	$A' \rightarrow \alpha_n A'$
	$A' \rightarrow \varepsilon$
Original nonleft-recursive rules:	Replace with:
$A \rightarrow \beta_1$	$A \rightarrow \beta_1 A'$
$A \rightarrow \beta_2$	$A \rightarrow \beta_2 A'$
...	...
$A \rightarrow \beta_m$	$A \rightarrow \beta_m A'$

Table 15.1 Eliminating left-recursive rules

If, in addition to removing left-recursion, we want to avoid introducing ε -rules, we can use a variant of this algorithm. Instead of always generating A' and then erasing it at the end of the recursive part of a derivation, we create rules that allow it not to be generated. So we replace each original left-recursive rule, $A \rightarrow A\alpha_k$, with *two* new rules: $A' \rightarrow \alpha_k A'$ and $A' \rightarrow \alpha_k$. Instead of replacing each original nonleft-recursive rule, $A \rightarrow \beta_k$, we keep it and add the new rule: $A \rightarrow$

$\beta_k A'$. We do not add the rule $A' \rightarrow \epsilon$. Because we will have another use for this variant of the algorithm (in converting grammars to Greibach normal form), we will give it the name *removeleftrecursion*(N : nonterminal symbol).

Unfortunately, the technique that we have just presented, while it does eliminate direct left recursion, does not solve the entire problem. Consider the following grammar G :

$$\begin{aligned} S &\rightarrow Ya \\ Y &\rightarrow Sa \\ Y &\rightarrow \epsilon \end{aligned}$$

G contains no directly left-recursive rules. But G does contain left-recursive rules, and a top-down parser that used G would get stuck building the infinite left-most derivation $S \Rightarrow Ya \Rightarrow Saa \Rightarrow Yaaa \Rightarrow Saaaa \Rightarrow \dots$. It is possible to eliminate this kind of left-recursion as well by using an algorithm that loops through all the nonterminal symbols in G and applies the algorithm that we just presented to eliminate direct left-recursion.

So left-recursion can be eliminated and the problem of infinite looping in top-down parsers can be solved. Unfortunately, the elimination of left-recursion comes at a price. Consider the input string `id + id + id`. Figure 15.2 (a) shows the parse trees that will be produced for it using our original expression grammar. Figure 15.2(b) shows the new one, with no left-recursive rules.

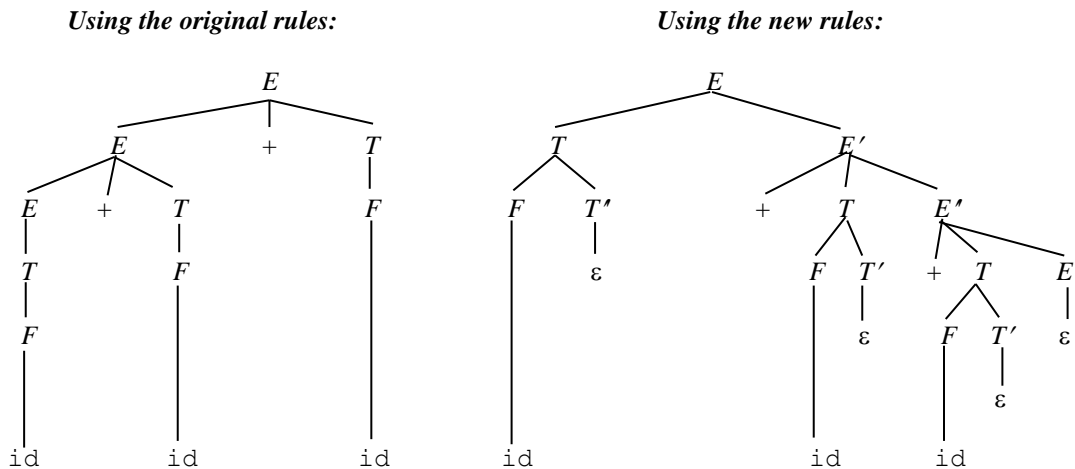


Figure 15.2 Removing left-recursion leads to different parse trees

Notice that, in the original parse tree, the `+` operator associates left, while in the new parse tree it associates right. Since the goal of producing a parse tree is to serve as the first step toward assigning meaning to the input string (for example, by writing code to correspond to it), this change is significant. In order to solve our parsing problem, we've changed the meaning of at least some strings in the language we are trying to parse.

Using Lookahead and Left Factoring to Reduce Nondeterminism

As we saw in Example 15.4 (the simple English example), a depth-first, top-down parser may have to explore multiple derivations before it finds the one that corresponds to the current input string. The process we just described for getting rid of left-recursive rules does nothing to affect that. So we would still like to find a technique for reducing or eliminating the need for search. Sometimes it is possible to analyze a grammar in advance and determine that some paths will never lead to a complete derivation of a string of terminal symbols. But the more important case arises, as it did even with our very simple grammar of English, when the correct derivation depends on the current input string. When that happens, the best source of guidance for the parser is the input string itself and its best strategy is to

procrastinate branching as long as possible in order to be able to use the input to inform its decisions. To implement this strategy, we'll consider doing two things:

- Changing the parsing algorithm so that it exploits the ability to look one symbol ahead in the input before it makes a decision about what to do next, and
- Changing the grammar to help the parser procrastinate decisions.

We can explore both of these issues by considering just a fragment of our arithmetic-expression grammar, which we'll augment with one new rule that describes simple function calls. So consider just the following set of three rules:

- (1) $F \rightarrow (E)$
- (2) $F \rightarrow \text{id}$
- (3) $F \rightarrow \text{id}(E)$ /* this is a new rule that describes a call to a unary function.

If a top-down parser needs to expand a node labeled F , which rule should it use? If it can look one symbol ahead in the input before deciding, it can choose between rule (1), which it should apply if the next character is $($, and rules (2) and (3), one of which should be applied if the next symbol is id .

But how can a parser choose between rules (2) and (3) if it can look only one symbol ahead? The answer is to change the grammar so that the decision can be procrastinated. In particular, we will rewrite the grammar as:

- (1) $F \rightarrow (E)$
- (1.5) $F \rightarrow \text{id } X$
- (2) $X \rightarrow \varepsilon$
- (3) $X \rightarrow (E)$

Now, if the lookahead symbol is id , the parser will apply rule (1.5). Then it will match the id and set the lookahead symbol to the following symbol. Next it must decide whether to expand X by rule (2) or rule (3). But it is one symbol farther along in the input as it faces this decision. If the next input symbol is $($, it is possible that either rule should be chosen (although see below for an additional technique that may resolve this conflict as well). But if the next input symbol is anything else, only rule (2) can possibly lead to a complete parse.

The operation that we just did is called *left factoring*. It can be described as follows:

Let G be a context-free grammar that contains two or more rules with the same left-hand side and the same initial sequence of symbols on the right-hand side. Suppose those rules are:

- $$\begin{aligned} A &\rightarrow \alpha\beta_1 \\ A &\rightarrow \alpha\beta_2 \\ &\dots \\ A &\rightarrow \alpha\beta_n \end{aligned}$$

where $\alpha \neq \varepsilon$ and $n \geq 2$. We remove those rules from G and replace them with the rules:

- $$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \\ A' &\rightarrow \beta_2 \\ &\dots \\ A' &\rightarrow \beta_n \end{aligned}$$

A parser that uses this new grammar will still have to make a decision about what to do after it has read the input sequence α . But it will probably be farther along in the input string by the time it has to do that.

15.2.3 Deterministic Top-Down Parsing with LL(1) Grammars

Can we do better? We know, from Theorem 13.13, that there exist context-free languages for which no deterministic PDA exists. So there will be context-free languages for which the techniques that we have just described will not be able to remove all sources of nondeterminism. But do we care about them? Could we build a deterministic, linear-time parser for:

1. A typical programming language like Java or C++ or Haskell?
2. A typical database query language?
3. A typical Web search query language?
4. English or Chinese?

The answer to questions 1-3 is yes. The answer to question 4 is mostly no, although there have been some partially successful attempts to do so. Using techniques such as the ones we just described, it is sometimes possible to craft a grammar for which a deterministic top-down parser exists. Such parsers are often also called *predictive parsers*. To simplify the rest of this discussion, assume that every input string ends with the end-of-string marker \$. This means that, until after the \$ is reached, there is always a next symbol, which we will call the lookahead character.

It will be possible to build a predictive top-down parser for a grammar G precisely in case every string that is generated by G has a unique left-most derivation and it is possible to determine each step in that derivation by looking ahead some fixed number k of characters in the input stream. In this case, we say that G is $LL(k)$, so named because an $LL(k)$ grammar allows a predictive parser that scans its input left to right (the origin of the first L in the name) to build a left-most derivation (the origin of the second L) if it is allowed k lookahead symbols. Note that every $LL(k)$ grammar is unambiguous (because every string it generates has a unique left-most derivation). It is not the case, however, that every unambiguous grammar is $LL(k)$.

Most predictive parsers use a single lookahead symbol. So we are interested in determining whether or not a grammar G is $LL(1)$. To do so, it is useful to define two functions:

- Given a grammar G and a sequence of symbols α , define $first(\alpha)$ to be the set of all terminal symbols that can occur as the first symbol in any string derived from α using R_G . If α derives ϵ , then $\epsilon \in first(\alpha)$.
- Given a grammar G and a nonterminal symbol A , define $follow(A)$ to be the set of all terminal symbols that can immediately follow whatever A produces in some string in $L(G)$.

Example 15.7 Computing $first$ and $follow$

Consider the following simple grammar G :

$$\begin{aligned} S &\rightarrow AXB\$ \\ A &\rightarrow aA \mid \epsilon \\ X &\rightarrow c \mid \epsilon \\ B &\rightarrow bB \mid \epsilon \end{aligned}$$

- $first(S) = \{a, c, b, \$\}$.
- $first(A) = \{a, \epsilon\}$.
- $first(AX) = \{a, c, \epsilon\}$.
- $first(AXB) = \{a, c, b, \epsilon\}$.
- $follow(S) = \emptyset$.
- $follow(A) = \{c, b, \$\}$.
- $follow(X) = \{b, \$\}$.
- $follow(B) = \{\$\}$.

We can now state the conditions under which a grammar G is $LL(1)$. It is iff, whenever G contains two competing rules $A \rightarrow \alpha$ and $A \rightarrow \beta$, all of the following are true:

- There is no terminal symbol that is an element of both $first(\alpha)$ and $first(\beta)$.
- ϵ cannot be derived from both of α and β .
- If ϵ can be derived from one of α or β , assume it is α . Then there may be two competing derivations:

$$\begin{array}{lcl}
S & \Rightarrow \gamma_1 A \gamma_2 & \text{and} \\
& \Rightarrow \gamma_1 \alpha \gamma_2 & \\
& \Rightarrow \gamma_1 \quad \gamma_2 & \\
S & \Rightarrow \gamma_1 A \gamma_2 & \\
& \Rightarrow \gamma_1 \beta \gamma_2 &
\end{array}$$

Consider the information available to a predictive parser when it has to choose how to expand A . It has consumed the input up through γ_1 . So, when it looks one character ahead, it will find the first character of γ_2 (in case $A \Rightarrow \alpha \Rightarrow \epsilon$) or it will find the first character of β (in case $A \Rightarrow \beta$). So we require that there be no terminal symbol that is an element of both $follow(A)$ (which describes the possible first terminal symbols in γ_2) and $first(\beta)$.

We define a language to be $LL(k)$ iff there exists an $LL(k)$ grammar for it. Not all context-free languages are $LL(k)$ for any fixed k . In particular, no inherently ambiguous one is, since every $LL(k)$ grammar is unambiguous. There are also languages for which there exists an unambiguous grammar but no $LL(k)$ one. For example, consider $\{a^n b^n c^m d : n, m \geq 0\} \cup \{a^n b^m c^m e : n, m \geq 0\}$, which is unambiguous, but not $LL(k)$, for any k , since there is no fixed bound on the number of lookahead symbols that must be examined in order to determine whether a given input string belongs to the first or the second sublanguage. There are even deterministic context-free languages that are not $LL(k)$, for any k . One such example is $\{a^n b^n, n \geq 0\} \cup \{a^n c^n, n \geq 0\}$. (Intuitively, the problem there is that, given a string w , it is not possible to determine the first step in the derivation of w until either a b or a c is read.) But many practical languages are $LL(k)$. In fact, many are $LL(1)$, so it is worth looking at ways to exploit this property in the design of a top-down parser.

There are two reasonably straightforward ways to go about building a predictive parser for a language L that is described by an $LL(1)$ grammar G . We consider each of them briefly here.

Recursive Descent Parsing

A *recursive-descent parser* contains one function for each nonterminal symbol A in G . The argument of each such function is a parse tree node labeled A , and the function's job is to create the appropriate parse tree beneath the node that it is given. The function corresponding to the nonterminal A can be thought of as a case statement, with one alternative for each of the ways that A can be expanded. Each such alternative checks whether the next chunk of input could have been derived from A using the rule in question. It checks each terminal symbol directly. To check each nonterminal symbol, it invokes the function that is defined for it. The name "recursive descent" comes from the fact that most context-free grammars contain recursive rules, so the parser will typically exploit recursive function calls.

Example 15.8 Recursive Descent Parsing

Let G include the rules:

$$\begin{array}{l}
A \rightarrow BA \mid a \\
B \rightarrow bB \mid b
\end{array}$$

The function associated with A will then be (ignoring many details, including how the next lookahead symbol is computed, how the parse tree is actually built, and what happens on input strings that are not in $L(G)$):

```

A(n: parse tree node labeled A) =
  case (lookahead = b : /* Use the rule A → BA.
                        Invoke B on a new daughter node labeled B.
                        Invoke A on a second new daughter node labeled A.
                        lookahead = a : /* Use the rule A → a.
                        Create a new daughter node labeled a.

```

Table-Driven $LL(1)$ Parsing

Instead of letting a set of recursive function calls implicitly maintain a stack, we could build a parser that works in much the same way that the top-down PDAs of Section 12.3.1 do. Such a parser would maintain its stack explicitly. Consider all of the transitions that such a parser can take. We can index them in a table called a *parse table*, which contains one row for each nonterminal that could be on the top of the stack and one column for each terminal symbol

that could correspond to the lookahead symbol. Then we can build a straightforward table-driven parser that chooses its next move by using the current top-of-stack and lookahead symbols as indices into the table.

Example 15.9 Building a Parse Table

Let G be:

$$\begin{aligned} S &\rightarrow AB\$ \mid AC\$ \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid b \\ C &\rightarrow c \end{aligned}$$

The parse table for G would be:


<i>Lookahead symbol</i> \ <i>Top of stack</i>	a	b	c	\$
<i>S</i>	$S \rightarrow AB\$$ $S \rightarrow AC\$$			
<i>A</i>	$A \rightarrow aA$ $A \rightarrow a$			
<i>B</i>		$B \rightarrow bB$ $B \rightarrow b$		
<i>C</i>			$C \rightarrow c$	

Notice two things about the parse table that we just built:

- Many of the cells are empty. If the parser looks in the table and finds an empty cell, it knows that it has hit a dead-end: the path it is currently following will never succeed in parsing the input string.
- Some of the cells contain more than one rule. A parser that used that table as the basis for choosing its next move would thus be nondeterministic. Suppose, on the other hand, we could guarantee that the table contained at most one rule in each cell. Then a parser that was driven by it would be deterministic.

Given any LL(1) grammar G , it is possible to build a parse table with at most one rule in each cell. Thus it is possible to build a deterministic (predictive) table-driven parser for G . The parser simply consults the table at each step and applies the rule that is specified.

Note that the grammar of Example 15.9 is not LL(1) because a is an element of both $first(AB\$)$ and $first(AC\$)$. Thus there are two ways to expand S if the lookahead symbol is a . There are also two ways to expand A if the lookahead symbol is a and two ways to expand B if the lookahead symbol is b . But the language described by that grammar is LL(1). We leave the construction of an LL(1) grammar for it as an exercise.

LL(1) parsers can be built by hand, but there exist tools  that greatly simplify the process.

15.3 Bottom-Up Parsing

Rather than parsing top-down, as we have just described, an alternative is to parse bottom-up, and thus drive the process directly by the current string of input symbols.

A bottom-up parser for a language defined by a grammar G works by creating the bottom nodes of a parse tree and labeling them with the terminal symbols in the input. Then it attempts to build a complete parse tree above those nodes. It does this by applying the rules in R_G backwards. In other words, suppose that a sequence of nodes labeled x_1, x_2, \dots, x_n has already been built and R_G contains the rule:

$$X \rightarrow x_1, x_2, \dots, x_n$$

Then the parser can build a node, label it X , and insert the nodes labeled x_1, x_2, \dots, x_n as its children. If the parser succeeds in building a tree that spans the entire input and whose root is labeled S_G , then it has succeeded. If there is no path by which it can do that, it fails and reports that the input string is not in $L(G)$.

Since there may be choices for which rule to apply at any point in this process, a bottom-up parser patterned after the PDAs that we built in Section 12.3.1 may be nondeterministic and its running time may grow exponentially in the length of the input string. That is clearly unacceptable for a practical parser. In the next section we describe a straightforward bottom-up parsing algorithm with running time that is $\mathcal{O}(n^3)$. But even that may not be good enough. Fortunately, just as we did for top-down parsing, we can construct a deterministic parser that runs in time that is $\mathcal{O}(n)$ if we impose some restrictions on the grammars that we use.

15.3.1 The Cocke-Kasami-Younger Algorithm

A straightforward, bottom-up parser that handles nondeterminism by backtracking typically wastes a lot of time building and rebuilding nodes as it backtracks. An alternative is a dynamic programming approach in which each possible constituent is built (bottom-up) exactly once and then made available to any later rules that want to use it.

The Cocke-Kasami-Younger algorithm (also called CKY or CYK) works by storing such constituents in a two dimensional table T that contains one column for each input symbol and one row for each possible substring length. Call the input string w and let its length be n . Then T contains one row and one column for each integer between 1 and n . We will number the rows of T starting at the bottom and the columns starting from the left. For all i and j between 1 and n , each cell $T[i, j]$ corresponds to the substring of w that extends for i symbols and starts in position j . The value that will eventually fill each such cell will be the set of nonterminal symbols that could derive the string to which the cell corresponds. For example, to parse the string $\text{id} + \text{id} * \text{id}$, we would need to fill in the cells as shown in Table 15.2. Note that each cell is labeled with the substring to which it corresponds, not with the value it will eventually take on.

Row 5	id + id * id				
Row 4	id + id *	+ id * id			
Row 3	id + id	+ id *	id * id		
Row 2	id +	+ id	id *	* id	
Row 1	id	+	id	*	id

Input string: **id** **+** **id** ***** **id**

Table 15.2 The table that a CKY parser builds

Let G be the grammar that is to be used. Initially, each cell in T will be blank. The parser will begin filling in T , starting from the bottom, and then moving upward to row n . When it is complete, each cell in the lower triangle of T will contain the set of nonterminal symbols that could have generated the corresponding substring. If the start symbol of G occurs in $T[n, 1]$, then G can generate the substring that starts in position 1 and has length n . But that is exactly w . So G can generate w .

The CKY algorithm requires that the grammar that it uses be in Chomsky normal form. Recall that, in a Chomsky normal form grammar, all rules have one of the following two forms:

- $X \rightarrow a$, where $a \in \Sigma$, or
- $X \rightarrow BC$, where B and C are elements of $V - \Sigma$.

So we need two separate techniques for filling in T :

- To fill in row 1, use rules of the form $X \rightarrow a$. In particular, if $X \rightarrow a$ and a is the symbol associated with column j , then add X to $T[1, j]$.
- To fill in rows 2 through n , use rules of the form $X \rightarrow BC$, since they are the ones that can combine constituents to form larger ones. Suppose the parser is working on some cell in row k . It wants to determine whether the rule $X \rightarrow BC$ can be used to generate the corresponding substring s of length k . If it can, then there must be some way to divide s into exactly two constituents, one corresponding to B and the other corresponding to C . Since both of those constituents must be shorter than s , any ways there are of building them must already be represented in cells in rows below k .

We can now state the CKY algorithm as follows:

```

CKY( $G$ : Chomsky normal form grammar,  $w = a_1a_2 \dots a_n$ : string) =
  /* Fill in the first (bottom-most) row of  $T$ , checking each symbol in  $w$  and finding all the nonterminals
  that could have generated it.
  1. For  $j = 1$  to  $n$  do:
      If  $G$  contains the rule  $X \rightarrow a_j$ , then add  $X$  to  $T[1, j]$ .

  /* Fill in the remaining rows, starting with row 2 and going upward.
  2. For  $i = 2$  to  $n$  do:                /* For each row after the first
      For  $j = 1$  to  $n-i+1$  do:          /* For each column in the lower triangle of  $T$ 
          For  $k = 1$  to  $i-1$  do:        /* For each character after which there could be a split
              into two constituents
              For each rule  $X \rightarrow YZ$  do:
                  If  $Y \in T[k, j]$  and  $Z \in T[i-k, j+k]$ , then:          /*  $Y$  and  $Z$  found.      #####
                  Insert  $X$  into  $T[i, j]$ .

  3. If  $S_G \in T[n, 1]$  then accept else reject.
  
```

The core matching operation occurs in the step flagged with #####. The parser must determine whether X could have generated the substring that starts in position j and has length i . It is currently considering splitting that substring after the k^{th} symbol. So it checks whether Y could have generated the first piece, namely the one that starts in position j and has length k . And it checks whether Z could have generated the second piece, namely the one that starts in position $j + k$ and whose length is equal to the length of the original substring minus the part that Y matched. That is $i - k$.

Example 15.10 The CKY Algorithm

Consider parsing the string `aab` with the grammar:

```

S → AB
A → AA
A → a
B → a
B → b
  
```

CKY begins by filling in the bottom row of T as follows:

Row 3			
Row 2			
Row 1	A, B	A, B	B
Input string:	a	a	b

Notice that, at this point, the algorithm has no way of knowing whether the `a`'s were generated by A or by B .

Next, the algorithm moves on to step 2. Setting i to 2, it fills in the second row, corresponding to substrings of length 2, as follows: When i is 2 and j is 1, it is considering ways of generating the initial substring aa . Setting k to 1, it considers splitting it into a and a . Then, considering the rule $S \rightarrow AB$, it finds the A and the B that it needs in row 1, so it adds S to $T[2, 1]$. Similarly for the rule $A \rightarrow AA$, so it adds A to $T[2, 1]$. It then sets j to 2 and looks at ways of generating substrings that start in position 2. Setting k to 1, it considers splitting ab into a and b . Considering the rule $S \rightarrow AB$, it finds the A and the B that it needs, so it adds S to $T[2, 2]$. At this point, T is:

Row 3			
Row 2	S, A	S	
Row 1	A, B	A, B	B
Input string:	a	a	b

Next CKY sets i to 3. So it is considering strings of length 3. There is only one, namely the one that starts at position 1. So the only value of j that will be considered is 1. There are now two values of k to consider, since there are two ways that the string aba can be split in two. Setting k to 1, it is considering the constituents a and ab . Considering the rule $S \rightarrow AB$, it looks for an A of length 1 starting in position 1 (which it finds) and a B of length 2 starting in position 2 (which it fails to find). It then considers the other rule, $A \rightarrow AA$. For this rule to succeed there would have to be an A of length 1 in position 1 (which it finds) and a second A of length 2 starting in position 2 (which it fails to find). Notice that, since it needs an A of length 2, it must look in row 2. The A in row 1 doesn't help. So it has found nothing by breaking the string after position 1. It sets k to 2 and considers breaking it, after position 2, into aa and b . Now it again tries the first rule, $S \rightarrow AB$. It looks in row 2 for an A that generated aa and in row 1 for a B that generated b . It finds both and inserts S into $T[3, 1]$. So it accepts.

The algorithm that we just presented does not actually build a parse tree. It simply decides whether the string w is in $L(G)$. It can easily be modified to build parse trees as it applies rules. Then the final parse tree for w is the one associated with the start symbol in $T[n, 1]$. If G is ambiguous, there may be more than one such tree.

We can analyze the complexity of CKY as follows: we will assume that the size of the grammar G is a constant, so any operation whose complexity is dependent only on the size of G takes constant time. This means that the code inside the loop of step 1 and the testing of all grammar rules that is done inside the loop of step 2 each take constant time. Step 1 takes time that is $\mathcal{O}(n)$. Step 2 can be analyzed as follows:

- The outer loop (i) is executed $n-1$ times.
- The next loop (j) is executed, on average $n/2$ times and at most $n-1$ times.
- The next loop (k) is also executed, on average $n/2$ times and at most $n-1$ times.
- The inner loop takes constant time.

So step 2 takes time that is $\mathcal{O}((n-1)(n/2)(n/2)) = \mathcal{O}(n^3)$. Step 3 takes constant time. So the total time is $\mathcal{O}(n^3)$.

If we also want to consider the size of G , then let $|G|$ be the number of rules in G . If G is in Chomsky Normal form, CKY takes time that is $\mathcal{O}(n^3 \cdot |G|)$. But if G is not already in Chomsky Normal form, it must first be converted, and that process can take time that is $\mathcal{O}(|G|^2)$. So we have that the total time required by CKY is $\mathcal{O}(n^3 \cdot |G|^2)$.

15.3.2 Context-Free Parsing and Matrix Multiplication

The CKY algorithm can be described in terms of Boolean matrix multiplication. Stated that way, its time efficiency depends on the efficiency of Boolean matrix multiplication. In particular, again assuming that the size of the grammar is constant, the running time becomes $\mathcal{O}(M(n))$, where $M(n)$ is the time required to multiply two $n \times n$ Boolean matrices. Straightforward matrix multiplication algorithms (such as Gaussian elimination) take time that is $\mathcal{O}(n^3)$, so this recasting of the algorithm has no effect on its complexity. But faster matrix multiplication algorithms exist. For example, Strassen's algorithm (described in Exercise 27.9)) reduces the time to $\mathcal{O}(n^{2.807})$, but at a price of increased complexity and a structure that makes it less efficient for small to medium values of n . The fastest known technique,

the Coppersmith-Winograd algorithm [1], has worst case running time that is $\mathcal{O}(n^{2.376})$, but it is too complex to be practical.

More recently, a further result [2] that links matrix multiplication and context-free parsing has been shown: let P be any context-free parser with time complexity $\mathcal{O}(gn^{3-\epsilon})$, where g is the size of the grammar and n is the length of the input string. Then P can be efficiently converted into an algorithm to multiply two $n \times n$ Boolean matrices in time $\mathcal{O}(n^{3-\epsilon/3})$. So, if there were a fast algorithm for parsing arbitrary context-free languages, there would also be a fast matrix multiplication algorithm. Substantial effort over the years has been expended looking for a fast matrix multiplication algorithm, and none has been found. So it appears relatively unlikely that there is a fast general algorithm for context-free parsing.

15.3.3 Shift-Reduce Parsing

The CKY algorithm works for any context-free language, but it has two important limitations:

- It is not efficient enough for many applications. We'd like a deterministic parser that runs in time that is linear in the length of the input.
- It requires that the grammar it uses be stated in Chomsky normal form. We'd like to be able to use more natural grammars and thus to extract more natural parse trees.

We'll next consider a bottom-up technique that can be made deterministic for a large, practically significant set of languages.

The parser that we are about to describe is called a *shift-reduce parser*. It will read its input string from left to right and can perform two basic operations:

1. Shift an input symbol onto the parser's stack and build, in the parse tree, a terminal node labeled with that input symbol.
2. Reduce a string of symbols from the top of the stack to a nonterminal symbol, using one of the rules of the grammar. Each time it does this, it also builds the corresponding piece of the parse tree.

We'll begin by considering a shift-reduce parser that may have to explore more than one path. Then we'll look at ways to make it deterministic in cases where that is possible.

To see how a shift-reduce parser might work, let's trace its operation on the string $id + id * id$, using our original term/factor grammar for arithmetic expressions:

- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T * F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow id$

We'll number the main steps in this process so that we can refer to them later.

(Step 1) When we start, the parser's stack is empty, so our only choice is to shift the first input symbol, id , onto the stack. Next, we have a choice. We can either use rule (6) to reduce id to F , or we can get the next input symbol and shift it onto the stack. It's clear that we need to apply rule (6) now. Why? Because there are no other rules that can consume an id directly. So we have to do this reduction before we can do anything else with id . But could we wait and do it later? No, because reduction always applies to the symbols at the top of the stack. If we push anything on before we reduce id , we'll never again get id at the top of the stack. It will just sit there, unable to participate in any rules. So we reduce id to F , giving us a stack containing just F , and the parse tree (remember we're building it up from the bottom):



The reasoning we just did is going to be the basis for the design of a “smart” deterministic bottom up parser. Without that reasoning, a dumb, brute force parser would have had to consider both paths at this first choice point: the one we took, as well as the one that fails to reduce and instead pushes + onto the stack. That second path will eventually reach a dead end, so even a brute force parser will eventually get the right answer. But our goal is to eliminate search.

(Step 2) At this point, the parser’s stack contains F and the remaining input is $+ id * id$. Again we must choose between reducing the top of the stack or shifting on the next input symbol. Again, by looking ahead and analyzing the grammar, we can see that eventually we will need to apply rule (1). To do so, the first id will have to have been reduced to a T and then to an E . So let’s next reduce by rule (4) and then again by rule (2), giving the parse tree and stack:



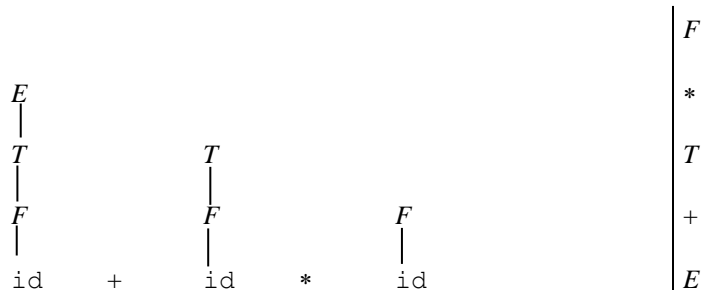
(Step 3) At this point, there are no further reductions to consider, since there are no rules whose right-hand side is just E . So we must consume the next input symbol + and shift it onto the stack. Having done that, there are again no available reductions. So we shift the next input symbol. The stack then contains $id + E$ (writing the stack with its top to the left). Again, we need to reduce id before we can do anything else, so we reduce it to F and then to T . Now we’ve got:



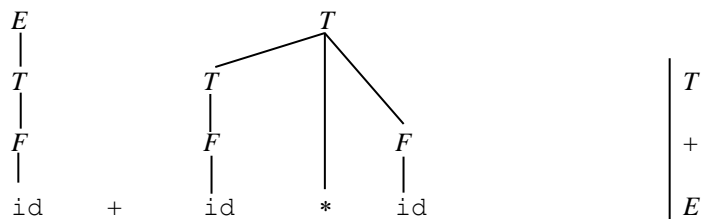
Notice that we have three parse tree fragments. Since we’re working up from the bottom, we don’t know yet how they’ll get put together. We now have three choices: reduce T to E using rule (2), reduce $T + E$ to E using rule (1) or shift on the next input symbol. Note, by the way, that we will always be matching the right-hand sides of the rules in reverse because the last symbol we read (and thus the right-most one we’ll match) is at the top of the stack.

(Step 4) In considering whether to reduce or shift at this point, we realize that, for the first time, there isn’t one correct answer for all input strings. When there was just one universally correct answer, we could compute it simply by examining the grammar. Now we can’t do that. In the example we’re working with, we don’t want to do either of the reductions, since the next input symbol is *. We know that the only complete parse tree for this input string will correspond to the interpretation in which * is applied before +. That means that + must be at the top of the tree. If we reduce now, it will be at the bottom. So we need to shift * onto the stack and do a reduction that will build the multiplication piece of the parse tree before we do a reduction involving +. But if the input string had been $id + id + id$, we’d want to reduce now in order to cause the first + to be done first, thus producing left associativity. So we appear to have reached a point where we’ll have to branch. If we choose the wrong path, we’ll eventually hit a dead end and have to back up. We’d like not to waste time exploring dead end paths, however. We’ll come back later to the question of how we can make a parser know how to avoid dead ends. For now, let’s just forge ahead and do the right thing and see what happens.

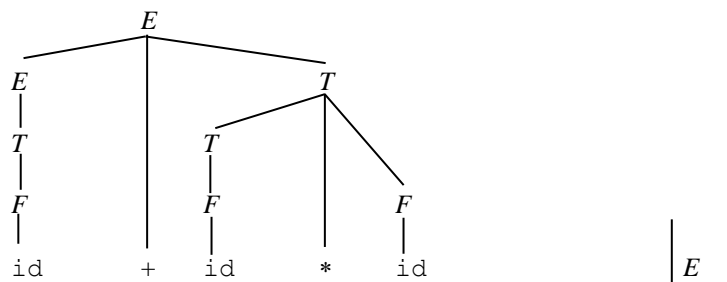
As we said, what we want to do here is not to reduce but instead to shift $*$ onto the stack. Once we do that, the stack will contain $* T + E$. At this point, there are no available reductions (since there are no rules whose right-hand side contains $*$ as the last symbol), so we shift the next symbol, resulting in the stack $id * T + E$. Clearly we next have to reduce id to F (following the same argument that we used above), so we've got:



(Step 5) Next, we need to reduce (since there aren't any more input symbols to shift), but now we have another decision to make: should we reduce the top F to T , using rule (4), or should we reduce the top three symbols to T , using rule (3)? The right answer is to use rule (3), producing:



(Step 6) Finally, we need to apply rule (1), to produce the single symbol E on the top of the stack, and the parse tree:



The job of a shift-reduce parser is complete once it has:

- built a parse tree that spans the entire input string, and
- produced a stack that contains just the start symbol.

So we are done, although we'll discuss below extending the input to include an end-of-input symbol $\$$. In that case, we will usually have a final rule that consumes the $\$$ and pops the start symbol from the stack.

Now let's return to the question of how we can build a parser that makes the right choices at each step of the parsing process. As we walked through the example parse above, there were two kinds of decisions that we had to make:

- whether to shift or reduce (we'll call these *shift-reduce conflicts*), and
- which of several available reductions to perform (we'll call these *reduce-reduce conflicts*).

Let's focus first on shift-reduce conflicts. At least in this example, it was always possible to make the right decision on these conflicts if we had two kinds of information:

- The symbol that is currently on the top of the stack, coupled with a good understanding of what is going on in the grammar. For example, we noted that there's nothing to be done with a raw `id` that hasn't been reduced to an `F`.
- A peek at the next input symbol (the one that we're considering shifting), which we call the lookahead symbol. For example, when we were trying to decide whether to reduce $T + E$ or shift on the next symbol, we looked ahead and saw that the next symbol was `*`. Since we know that `*` has higher precedence than `+`, we knew not to reduce `+`, but rather to wait and deal with `*` first. In order to guarantee that there always is a lookahead symbol, even after the last real input symbol has been read, we'll assume from now on that every string ends with `$`, a special end-of-input symbol.

If the decision about whether to shift or to reduce is dependent only on the current top of stack symbol and the current lookahead symbol, then we can define a procedure for resolving shift-reduce conflicts by specifying a precedence relation $P \subseteq V \times \{\Sigma \cup \$\}$. P will contain the pair (s, c) iff, whenever the top of stack symbol is s and the lookahead symbol is c , the parser should reduce. If the current situation is described by a pair that is not in P , then the parser will shift the lookahead symbol onto the stack.

An easy way to encode a precedence relation is as a table, which we'll call a *precedence table*. As an example, consider the following precedence table for our arithmetic expression grammar, augmented with `$`:

	()	id	+	*	\$
(
)		R		R	R	R
id		R		R	R	R
+						
*						
E						
T		R		R		R
F		R		R	R	R

This table should be read as follows: Compare the left-most column to the top of the stack and find the row that matches. Now compare the symbols along the top of the chart to the lookahead symbol and find the column that matches. If there's an **R** in the corresponding square of the table, then reduce. Otherwise, shift.

Let's now go back to the problem of parsing our example input string, `id + id * id`. Remember that we had a shift/reduce conflict at step 4, when the stack's contents were $T + E$ and the next input symbol was `*`. We can now resolve that conflict by checking the precedence table. We look at the next to the last row of the table, the one that has T as the top of stack symbol. Then we look at the column headed `*`. There's no **R**, so we don't reduce. But notice that if the lookahead symbol had been `+`, we'd have found an **R**, telling us to reduce, which is exactly what we'd want to do. Thus this table captures the precedence relationships between the operators `*` and `+`, plus the fact that we want to associate left when faced with operators of equal precedence.

Now consider the problem of resolving reduce-reduce conflicts. Here's a simple strategy called the *longest-prefix heuristic*: Given a choice of right-hand sides that match the current stack, choose the longest one.

Returning to our example parse, we encountered a reduce-reduce conflict at step 5. The longest-prefix heuristic tells us to reduce $F * T$ rather than just F , which is the right thing to do.

15.3.4 Deterministic, Bottom-UP LR Parsing


There is a large and very useful class of languages for which it is possible to build a deterministic, bottom-up parser by extending the notion of a precedence table so that it includes even more information about paths that will eventually succeed versus those that will eventually fail. We'll call the resulting table a *parse table*.


We define a grammar G to be **LR(k)**, for any positive integer k , iff it is possible to build a deterministic parser for G that scans its input left to right (thus the L in the name) and, for any input string in $L(G)$, builds a rightmost derivation (thus the R in the name), looking ahead at most k symbols. We define a language to be LR(k) iff there exists an LR(k) grammar for it. We'll state here, without proof, two important facts about the LR(k) languages:

- The class of LR(k) languages is exactly the class of deterministic context-free languages, as defined in Section 13.5.
- If a language is LR(k), for some k , then it is also LR(1).

Given an LR(1) grammar, it is possible to build a parse table that can serve as the basis for a deterministic shift-reduce parser. The parse table, like the precedence table we built in the last section, tells the parser when to shift and when to reduce. It also tells it how to resolve reduce-reduce conflicts. Unfortunately, for many LR(1) languages, the parse table is too large to be practical.

But there is a technique, called LALR (lookahead LR) parsing, that works on a restricted class of LR(1) grammars. LALR parsers are deterministic, shift-reduce parsers. They are widely used for a combination of three important reasons:

- Most practical languages can be described by an LALR grammar.
- The parse tables that are required by an LALR parser are reasonably small.
- There exist powerful tools  to build those tables. So efficient parsers are very easy to build.

This last point is key. While it is possible to build parse tables for top-down LL parsers by hand, it isn't possible, for any but the simplest grammars, to build LALR parse tables by hand. As a result, bottom-up parsing was not widely used until the development of parser-generation tools. The most influential such tool has been Yacc , which is designed to work together with Lex (described briefly in Section 15.1) to build a combined lexical analyzer/parser. There have been many implementations of Yacc and it has many descendants.

15.4 Parsing Natural Languages

Programming languages are artificial. They are designed by human designers, who are free to change them so that they possess various desirable properties, including parsability. But now consider English or Spanish or Chinese or Swahili. These languages are natural. They have evolved to serve a purpose, but that purpose is communication among people. The need to build programs to analyze them, index them, retrieve them, translate them, and so forth, has been added very late in the game. It should therefore come as little surprise that the efficient parsing techniques that we have described in the last two sections do not work as well for natural languages as they do for artificial ones.

15.4.1 The Problems

Parsers for natural languages must face at least five problems that are substantially more severe in the case of natural languages than they are for artificial ones:

- **Ambiguity:** There do not exist unambiguous grammars with the power to generate all the parse trees that correspond to the meanings of sentences in the language. Many sentences are syntactically ambiguous. (Recall Example 11.22.) Choosing the correct parse tree for a sentence generally requires appeal to facts about the larger context in which the sentence occurs and facts about what makes sense. Those facts can be encoded in separate functions that choose from among a set of parse trees or partial parse trees. Or they may be encoded probabilistically in a stochastic grammar, as described in Section 11.10. Even when the information required to make a choice is available in the input string, it may be many words away, so the single symbol lookahead that we used in LL and LR parsing is rarely adequate.
- **Gaps:** In the sentence, *What did Jen eat?*, the word *What* is the object of the verb *eat* but it is not near it in the sentence. See [C 750](#) for a discussion of this issue.
- **Dialect:** English is not one language. It is hundreds at least. Chinese is worse. There is no ISO standard for English or Chinese. So what language should we build a grammar for?

- Evolution: Natural languages change as they are used. The sentences, *You wanted to do that why?* and *They' re open 24/7* are fine American English sentences today. But they wouldn't have been twenty years ago.
- Errors: Even among speakers who agree completely on how they ought to talk, what they actually say is a different story. While it is acceptable (and even desirable) for a compiler to throw out syntactically ill-formed programs, imagine the usefulness of a translating telephone that objected to every sentence that stopped in the middle, started over, and got a pronoun wrong. Parsers for natural languages must be robust in a way that parsers for artificial languages are not required to be.

In addition, natural languages share with many artificial languages the problem of checking for agreement between various constituents:

- For programming languages, it is necessary to check variable declarations against uses.
- For natural languages, it is necessary to check for agreement between subject and verb, for agreement between nouns and modifiers (in languages like Spanish), and so forth.

In \mathbb{C} 666, we prove that one typical programming language, Java, is not context-free because of the requirement that variables be declared before they are used. So parsers for programming languages exploit additional mechanisms, such as symbol tables, to check such features. In \mathbb{C} 747, we address the question of whether natural languages such as English are formally context-free. There are no proofs, consistent with the empirical facts about how people actually talk, that English is not context-free. There is, on the other hand, a proof that one grammatical feature of one natural language, Swiss German, is not context-free. But, even for English, it is more straightforward to describe agreement features, just as we do for Java, with additional mechanisms that check agreement features before context-free rules can be applied during parsing.

15.4.2 The Earley Algorithm

Despite the problems we just described, context-free parsers, augmented as necessary, are widely used in natural language processing systems. Although efficient parsing is important, deterministic parsing is generally not possible. So LL and LR techniques won't work. And it is usually not acceptable to require grammars to be in Chomsky normal form (because parsers based on such grammars will not generate natural parse trees). So the CKY algorithm can't be used. The Earley algorithm, which we present next, is a reasonably efficient ($\mathcal{O}(n^3)$) algorithm that works with an arbitrary context-free grammar. Because of its importance in natural language processing, we'll describe this algorithm as it operates on English sentences. But it can be applied to any context-free language. So, for example, to imagine it being used in a compiler, substitute the term "token" each time we mention a "word" here.

The Earley algorithm, like the CKY algorithm, is a dynamic programming technique. It works top-down but, unlike the simple depth-first search algorithm that we discussed in Section 15.2.1, it builds each potential constituent only once (and then may reuse smaller constituents as it explores alternative ways to build larger ones). The structure that is used to record the constituents as they are found can be thought of as a simple chart. So parsers that are based on the Earley algorithm, and on others that are related to it, are often called *chart parsers*.

To describe the way that the Earley algorithm works, we introduce the dot notation, which we will use to indicate the progress that the parser has made so far in matching the right-hand side of a grammar rule against the input. Let:

- $A \rightarrow \alpha \bullet \beta \gamma$ describe an attempt to apply the rule $A \rightarrow \alpha \beta \gamma$, where everything before the \bullet has already matched against the input and the parser is still trying match everything after the \bullet .
- $A \rightarrow \bullet \alpha \beta \gamma$ describe a similar attempt except that nothing has yet matched against the input.
- $A \rightarrow \alpha \beta \gamma \bullet$ describe a similar attempt except that the entire right-hand side (and thus also A) has matched against the input.

The overall progress of the parsing process can be described by listing each rule that is currently being attempted and indicating, for each:

- Where in the sentence the parser is trying to match the right hand side, and
- How much progress (as indicated by the position of the dot) has been made in doing that matching.

All of this information can be summarized in a chart with $n+1$ rows, where n is the number of words in the input string. In creating the chart, we won't assign indices to the words of the input string. Instead we'll assign the indices to the points in between the words. So, for example, we might have:

0 Jen 1 saw 2 Bill 3

We'll let row i of the chart contain every instance of an attempt to match a rule whose \bullet is in position i . The easiest way to envision the chart is to imagine that it also has $n+1$ columns, which will correspond to the location in the input at which the partial match began. We'll reverse our usual convention and list the column index first so that the pair describes the start and then the end of a partial match. So associating the indices $[i, j]$ with a rule $A \rightarrow \alpha\bullet\beta\gamma$ means that the parser began matching α in position i and the \bullet is currently in position j .

The Earley algorithm works top-down. So it starts by inserting into the chart every rule whose left-hand side is the start symbol of the grammar. The indices associated with each such rule are $[0, 0]$, since the parser must try to match the right-hand side starting before the first word (i.e., in position 0) and it has so far matched nothing. The job of the parser is to find a match, for the right-hand side of at least one of those rules, that spans the entire sentence. In other words, for at least one of those initial rules, the \bullet must move all the way to the right and the index pair $[0, n]$, indicating a match starting before the first word and ending after the last one, must be assigned to the rule.

To see how the algorithm works, we'll trace its operation on the simple sentence we showed above, given the following grammar:

$S \rightarrow NP VP$
 $NP \rightarrow ProperNoun$
 $VP \rightarrow V NP$

After initialization, the chart will be:

3	
2	
1	
0	$S \rightarrow \bullet NP VP [0, 0]$

0 Jen 1 saw 2 Bill 3

Next, the algorithm predicts that an NP must occur, starting at position 0. So it looks for rules that tell it how to construct such an NP . It finds one, and adds it to the chart, giving:

3	
2	
1	
0	$NP \rightarrow \bullet ProperNoun [0, 0]$ $S \rightarrow \bullet NP VP [0, 0]$

0 Jen 1 saw 2 Bill 3

Now it predicts the existence of a *ProperNoun* that starts in position 0. It isn't generally practical to handle part of speech tags like *ProperNoun* by writing a rule like $ProperNoun \rightarrow Jen | Bill | Chris | \dots$. Instead, we'll assume that the input has already been tagged with part of speech markers like *Noun*, *ProperNoun*, *Verb*, and so forth. (See § 741 for a discussion of how this is process, called part of speech or POS tagging is done.) So, whenever the next symbol the parser is looking for is a part of speech tag, it will simply check the next input symbol and see whether it

has the required tag. If it does, a match has occurred and the parser will behave as though it just matched the implied rule. If it does not, then no match has been found and the rule can make no progress. In this case, *Jen* is a *ProperNoun*, so there is a match. The parser can apply the implied rule $ProperNoun \rightarrow Jen$. Notice that whenever the parser actually matches against the input, the \bullet moves. So the parser adds this new rule to the next row of the chart, which now becomes:

3	
2	
1	$ProperNoun \longrightarrow \rightarrow Jen \bullet [0, 1]$
0	$NP \rightarrow \bullet ProperNoun [0, 0]$ $S \rightarrow \bullet NP VP [0, 0]$

$ProperNoun$ V, N $ProperNoun$
 0 *Jen* 1 *saw* 2 *Bill* 3

The parser has now finished considering both of the rules in row 0, so it moves on to row 1. It notices that it has found a complete *ProperNoun*. Whenever it finds a complete constituent, it must look back to see what rules predicted the occurrence of that constituent (and thus are waiting for it). The new, complete constituent starts at position 0, so the parser looks for rules whose \bullet is in position 0, indicating that they are waiting for a constituent that starts there. So it looks back in row 0. It finds that the *NP* rule is waiting for a *ProperNoun*. Since a *ProperNoun* has just been found, the parser can create the rule $NP \rightarrow ProperNoun \bullet [0, 1]$ and add it to row 1. Then it looks at that rule and realizes that it has found a complete *NP* starting in position 0. So it looks back in row 0 again, this time to see what rule is waiting for an *NP*. It finds that *S* is, so it creates the rule $S \rightarrow NP \bullet VP [0, 1]$. At this point, the chart looks like this (using \checkmark to mark rules that have already been processed):

3	
2	
1	$S \longrightarrow \rightarrow NP \bullet VP [0, 1]$ $\checkmark NP \longrightarrow \rightarrow ProperNoun \bullet [0, 1]$ $\checkmark ProperNoun \longrightarrow \rightarrow Jen \bullet [0, 1]$
0	$\checkmark NP \rightarrow \bullet ProperNoun [0, 0]$ $\checkmark S \rightarrow \bullet NP VP [0, 0]$

$ProperNoun$ V, N $ProperNoun$
 0 *Jen* 1 *saw* 2 *Bill* 3

The remaining unprocessed rule tells the parser that it needs to predict again. It needs to find a *VP* starting in position 1. Because no progress has been made in finding a *VP*, any rule that could describe one will have its \bullet still in position 1. So the parser adds the rule $VP \rightarrow \bullet V NP [1, 1]$ to row 1 of the chart. At this point, the chart will be:

3	
2	
1	$VP \rightarrow \bullet V NP [1, 1]$ $\checkmark S \longrightarrow \rightarrow NP \bullet VP [0, 1]$ $\checkmark NP \longrightarrow \rightarrow ProperNoun \bullet [0, 1]$ $\checkmark ProperNoun \longrightarrow \rightarrow Jen \bullet [0, 1]$
0	$\checkmark NP \rightarrow \bullet ProperNoun [0, 0]$ $\checkmark S \rightarrow \bullet NP VP [0, 0]$

$ProperNoun$ V, N $ProperNoun$
 0 *Jen* 1 *saw* 2 *Bill* 3

In processing the next rule, the parser notices that the predicted symbol is a part of speech, so it checks the next input word to see if it can be a Verb. *saw* has been tagged as a possible Verb or a possible Noun. So a new rule is added, this time to row 2 since the • moves to the right to indicate that the match has moved one word farther in the input. Notice that because *Earleyparse* works top-down, it will ignore part of speech tags (such as *saw* as a Noun) that don't fit in the larger sentence context. The chart is now:

3		
2		$V \longrightarrow \text{saw} \bullet [1, 2]$
1	✓	$VP \rightarrow \bullet VNP [1, 1]$
	✓	$S \longrightarrow \text{NP} \bullet VP [0, 1]$
	✓	$NP \longrightarrow \text{ProperNoun} \bullet [0, 1]$
	✓	$\text{ProperNoun} \longrightarrow \text{Jen} \bullet [0, 1]$
0	✓	$NP \rightarrow \bullet \text{ProperNoun} [0, 0]$
	✓	$S \rightarrow \bullet NP VP [0, 0]$

ProperNoun
V, N
ProperNoun
 0 Jen 1 saw 2 Bill 3

Having found a complete constituent (the *V*), starting in position 1, the parser looks back to row 1 to find rules that are waiting for it. It finds one: $VP \rightarrow \bullet VNP [1, 1]$. So it can advance this rule's • and create the rule $VP \rightarrow V \bullet NP [1, 2]$, which can be added to row 2. That rule will be processed next. It will predict the existence of an *NP* starting in position 2, so the parser will create rules that describe the possible structures for such an *NP*. Our simple grammar has only one *NP* rule, so the parser will create the rule $NP \rightarrow \bullet \text{ProperNoun} [2, 2]$ and add it to the chart in row 2. Next the parser looks for, and finds, a *ProperNoun*, *Bill*, starting in position 2 and ending at position 3. So it enters it in row 3. At this point, the chart will be:

3		$\text{ProperNoun} \longrightarrow \text{Bill} \bullet [2, 3]$
2	✓	$NP \rightarrow \bullet \text{ProperNoun} [2, 2]$
	✓	$VP \longrightarrow \text{V} \bullet NP [1, 2]$
	✓	$V \longrightarrow \text{saw} \bullet [1, 2]$
1	✓	$VP \rightarrow \bullet VNP [1, 1]$
	✓	$S \longrightarrow \text{NP} \bullet VP [0, 1]$
	✓	$NP \longrightarrow \text{ProperNoun} \bullet [0, 1]$
	✓	$\text{ProperNoun} \longrightarrow \text{Jen} \bullet [0, 1]$
0	✓	$NP \rightarrow \bullet \text{ProperNoun} [0, 0]$
	✓	$S \rightarrow \bullet NP VP [0, 0]$

ProperNoun
V, N
ProperNoun
 0 Jen 1 saw 2 Bill 3

Having found a complete constituent (the *ProperNoun*), starting in position 2, the parser looks in row 2 to find rules that are waiting for a *ProperNoun* starting in position 2. It finds one: $NP \rightarrow \bullet \text{ProperNoun} [2, 2]$. It can advance that rule's • and add the rule $NP \rightarrow \text{ProperNoun} \bullet [2, 3]$ to row 3. This rule tells the parser that another complete constituent, an *NP*, has been found, starting in position 2. So it again looks back to row 2 and finds that the rule $VP \rightarrow V \bullet NP [1, 2]$ is looking for that *NP*. So its • can be advanced, and the rule $VP \rightarrow VNP \bullet [1, 3]$ can be added to row 3. That rule describes yet another complete constituent, a *VP*, starting back in position 1. So the parser looks back at row 1 to find a rule that is waiting for that *VP*. It finds $S \rightarrow NP \bullet VP [0, 1]$. So its • can be advanced, and the rule $S \rightarrow NP VP \bullet [0, 3]$ can be added to row 3. Now the chart is:

3	✓	$S \longrightarrow NP VP \bullet [0, 3]$
	✓	$VP \longrightarrow V NP \bullet [1, 3]$
	✓	$NP \longrightarrow ProperNoun \bullet [2, 3]$
	✓	$ProperNoun \longrightarrow Bill \bullet [2, 3]$
2	✓	$NP \rightarrow \bullet ProperNoun [2, 2]$
	✓	$VP \longrightarrow V \bullet NP [1, 2]$
	✓	$V \longrightarrow saw \bullet [1, 2]$
1	✓	$VP \rightarrow \bullet V NP [1, 1]$
	✓	$S \longrightarrow NP \bullet VP [0, 1]$
	✓	$NP \longrightarrow ProperNoun \bullet [0, 1]$
	✓	$ProperNoun \longrightarrow Jen \bullet [0, 1]$
0	✓	$NP \rightarrow \bullet ProperNoun [0, 0]$
	✓	$S \rightarrow \bullet NP VP [0, 0]$

0 *ProperNoun* 1 *V, N* 2 *ProperNoun* 3
 Jen *saw* *Bill*

At this point, the parsing process halts. A complete S that spans the entire input has been found. In this simple example, there is only one parse. Given a more complex sentence and a more realistic grammar, there could be several parses. If we want to find them all, the parser can be allowed to continue until no new edges can be added.

We can now state the algorithm that we have just described:

Earleyparse(w : input string containing n words, G : context-free grammar) =

1. For every rule in G of the form $S \rightarrow \alpha$, where S is the start symbol of G , do: /* Initialize *chart*.
 $insert(chart, S \rightarrow \bullet \alpha [0, 0])$. /* Insert the rule $S \rightarrow \bullet \alpha [0, 0]$ into row 0 of *chart*.
2. For $i = 0$ to n do: /* Go through the rows one at a time.
 For each rule r in row $_i$ of *chart* do:
 If r corresponds to finding a complete constituent, then *extendothers*(*chart*, r).
 Else if the symbol after the \bullet of r is a part of speech tag, then *scaninput*(w , *chart*, r).
 Else *predict*(*chart*, r).

insert(*chart*, $r [j, k]$): rule that spans from j to k in *chart* =

If r is not already on *chart*, spanning from j to k , then add it in row k .

extendothers(*chart*: *chart*, $r [j, k]$: rule of the form $A \rightarrow \alpha \bullet$ that spans from j to k in *chart*) =

For each rule p of the form $X \rightarrow \beta \bullet A \gamma [i, j]$ on *chart* do: /* Find rules waiting for A starting at j .
 $insert(chart, X \rightarrow \beta A \bullet \gamma [i, k])$. /* Move the \bullet one symbol to the right and add rule to row $_k$

scaninput(w : input string, *chart*: *chart*, $r [j, k]$: rule of the form $A \rightarrow \beta \bullet A \gamma$, where A is a part of speech tag, and the rule spans from j to k in *chart*) =

If w_k (the k^{th} word of the input) has been labeled with the tag A then:

$insert(chart, A \rightarrow w_k \bullet [k, k+1])$. /* Add this one to the next row.

predict(*chart*: *chart*, $r [j, k]$: rule of the form $A \rightarrow \alpha \bullet B \beta$ that spans from j to k in *chart*) =

For each rule in G of the form $B \rightarrow \gamma$ do:

$insert(chart, B \rightarrow \bullet \gamma [k, k])$. /* Try to find a B starting at k .

As we have presented it, *Earleyparse* doesn't actually build a parse tree. It simply decides whether a parse exists. But it is straightforward to modify it so that the parse tree(s) that correspond to successful S rules can be extracted.

Notice that *Earleyparse* avoids the two major pitfalls of the more straightforward top-down parsing algorithm that exploits simple depth-first search. First, we observe that it will always halt, even if provided with a grammar that contains left-recursive rules. This must be true because a rule cannot be added to the chart at a given location more

than once. Since there is a finite number of rules and a finite number of locations in the chart, only a finite number of rules can be placed on the chart and *Earleyparse* terminates after it has processed each of them.

Second, we observe that *Earleyparse* avoids the wasted effort of backtracking search. Instead it reuses constituents. How it does so may not have been obvious in the very simple example that we just considered. But suppose that we added to our grammar a few more *NP* rules, the necessary prepositional phrase (*PP*) rules and the rule:

$$VP \rightarrow VP PP$$

Now suppose that we try to parse the sentence `Jen saw Bill through the window`. A backtracking, top-down parser would try the $VP \rightarrow V NP$ rule first (assuming it was listed first). It would build an *S* using that *VP* and then realize that the *S* didn't span the entire sentence. So it would back up and throw away all the work it did to build the *VP*, including building the *NP* that dominates `Bill`. In this simple example, that *NP* doesn't represent a lot of work, but in a less trivial sentence, it might. Then the parser would start over to build a *VP* using the new rule that allows for a prepositional phrase. *Earleyparse*, on the other hand, will build each of those constituents once. Since rules are never removed from the chart, they can be reused as necessary by other, higher-level rules. We leave working out the details of this example as an exercise.

We can analyze the complexity of *Earleyparse* as follows: The loop of step 2 is executed n times. The inner loop is executed once for each rule that is already in the row. There are $\mathcal{O}(n)$ of them. Whenever *extendothers* is called, it must compare its edge to all the other edges in the row. And there are $\mathcal{O}(n)$ of them. Multiplying these together, we get that the total number of steps is $\mathcal{O}(n^3)$. If we want to consider the size of G , then let $|G|$ be the number of rules in G . The total number of steps executed by *Earleyparse* becomes $\mathcal{O}(n^3 \cdot |G|^2)$.

15.5 Exercises

- 1) Consider the following grammar that we presented in Example 15.9:

$$\begin{aligned} S &\rightarrow AB\$ \mid AC\$ \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid b \\ C &\rightarrow c \end{aligned}$$

Show an equivalent grammar that is LL(1) and prove that it is.

- 2) Assume the grammar:

$$\begin{aligned} S &\rightarrow NP VP \\ NP &\rightarrow ProperNoun \\ NP &\rightarrow Det N \\ VP &\rightarrow V NP \\ VP &\rightarrow VP PP \\ PP &\rightarrow Prep NP \end{aligned}$$

Assume that `Jen` and `Bill` have been tagged *ProperNoun*, `saw` has been tagged *V*, `through` has been tagged *Prep*, `the` has been tagged *Det*, and `window` has been tagged *N*. Trace the execution of *Earleyparse* on the input sentence `Jen saw Bill through the window`.

- 3) Trace the execution of *Earleyparse* given the string and grammar of Example 15.5.
- 4) Trace the execution of a CKY parser on the input string `id + id * id`, given the unambiguous arithmetic expression grammar shown in Example 11.19, by:
- Converting the grammar to Chomsky normal form.
 - Showing the steps of the parser.

16 Summary and References

The theory of context-free languages is not as tidy as the theory of regular languages. Interesting subsets, including the deterministic context-free languages and the context-free languages that are not inherently ambiguous, can be shown to be only proper subsets of the larger class of context-free languages. The context-free languages are not closed under many common operations. There is no algorithm for minimizing PDAs. There is no fast recognition algorithm that works for arbitrary context-free languages. There are no decision procedures for many important questions. Yet substantial effort has been invested in studying the context-free languages because they are useful. The results that we have presented here have been developed by many people, including theoreticians (who were particularly interested in the formal properties of the set), linguists (who were interested in modeling natural languages and who found context-free grammars to be a useful tool), and compiler writers (who were interested in building efficient parsers for programming languages). The theory that was developed out of the confluence of those efforts continues to provide the basis for practical parsing systems today.

Table16.1 summarizes the properties of the context-free languages and compares them to the regular languages:

	<i>Regular</i>	<i>Context-Free</i>
<i>Automaton</i>	FSM	PDA
<i>Grammar(s)</i>	Regular grammar Regular expressions	Context-free grammar
<i>ND = D?</i>	Yes	No
<i>Closed under:</i>		
<i>Concatenation</i>	Yes	Yes
<i>Union</i>	Yes	Yes
<i>Kleene star</i>	Yes	Yes
<i>Complement</i>	Yes	No
<i>Intersection</i>	Yes	No
\cap with Regular	Yes	Yes
<i>Decidable:</i>		
<i>Membership</i>	Yes	Yes
<i>Emptiness</i>	Yes	Yes
<i>Finiteness</i>	Yes	Yes
$= \Sigma^*$	Yes	No
<i>Equivalence</i>	Yes	No

Table16.1 Comparing the regular and the context-free languages

References

The context-free grammar formalism grew out of the efforts of linguists to describe the structure of sentences in natural languages such as English. By the mid 1940's, it was widely understood that sentences could be described hierarchically, with a relatively small number of immediate constituents (or ICs) at each level. For example, many English sentences can be described as a noun phrase followed by a verb phrase. Each such IC, until the smallest ones, could in turn be further described as a set of smaller constituents, and so forth. [Chomsky 1956] introduced phrase structure (production-rule) grammars as a way to describe such a structural analysis of a sentence. In [Chomsky 1959], Chomsky defined a four-level hierarchy of language classes based on the form of the grammar rules that are allowed. Context-free grammars, in the sense in which we have defined them, with their particular restrictions on the form of the rules, were described there. We'll say more about the Chomsky hierarchy in Section **Error! Reference source not found.**

Chomsky normal form was also introduced in [Chomsky 1959]. Greibach normal form was introduced in [Greibach 1965].

Island grammars are described in [Moonen 2001]. There are many other applications of them as well \square . Two early uses of the related idea, island parsing, are described in [Carroll 1983] and [Stock, Falcone and Insinnamo 1988]. For a discussion of stochastic (probabilistic) context-free grammars and parsing techniques, see [Jurafsky and Martin 2000].

The idea of using a pushdown stack to process naturally recursive structures, like formulas in logic or in programming languages, was developed independently by many people in the 1950s. (For a brief discussion of this history, as well as many other aspects of the theory of context-free languages, see [Greibach 1981].) The pushdown automaton was described both in [Oettinger 1961] (where it was used for the syntactic analysis of natural language sentences as part of a machine translation system) and in [Schutzenberger 1963]. The proof of the equivalence of context-free grammars and PDAs appeared independently in [Evey 1963], [Chomsky 1962] and [Schutzenberger 1963].

Many key properties of context-free languages, including the Pumping Theorem and the fact that the context-free languages are closed under intersection with the regular languages, were described in [Bar-Hillel, Perles and Shamir 1961]. Our claim that, if a grammar G is not self-embedding then $L(G)$ is regular, is proved in [Hopcroft and Ullman 1969]. The fact that the context-free languages are closed under union but not under intersection and complement was shown in [Scheinberg 1960]. Ogden's Lemma appeared in [Ogden 1968]. Parikh's Theorem was presented in [Parikh 1966]. The fact that every context-free language over a single character alphabet is regular is from [Ginsburg and Rice 1962]. For a comprehensive treatment of the mathematical theory of context-free languages, see [Ginsburg 1966].

[Parikh 1966] proved that there exist inherently ambiguous context-free languages. It showed that $\{a^i b^j a^k b^l : i, j, k, l \geq 0, i = k \text{ or } j = l\}$ is inherently ambiguous. The claim that $\{a^i b^j c^k : i, j, k \geq 0, i = j \text{ or } j = k\}$ is inherently ambiguous is proved in [Harrison 1978] and in [Du and Ko 2001], using Ogden's Lemma.

The proof in Example 11.10 is taken from [Martin 2003].

See [Aho, Sethi and Ullman 1986] for a more detailed treatment of the issues that arise in parsing artificial context-free languages, as well as a good survey of the history of the development of parsing techniques, including the definition of $LL(k)$ and $LR(k)$ languages. In particular, algorithms for computing *first* and *follow* and for eliminating all left recursion are presented there. *Lex* is described in [Lesk and Schmidt 1979]. *Yacc* is described in [Johnson 1979]. The CKY algorithm was independently discovered by J. Cocke, Daniel Younger, and Tadao Kasami, and described in [Younger 1967] and [Kasami 1965]. [Valiant 1975] showed that context-free parsing could be recast as matrix multiplication. Strassen's algorithm for fast matrix multiplication was described in [Strassen 1969]. The $\mathcal{O}(n^{2.376})$ matrix multiplication algorithm was presented in [Coppersmith and Winograd 1990]. The claim that if there were a fast context-free parsing algorithm there would be a fast algorithm for Boolean matrix multiplication is proved in [Lee 2002]. The claims we made about the $LR(k)$ languages are proved in [Hopcroft and Ullman 1969].

Earley's algorithm for general top-down parsing was first presented in [Earley 1970]. The version given here is patterned after the one in [Jurafsky and Martin 2000].