FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU, Currently for CSE – Computer Science Engineering...

Join Telegram to get Instant Updates: https://bit.ly/2GKiHnJ

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

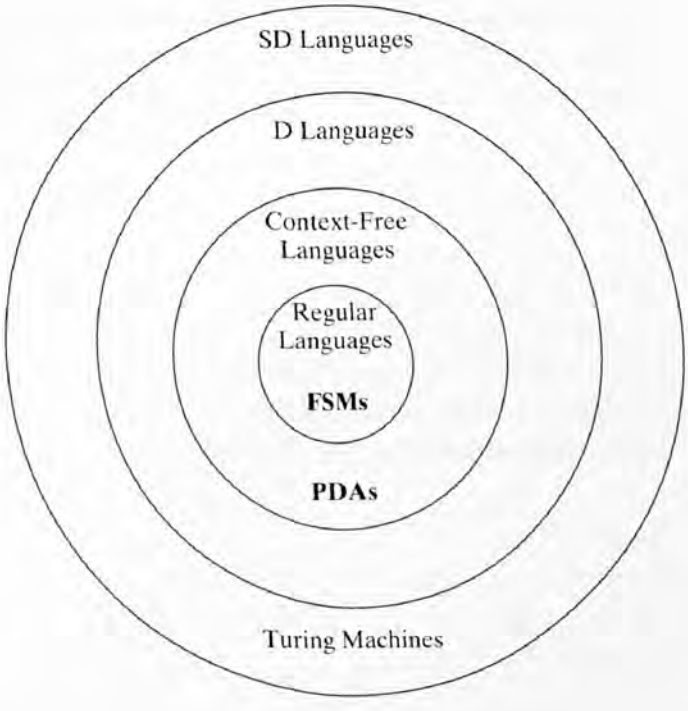
PART III: CONTEXT-FREE LANGUAGES AND PUSHDOWN AUTOMATA 201

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.

CHAPTER 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:

$$S \rightarrow \mathrm{a}S\mathrm{b}$$
$$\mathrm{a}S \rightarrow \varepsilon$$
$$\mathrm{a}S\mathrm{b} \rightarrow \mathrm{b}S\mathrm{ab}S\mathrm{a}$$

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, (M.3.3) 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 (M.3.4) and as the basis for reasoning about the behavior of nonplayer characters in computer games. (N.3.3)

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. (Q.2.2)

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. (Appendix O)

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. Rewrite systems can also 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 (M.2.3) 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 (L.3), programming languages like Java (G.1), music (N.1), dance (Q.2.1), the growth of living organisms (Q.2.2), and the structure of RNA. (K.4)

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 (Q.1.3) 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 $S \rightarrow \varepsilon$. Then a derivation could begin with:

$$S \Rightarrow \text{a}S\text{b} \Rightarrow \text{aa}S\text{bb} \Rightarrow \ldots$$

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:

$S \Rightarrow \text{a}S\text{b} \Rightarrow \text{aa}S\text{bb} \Rightarrow \text{aaa}S\text{bbb}$ (using the first rule),
$S \Rightarrow \text{a}S\text{b} \Rightarrow \text{aa}S\text{bb} \Rightarrow \text{aab}S\text{abb}$ (using the second rule), and
$S \Rightarrow \text{a}S\text{b} \Rightarrow \text{aa}S\text{bb} \Rightarrow \text{aabb}$ (using the third rule).

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 there is no match with the left-hand side of any rule in the grammar. For example, if the working string were $A\text{a}B\text{b}$, this would happen if the only left-hand side were C.

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 nonterminated 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 $S \rightarrow B\mathrm{a}$ and $B \rightarrow \mathrm{b}B$, with S the start symbol. Then all derivations proceed in the following way:

$$S \Rightarrow B\mathrm{a} \Rightarrow \mathrm{b}B\mathrm{a} \Rightarrow \mathrm{bb}B\mathrm{a} \Rightarrow \mathrm{bbb}B\mathrm{a} \Rightarrow \mathrm{bbbb}B\mathrm{a} \Rightarrow \cdots$$

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 $\varnothing$.

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):

$$S \rightarrow \mathrm{a}S\mathrm{b}$$
$$S \rightarrow \varepsilon$$
$$T \rightarrow T$$
$$S \rightarrow \mathrm{a}S\mathrm{bb}TT$$

The following are not allowable context-free grammar rules:

$$ST \rightarrow \mathrm{a}S\mathrm{b}$$
$$\mathrm{a} \rightarrow \mathrm{a}S\mathrm{b}$$
$$\varepsilon \rightarrow \mathrm{a}$$

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 a$Sa \rightarrow aTa$ 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 Appendix G.

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 $V - \Sigma$.

Given a grammar G, define $x \Rightarrow_G y$ (abbreviated $\Rightarrow$ when G is clear from context) to be the binary relation *derives-in-one-step*, defined so that:

$$\forall x, y \in V^* (x \Rightarrow_G y \text{ iff } x = \alpha A \beta, y = \alpha \gamma \beta, \text{ and there exists a rule } A \rightarrow \gamma \text{ in } R_G).$$

Any sequence of the form $w_0 \Rightarrow_G w_1 \Rightarrow_G w_2 \Rightarrow_G \ldots \Rightarrow_G w_n$ is called a ***derivation*** in G. Let $\Rightarrow_G^*$ be the reflexive, transitive closure of $\Rightarrow_G$. We'll call $\Rightarrow_G^*$ the ***derives*** relation.

The ***language generated by G***, denoted $L(G)$, is $\{w \in \Sigma^* : S \Rightarrow_G^* w\}$. In other words, the language generated by G is the set of all strings of terminals that can be derived from S using zero or more applications of rules in G.

A language L is ***context-free*** iff it is generated by some context-free grammar G. The context-free languages (or CFLs) are a proper superset of the regular languages. In the next several examples, we will see languages that are context-free but not regular. Then, in Chapter 13, we will prove the other part of this claim, namely that every regular language is also context-free.

EXAMPLE 11.1 The Balanced Parentheses Language

Consider Bal $= \{w \in \{), (\}^* :$ the parentheses are balanced$\}$. We showed in Example 8.10 that Bal is not regular. But it is context-free because it can be generated by the grammar $G = \{\{S,), (\}, \{), (\}, R, S)$, where:

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

Some example derivations in G:

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

So, $S \Rightarrow^* ()$ and $S \Rightarrow^* (()())$.

The syntax of Boolean query languages is describable with a context-free grammar. (Q.11)

EXAMPLE 11.2 A^nB^n

Consider $A^nB^n = \{a^nb^n : n \geq 0\}$. We showed in Example 8.8 that A^nB^n 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:

$$R = \{S \rightarrow aSb$$
$$S \rightarrow \varepsilon\}.$$

What is it about context-free grammars that gives them the power to define languages like Bal and A^nB^n?

We can begin answering that question by defining a rule in a grammar G to be ***recursive*** iff it is of the form $X \rightarrow w_1Yw_2$, where $Y \Rightarrow_G^* w_3Xw_4$ and all of w_1, w_2, w_3, and w_4 may be any element of V^*. A grammar is recursive iff it contains at least one recursive rule. For example, the grammar we just presented for *Bal* is recursive because it contains the rule $S \rightarrow (S)$. The grammar we presented for A^nB^n is recursive because it contains the rule $S \rightarrow aSb$. A grammar that contained the rule $S \rightarrow aS$ would also be recursive. So the regular grammar whose rules are $\{S \rightarrow aT, T \rightarrow aW, W \rightarrow aS, W \rightarrow a\}$ is recursive. Recursive rules make it possible for a finite grammar to generate an infinite set of strings.

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 $X \rightarrow w_1Yw_2$, where $Y \Rightarrow_G^* w_3Xw_4$ and both w_1w_3 and w_4w_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^nB^n is self-embedding because it contains the rule $S \rightarrow aSb$. The presence of a rule like $S \rightarrow aS$ does not by itself make a grammar self-embedding. But the rule $S \rightarrow aT$ is self-embedding in any grammar G that also contains the rule $T \rightarrow Sb$, since $S \rightarrow aT$ and $T \Rightarrow_G^* Sb$. Self-embedding grammars are able to define languages like Bal, A^nB^n, and others whose strings must contain pairs of matching regions, often of the form uv^ixy^iz. 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, $G_1 = (\{S, a\}, \{a\}, \{S \rightarrow \varepsilon, S \rightarrow a, S \rightarrow aSa\}, S)$ 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 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:

$$R = \{S \rightarrow aSa$$
$$S \rightarrow bSb$$
$$S \rightarrow \varepsilon\}.$$

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:

$$R = \{S \rightarrow aSb$$
$$S \rightarrow bSa$$
$$S \rightarrow SS$$
$$S \rightarrow \varepsilon\}.$$

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:

$$S \rightarrow aSb|bSa|SS|\varepsilon$$

- 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. (G.1.1)

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. (Q.1.2)

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

EXAMPLE 11.6 (*Continued*)

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:

$S \rightarrow NP\ VP$
$NP \rightarrow$ `the` *Nominal* | `a` *Nominal* | *Nominal* | *ProperNoun* | *NP PP*
Nominal $\rightarrow N \mid Adjs\ N$
$N \rightarrow$ `cat` | `dogs` | `bear` | `girl` | `chocolate` | `rifle`
ProperNoun $\rightarrow$ `Chris` | `Fluffy`
$Adjs \rightarrow Adj\ Adjs \mid Adj$
$Adj \rightarrow$ `young` | `older` | `smart`
$VP \rightarrow V \mid V\ NP \mid VP\ PP$
$V \rightarrow$ `like` | `likes` | `thinks` | `shot` | `smells`
$PP \rightarrow Prep\ NP$
Prep $\rightarrow$ `with`

Is English (or German or Chinese) really context-free? (L.3.3)

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:

$$A \rightarrow BC\ldots$$

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.

- 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.

> The outside-in structure of context-free grammars makes them well suited to describing physical things, like RNA molecules, that fold. (K.4)

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:

$$
\begin{aligned}
R = \{&S \rightarrow NC && \text{/* Generate the two independent portions.}\\
&N \rightarrow aNb && \text{/* Generate the } a^n b^n \text{ portion, from the outside in.}\\
&N \rightarrow \varepsilon\\
&C \rightarrow cC && \text{/* Generate the } c^m \text{ portion.}\\
&C \rightarrow \varepsilon\}.
\end{aligned}
$$

EXAMPLE 11.8 The Kleene Star of a Language

Let $L = \{a^{n_1} b^{n_1} a^{n_2} b^{n_2} \ldots a^{n_k} b^{n_k} : k \geq 0 \text{ and } \forall i\ (n_i \geq 0)\}$. For example, the following strings are in L: ε, abab, aabbaaabbbabab. 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:

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

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:

$$
\begin{aligned}
R = \{&S \rightarrow AB \mid AC\\
&A \rightarrow aAb \mid \varepsilon
\end{aligned}
$$

$B \rightarrow \mathrm{b}A$
$C \rightarrow \mathrm{b}C\mathrm{a}$
$D \rightarrow AB\}$.

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 $L(G) = \varnothing$. 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 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* 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^nB^n Grammar

In Example 11.2, we considered the language A^nB^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:

$$\begin{aligned} R = \{&S \rightarrow aSb \quad (1)\\ &S \rightarrow bSa \quad (2)\\ &S \rightarrow SS \quad (3)\\ &S \rightarrow \varepsilon\}. \quad (4) \end{aligned}$$

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 a 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 we 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.

EXAMPLE 11.10 (*Continued*)

- 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. $\Delta(\text{a}) = 1$. Since $w \in L$, $\Delta(w) = 0$. So $\Delta(\text{a}x) = -1$ (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.

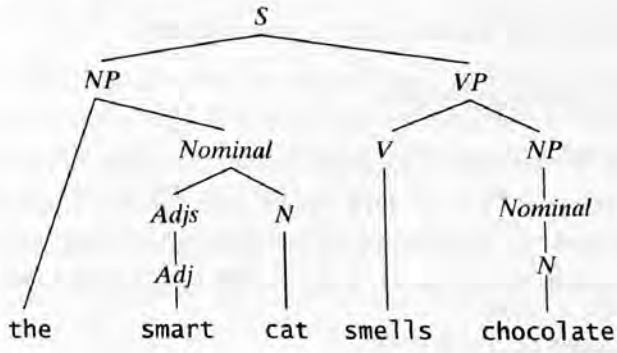
A parse tree, derived by a grammar $G = (V, \Sigma, R, S)$, is a rooted, ordered tree in which:

- 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 $x_1, x_2, \ldots, x_n$, then R contains the rule $X \rightarrow x_1, x_2, \ldots, x_n$.

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:

$$S \Rightarrow NP\ VP \Rightarrow \ldots$$

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

$$S \Rightarrow NP\ VP \Rightarrow \text{The } Nominal\ VP \Rightarrow$$
$$S \Rightarrow NP\ VP \Rightarrow NP\ V\ NP \Rightarrow$$

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:

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

- A right-most derivation is:

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

11.7 Ambiguity

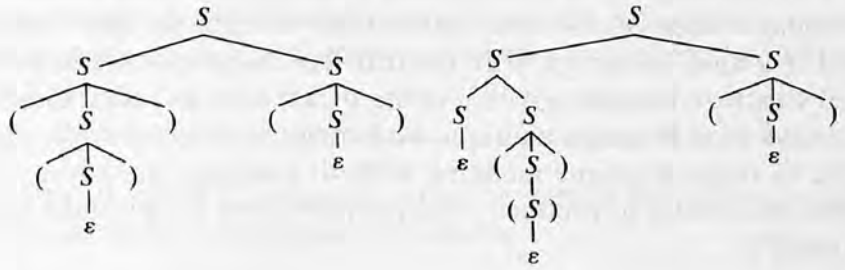
Sometimes a grammar may produce more than one parse tree for some (or all) of the strings it generates. When this happens, we say that the grammar is ambiguous. More precisely, a grammar G is ***ambiguous*** iff there is at least one string in $L(G)$ for which G produces more than one parse tree. It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

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

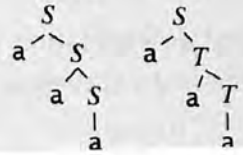
11.7.1 Why Is Ambiguity a Problem?

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 and 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	**Regular Grammar**
(a ∪ b)*a (a∪ b)*.	$S \rightarrow a$
	$S \rightarrow bS$
choose a from (a∪ b), then	$S \rightarrow aS$
choose a from (a∪ b), then	$S \rightarrow aT$
choose a, then	$T \rightarrow a$
choose ε from (a∪ b)*.	$T \rightarrow b$
	$T \rightarrow aT$
or	$T \rightarrow bT$
choose ε from (a∪ b)*, then	
choose a, then	
choose a from (a∪ b), then	
choose a from (a∪ b).	



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.

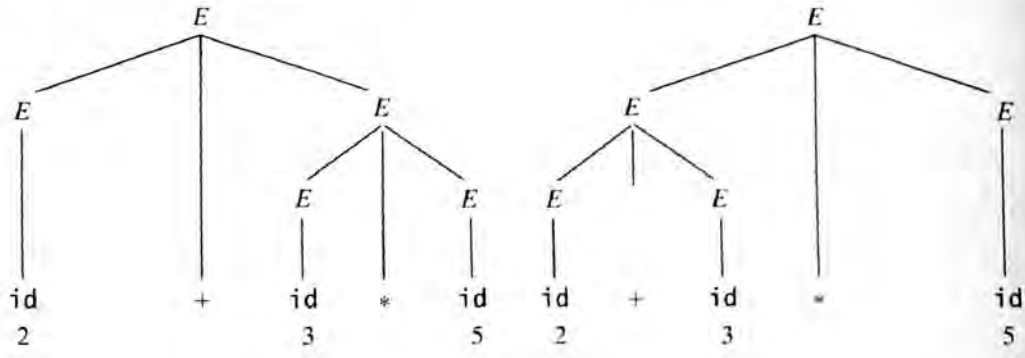
EXAMPLE 11.14 An Ambiguous Expression Grammar

Consider E_{xpr}, 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 a programming language. We can define E_{xpr} with the following context-free grammar $G = \{\{E, \texttt{id}, +, *, (,)\}, \{\texttt{id}, +, *, (,)\}, R, E\}$, where:

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

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 `id + id * 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 L.3.4.

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

11.7.2 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:

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

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?

11.7.3 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:

1. ε rules like $S \rightarrow \varepsilon$.
2. 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.
3. 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 $S \rightarrow \varepsilon$. If we could eliminate the rule $S \rightarrow \varepsilon$, 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 $\varepsilon \in L(G)$, 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 $\varepsilon \in L(G)$, 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 $L(G') = L(G) - \{\varepsilon\}$ 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 $X \rightarrow \varepsilon$, or
   (2) there is a rule $X \rightarrow PQR\ldots$ such that $P, Q, R, \ldots$ 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 $\alpha\beta \neq \varepsilon$ 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 $X \rightarrow \varepsilon$.
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 $G = (\{S, T, A, B, C, \mathsf{a}, \mathsf{b}, \mathsf{c}\}, \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}, R, S)$, where:

$$\begin{aligned} R = \{ & S \rightarrow \mathsf{a}T\mathsf{a} \\ & T \rightarrow ABC \\ & A \rightarrow \mathsf{a}A \mid C \\ & B \rightarrow B\mathsf{b} \mid C \\ & C \rightarrow \mathsf{c} \mid \varepsilon\}. \end{aligned}$$

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':

$S \rightarrow \mathsf{aa}$	/* Since T is nullable.
$T \rightarrow BC$	/* Since A is nullable.
$T \rightarrow AC$	/* Since B is nullable.
$T \rightarrow AB$	/* Since C is nullable.
$T \rightarrow C$	/* From $T \rightarrow BC$, since B is nullable. Or from $T \rightarrow AC$.
$T \rightarrow B$	/* From $T \rightarrow BC$, since C is nullable. Or from $T \rightarrow AB$.

EXAMPLE 11.17 (*Continued*)

$T \rightarrow A$ /* From $T \rightarrow AC$, since C is nullable. Or from $T \rightarrow AB$.
$A \rightarrow$ a /* Since A is nullable.
$B \rightarrow$ b /* Since B is nullable.

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. $G'' = removeEps(G)$.
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 Parens Grammar

We again consider Bal = $\{w \in \{), (\}^* :$ 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.

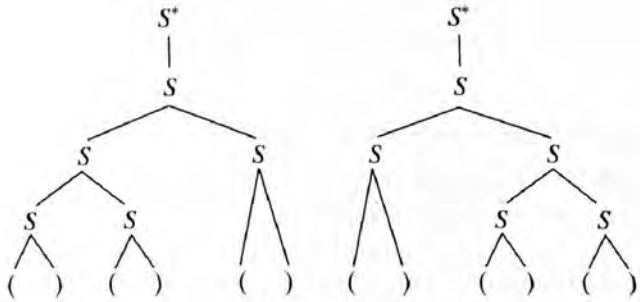


FIGURE 11.1 Two parse trees for the string ()()().

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:

$S \rightarrow SS_1$ /* force branching to the left.

$S \rightarrow S_1S$ /* 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:

$R = \{S* \rightarrow \varepsilon$ (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)

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_{xpr}, which we defined with the following context-free grammar $G = \{\{E, \texttt{id}, +, *, (,)\}, \{\texttt{id}, +, *, (,)\}, R, E\}$, where:

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

G is ambiguous in two ways:

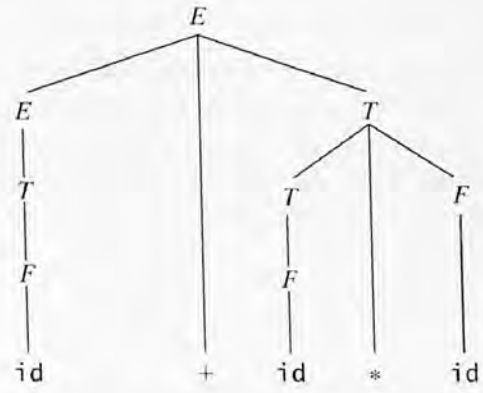
1. It fails to specify associativity. So, for example, there are two parses for the string `id + id + id`, corresponding to the bracketings `(id + id) + id` and `id + (id + id)`.
2. It fails to define a precedence hierarchy for the operators + and *. So, for example, there are two parses for the string `id + id * id`, corresponding to the bracketings `(id + id) * id` and `id + (id * 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 $G' = \{\{E, T, F, \texttt{id}, +, *, (,)\}, \{\texttt{id}, +, *, (,)\}, R, E\}$, where:

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

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 `id + id * 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` $cond_1$ `then if` $cond_2$ `then` st_1 `else` st_2

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

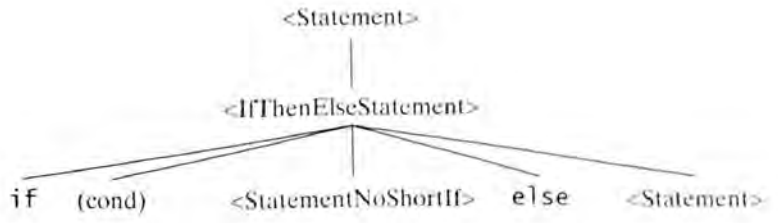
Most programming languages that have the dangling else problem (including C, C++, and Java) specify that each `else` goes with the innermost `if` to which it can be attached. The Java grammar forces this to happen by changing the rules to something like these (presented here in a simplified form that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

EXAMPLE 11.20 (*Continued*)

<IfThenElseStatementNoShortIf> ::= `if` (<Expression>)
<StatementNoShortIf> `else` <StatementNoShortIf>

In this grammar, there is a special class of statements called <Statement NoShortIf>. 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 other ways in which programming languages can solve this problem, see G.3.

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 Parens Grammar is Unambiguous

We return to the final grammar G that we produced for Bal. $G = (\{S,), (\}, \{), (\}, R, S)$, 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 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, on the right, to m and all of p's siblings. So, for example, consider the string:

$$\underset{5}{\underline{\left(\underset{1}{\underline{()}}\,\underset{2}{\underline{()}}\right)}}\,\underset{3}{\underline{()}}\,\underset{4}{\underline{()}}$$

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 L.3.4 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. (L.3.4)

11.8 Normal Forms ✹

So far, we've imposed no restrictions on the form of the right-hand sides 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. (H.5) Normal forms for logical formulas can simplify automated reasoning in artificial intelligence systems (M.2) and in program verification systems. (H.1.1)

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 a Chomsky normal form grammar G_C such that $L(G_C) = L(G_C) - \{\varepsilon\}$.

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 a Greibach normal form grammar G_G such that $L(G_G) = L(G) - \{\varepsilon\}$.

Proof: The proof is also by construction. We present it in D.1.

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:

$X \rightarrow \mathrm{a}Y\mathrm{c}$
$Y \rightarrow \mathrm{b}$
$Y \rightarrow ZZ$

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

$X \rightarrow \mathrm{abc}$
$X \rightarrow \mathrm{a}ZZ\mathrm{c}$

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 $Y \rightarrow \gamma_1 | \gamma_2 | \ldots | \gamma_n$ 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 $X \rightarrow \alpha\gamma_1\beta, X \rightarrow \alpha\gamma_2\beta, \ldots, X \rightarrow \alpha\gamma_n\beta$. 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:

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

Then G' can derive w with the derivation:

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

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 $L(G_C) = L(G) - \{\varepsilon\}$. 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 11.7.4.
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

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:

$S \rightarrow XY$
$X \rightarrow A$
$A \rightarrow B \mid a$
$B \rightarrow b$

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:

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

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 $A \rightarrow B \mid \mathsf{a}$, add $X \rightarrow B \mid \mathsf{a}$.
Remove $X \rightarrow B$. Add $X \rightarrow \mathsf{b}$.
Remove $Y \rightarrow T$. Add $Y \rightarrow Y \mid \mathsf{c}$. 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 $A \rightarrow \mathsf{b}$.
Remove $T \rightarrow Y$. Add $T \rightarrow \mathsf{c}$, but with no effect since it was already present.

At this point, the rules of G are:

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

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:

$A \rightarrow \mathrm{a}$
$A \rightarrow \mathrm{a}B$
$A \rightarrow B\mathrm{a}C$
$A \rightarrow B\mathrm{b}C$

is the grammar:

$A \rightarrow \mathrm{a}$
$A \rightarrow T_a B$
$A \rightarrow B T_a C$
$A \rightarrow B T_b C$
$T_a \rightarrow \mathrm{a}$
$T_b \rightarrow \mathrm{b}$

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 $A \rightarrow N_1 N_2 N_3 N_4 \ldots N_n$, $n > 2$, create new nonterminals $M^k_2, M^k_3, \ldots M^k_{n-1}$.
3. In G', replace r^k with the rule $A \rightarrow N_1 M^k_2$.
4. To G', add the rules $M^k_2 \rightarrow N_2 M^k_3$, $M^k_3 \rightarrow N_3 M^k_4, \ldots M^k_{n-1} \rightarrow N_{n-1} N_n$.
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:

$A \rightarrow BM_2$
$M_2 \rightarrow CM_3$
$M_3 \rightarrow DM_4$
$M_4 \rightarrow EF$

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

EXAMPLE 11.26 Converting a Grammar to Chomsky Normal Form

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

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

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:

$$S \rightarrow \mathrm{a}AC\mathrm{a} \mid \mathrm{a}A\mathrm{a} \mid \mathrm{a}C\mathrm{a} \mid \mathrm{aa}$$

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

$$\begin{aligned} & S \rightarrow \mathrm{a}AC\mathrm{a} \mid \mathrm{a}A\mathrm{a} \mid \mathrm{a}C\mathrm{a} \mid \mathrm{aa} \\ & A \rightarrow B \mid \mathrm{a} \\ & B \rightarrow C \mid \mathrm{c} \\ & C \rightarrow \mathrm{c}C \mid \mathrm{c} \end{aligned}$$

Next we apply *removeUnits*:

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

So *removeUnits* returns the rule set:

$$\begin{aligned} & S \rightarrow \mathrm{a}AC\mathrm{a} \mid \mathrm{a}A\mathrm{a} \mid \mathrm{a}C\mathrm{a} \mid \mathrm{aa} \\ & A \rightarrow \mathrm{a} \mid \mathrm{c} \mid \mathrm{c}C \\ & B \rightarrow \mathrm{c} \mid \mathrm{c}C \\ & C \rightarrow \mathrm{c}C \mid \mathrm{c} \end{aligned}$$

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

$$\begin{aligned} & S \rightarrow T_a A C T_a \mid T_a A T_a \mid T_a C T_a \mid T_a T_a \\ & A \rightarrow \mathrm{a} \mid \mathrm{c} \mid T_c C \\ & B \rightarrow \mathrm{c} \mid T_c C \\ & C \rightarrow T_c C \mid \mathrm{c} \end{aligned}$$

EXAMPLE 11.26 (*Continued*)

$T_a \rightarrow \mathsf{a}$
$T_c \rightarrow \mathsf{c}$

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

$S \rightarrow T_a S_1 \quad S \rightarrow T_a S_3 \quad S \rightarrow T_a S_4 \quad S \rightarrow T_a T_a$
$S_1 \rightarrow A S_2 \quad S_3 \rightarrow A T_a \quad S_4 \rightarrow C T_a$
$S_2 \rightarrow C T_a$

$A \rightarrow \mathsf{a} \mid \mathsf{c} \mid T_c C$
$B \rightarrow \mathsf{c} \mid T_c C$
$C \rightarrow T_c C \mid \mathsf{c}$
$T_a \rightarrow \mathsf{a}$
$T_c \rightarrow \mathsf{c}$

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 $X \rightarrow A_1 A_2 A_3 \ldots A_k$. 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 grows linearly.

4. Use *removeLong* to remove rules with long right-hand sides: This step runs in linear time and constructs a grammar whose size grows linearly.

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 `<b><i>bold italic</b></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. (H.4.2)

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]	<input> → <chunk>*	
[2]	<chunk> → CALL <id> (<expr>)	{cons(CALL)}
[3]	<chunk> → CALL ERROR (<expr>)	{reject}
[4]	<chunk> → <water>	
[5]	<water> → Σ^*	{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 $V - \Sigma$, and

- 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 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:

$S \rightarrow aSa$ [.72]
$S \rightarrow bSb$ [.24]
$S \rightarrow \varepsilon$ [.04]

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 $\Pr(r)$ be the probability associated with rule r, then:

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

Stochastic context-free grammars play an important role in natural language processing. (L.3.6)

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:

$$s = \underset{w \in X}{\text{argmax}}\, Pr\,(w|o)$$

$$= \underset{w \in X}{\text{argmax}} \frac{\Pr\,(o|w)\Pr\,(w)}{\Pr\,(o)}.$$

Stochastic context-free grammars can be used model the three-dimensional structure of RNA. (K.4)

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.

Exercises

1. Let $\Sigma = \{\mathsf{a}, \mathsf{b}\}$. 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., $\mathsf{a}^n\mathsf{b}^n$, or set theoretic expressions (e.g., $\{x: \ldots\}$).
   - **iv.** Indicate whether or not L is regular. Prove your answer.

   a. $S \rightarrow \mathsf{a}S \mid S\mathsf{b} \mid \varepsilon$

   b. $S \rightarrow \mathsf{a}S\mathsf{a} \mid \mathsf{b}S\mathsf{b} \mid \mathsf{a} \mid \mathsf{b}$

   c. $S \rightarrow \mathsf{a}S \mid \mathsf{b}S \mid \varepsilon$

   d. $S \rightarrow \mathsf{a}S \mid \mathsf{a}S\mathsf{b}S \mid \varepsilon$
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:

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

   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:

   $$S \rightarrow \mathsf{a}S\mathsf{a}$$

$$S \rightarrow T$$
$$S \rightarrow \varepsilon$$
$$T \rightarrow \mathsf{b}T$$
$$T \rightarrow \mathsf{c}T$$
$$T \rightarrow \varepsilon$$

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:
   a. `The bear shot Fluffy with the rifle.`
   b. `Fluffy likes the girl with the chocolate.`
6. Show a context-free grammar for each of the following languages L:
   a. BalDelim = $\{w$: where w is a string of delimiters: (,), [,], {, }, that are properly balanced$\}$.
   b. $\{\mathsf{a}^i\mathsf{b}^j : 2i = 3j + 1\}$.
   c. $\{\mathsf{a}^i\mathsf{b}^j : 2i \neq 3j + 1\}$.
   d. $\{w \in \{\mathsf{a}, \mathsf{b}\}^* : \#_{\mathsf{a}}(w) = 2 \cdot \#_{\mathsf{b}}(w)\}.\}$.
   e. $L = \{w \in \{\mathsf{a}, \mathsf{b}\}^* : w = w^{\mathrm{R}}\}$.
   f. $\{\mathsf{a}^i\mathsf{b}^j\mathsf{c}^k : i, j, k \geq 0$ and $(i \neq j$ or $j \neq k)\}$.
   g. $\{\mathsf{a}^i\mathsf{b}^j\mathsf{c}^k : i, j, k \geq 0$ and $(k \leq i$ or $k \leq j)\}$.
   h. $\{w \in \{\mathsf{a}, \mathsf{b}\}^*$: every prefix of w has at least as many a's as b's$\}$.
   i. $\{\mathsf{a}^n\mathsf{b}^m : m \geq n, m\text{-}n$ is even$\}$.
   j. $\{\mathsf{a}^m\mathsf{b}^n\mathsf{c}^p\mathsf{d}^q : m, n, p, q \geq 0$ and $m + n = p + q\}$.
   k. $\{x\mathsf{c}^n : x \in \{\mathsf{a}, \mathsf{b}\}^*$ and $(\#_{\mathsf{a}}(x) = n$ or $\#_{\mathsf{b}}(x) = n)\}$.
   l. $\{b_i\#b_{i+1}{}^{\mathrm{R}} : b_i$ is the binary representation of some integer i, $i \geq 0$, without leading zeros$\}$. (For example 101#011 $\in L$.)
   m. $\{x^{\mathrm{R}}\#y : x, y \in \{0, 1\}^*$ and x 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.
   a. Trace a derivation of the string `id+id*id*id` in G'.
   b. 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 $L = \{w \in \{\mathsf{a}, \mathsf{b}, \cup, \varepsilon, (,), {}^*, {}^+\}^* : w$ is a syntactically legal regular expression$\}$.
   a. 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
  - $\cup$

b. Show the parse tree that your grammar will produce for the string (a $\cup$ b) ba*.

10. Let $L = \{w \in \{A - Z, \neg, \wedge, \vee, \rightarrow, (,)\}^* : w$ is a syntactically legal Boolean expression$\}$.

a. Write an unambiguous context-free grammar that generates L and that creates 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): $\neg$, $\wedge$, $\vee$, and $\rightarrow$.

b. Show the parse tree that your grammar will produce for the string:

$$\neg P \vee R \rightarrow Q \rightarrow S$$

11. In I.3.1, we present a simplified grammar for URIs (Uniform Resource Identifiers), the names that we use to refer to objects on the Web.

a. Using that grammar, show a parse tree for:

```
https://www.mystuff.wow/widgets/fradgit#sword
```

b. Write a regular expression that is equivalent to the grammar that we present.

12. Prove that each of the following grammars is correct:

a. The grammar, shown in Example 11.3, for the language PalEven.

b. 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.

a. $(\{S, A, B, T, \mathsf{a}, \mathsf{c}\}, \{\mathsf{a}, \mathsf{c}\}, R, S)$, where $R = \{S \rightarrow AB, S \rightarrow BA, A \rightarrow \mathsf{a}A, A \rightarrow \mathsf{ac}, B \rightarrow T\mathsf{c}, T \rightarrow \mathsf{a}T, T \rightarrow \mathsf{a}\}$.

b. $(\{S, \mathsf{a}, \mathsf{b}\}, \{\mathsf{a}, \mathsf{b}\}, R, S)$, where $R = \{S \rightarrow \varepsilon, S \rightarrow \mathsf{a}S\mathsf{a}, S \rightarrow \mathsf{b}S\mathsf{b}, S \rightarrow \mathsf{a}S\mathsf{b}, S \rightarrow \mathsf{b}S\mathsf{a}, S \rightarrow SS\}$.

c. $\{\{S, A, B, T, \mathsf{a}, \mathsf{c}\}, \{\mathsf{a}, \mathsf{c}\}, R, S)$, where $R = \{S \rightarrow AB, A \rightarrow AA, A \rightarrow \mathsf{a}, B \rightarrow T\mathsf{c}, T \rightarrow \mathsf{a}T, T \rightarrow \mathsf{a}\}$.

d. $(\{S, \mathsf{a}, \mathsf{b}\}, \{\mathsf{a}, \mathsf{b}\}, R, S)$, where $R = \{S \rightarrow \mathsf{a}S\mathsf{b}, S \rightarrow \mathsf{b}S\mathsf{a}, S \rightarrow SS, S \rightarrow \varepsilon\}$. ($G$ is the grammar that we presented in Example 11.10 for the language $L = \{w \in \{\mathsf{a},\mathsf{b}\}^* : \#_{\mathsf{a}}(w) = \#_{\mathsf{b}}(w)\}$.)

e. $(\{S, \mathsf{a}, \mathsf{b}\}, \{\mathsf{a}, \mathsf{b}\}, R, S)$, where $R = \{S \rightarrow \mathsf{a}S\mathsf{b}, S \rightarrow \mathsf{aa}S\mathsf{b}, S \rightarrow \varepsilon\}$.

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 G.5, avoid the dangling else problem?

17. Consider the grammar G' of Example 11.19.

a. Convert G' to Chomsky normal form.

b. Consider the string `id*id+id`.

i. Show the parse tree that G' produces for it.

ii. Show the parse tree that your Chomsky normal form grammar produces for it.

18. Convert each of the following grammars to Chomsky normal form:

a. $S \rightarrow \mathtt{a}S\mathtt{a}$
$S \rightarrow B$
$B \rightarrow \mathtt{bb}C$
$B \rightarrow \mathtt{bb}$
$C \rightarrow \varepsilon$
$C \rightarrow \mathtt{c}C$

b. $S \rightarrow ABC$
$A \rightarrow \mathtt{a}C \mid D$
$B \rightarrow \mathtt{b}B \mid \varepsilon \mid A$
$C \rightarrow A\mathtt{c} \mid \varepsilon \mid C\mathtt{c}$
$D \rightarrow \mathtt{aa}$

c. $S \rightarrow \mathtt{a}T V\mathtt{a}$
$T \rightarrow \mathtt{a}T\mathtt{a} \mid \mathtt{b}T\mathtt{b} \mid \varepsilon \mid V$
$V \rightarrow \mathtt{c}V\mathtt{c} \mid \varepsilon$

CHAPTER 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 sextuple $(K, \Sigma, \Gamma, \Delta, s, A)$, 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:

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

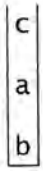
state | input or ε | string of symbols to pop from top of stack | state | string of symbols to push on top of stack

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:



will be written as cab

If a sequence $c_1c_2 \ldots c_n$ 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 $c_1c_2 \ldots c_ns$.

Analogously to what we did for FSMs, we define the relation *yields-in-one-step*, written $|\text{-}_M$. *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:

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

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 $|\text{-}_M^*$, is the reflexive, transitive closure of $|\text{-}_M$. So configuration C_1 yields configuration C_2 iff:

$$C_1 |\text{-}_M^* C_2.$$

A ***computation*** by M is a finite sequence of configurations $C_0, C_1, \ldots, C_n$ 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
- $C_0 |\text{-}_M C_1 |\text{-}_M C_2 |\text{-}_M \ldots |\text{-}_M C_n$.

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 $C = (s, w, \varepsilon) |\text{-}_M^* (q, \varepsilon, \varepsilon)$, 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 $C = (s, w, \varepsilon) |\text{-}_M^* (q, w', \alpha)$, 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.

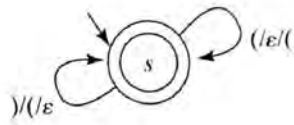
The ***language accepted by*** M, denoted $L(M)$, is the set of all strings accepted by M. Note that it is possible that, on input w, M neither accepts nor rejects.

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 Bal = $\{w \in \{), (\}^* :$ 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)

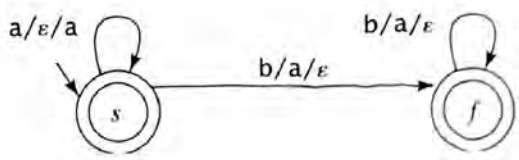
$\Delta = \{((s, (, \varepsilon), (s, ( )),$
$((s,), ( ), (s, \varepsilon))\}$.

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^nB^n

Consider again $A^nB^n = \{a^nb^n : n \geq 0\}$. The following PDA M accepts A^nB^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:



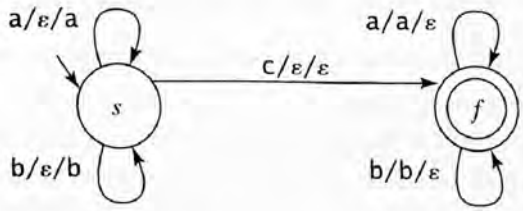
Writing it out, we have $M = (K, \Sigma, \Gamma, \Delta, s, A)$, where:

$$
\begin{aligned}
K &= \{s, f\}, && \text{(the states)}\\
\Sigma &= \{\mathsf{a}, \mathsf{b})\}, && \text{(the input alphabet)}\\
\Gamma &= \{\mathsf{a}\}, && \text{(the stack alphabet)}\\
A &= \{s, f\}, \text{ and} && \text{(the accepting states)}\\
\Delta &= \{((s, \mathsf{a}, \varepsilon), (s, \mathsf{a})),\\
&\quad\ ((s, \mathsf{b}, \mathsf{a}), (f, \varepsilon)),\\
&\quad\ ((f, \mathsf{b}, \mathsf{a}), (f, \varepsilon))\}.
\end{aligned}
$$

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 $\mathrm{WcW^R}$

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



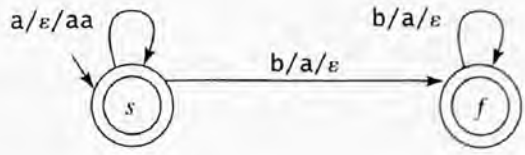
M moves from state s, in which it is recording w, to state f, in which it is checking for w^R, when it sees the character c. Since every string in $\mathrm{WcW^R}$ must contain the middle c, state s is not an accepting state.

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 $\mathrm{A^nB^{2n}}$

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

EXAMPLE 12.4 (*Continued*)



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 $((q, \varepsilon, \varepsilon), (p, a))$ 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 may correspond to one computation that M might perform.

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.

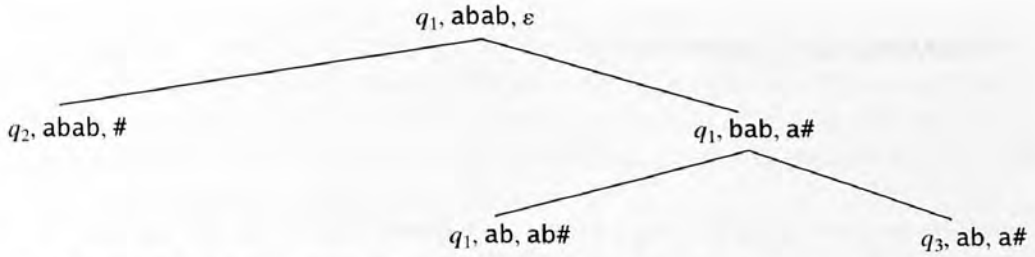
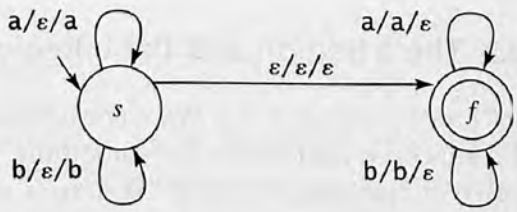


FIGURE 12.1 Viewing nondeterminism as search through a space of computation paths.

EXAMPLE 12.5 Even Length Palindromes

Consider again 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:

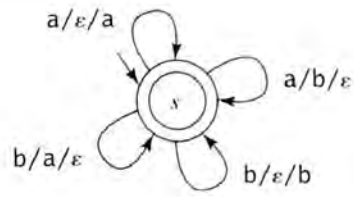


M is nondeterministic because it cannot know when it has reached the middle of its input. Before each character is read, it has two choices: It can guess that it has not yet gotten to the middle. In that case, it stays in state s, where it pushes each symbol it reads. Or it can guess that it has reached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

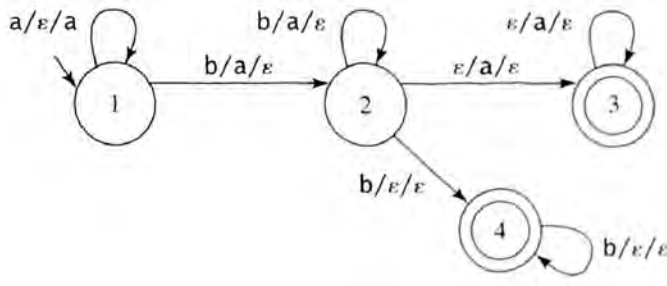
EXAMPLE 12.6 (*Continued*)



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 $\#_{\mathsf{a}}(w) = \#_{\mathsf{b}}(w)$, there will be one computation that will accept.

EXAMPLE 12.7 The a Region and the b Region are Different

Let $L = \{\mathsf{a}^m\mathsf{b}^n : m \neq n; m, n > 0\}$. 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 $\neg A^nB^nC^n$

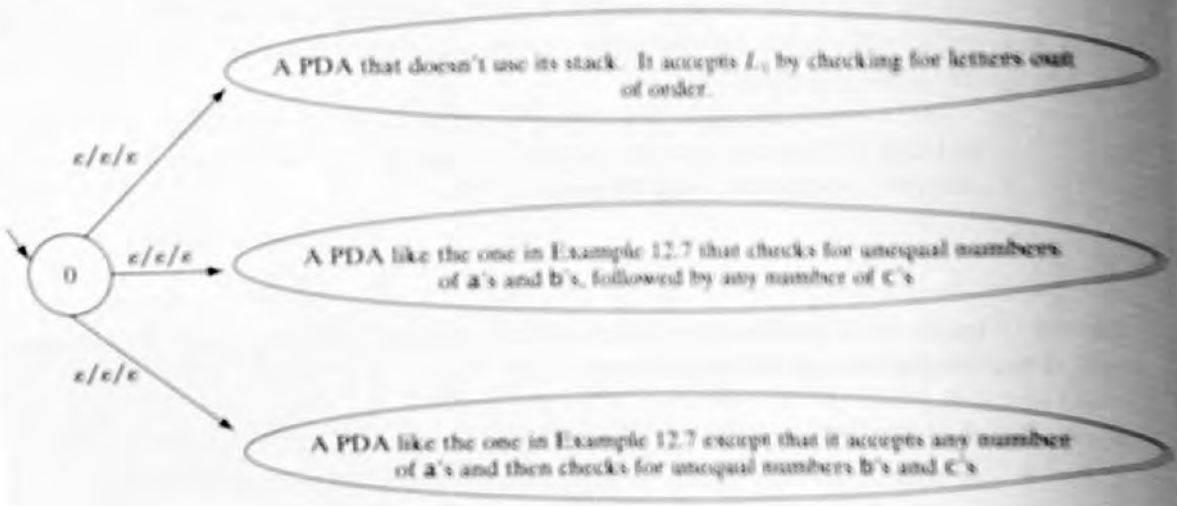
Let's first consider $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$. If we try to think about building a PDA to accept $A^nB^nC^n$, we immediately run into trouble. We can use the stack to count a's and then compare them to the b's. But then the stack will be empty and it won't be possible to compare the c's. 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 $L = \neg A^nB^nC^n$. There is a PDA that accepts L. $L = L_1 \cup L_2$, where:

- $L_1 = \{w \in \{a, b, c\}^* :$ the letters are out of order$\}$.
- $L_2 = \{a^ib^jc^k : i, j, k \geq 0$ and $(i \neq j$ 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:

EXAMPLE 12.8 (*Continued*)



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:

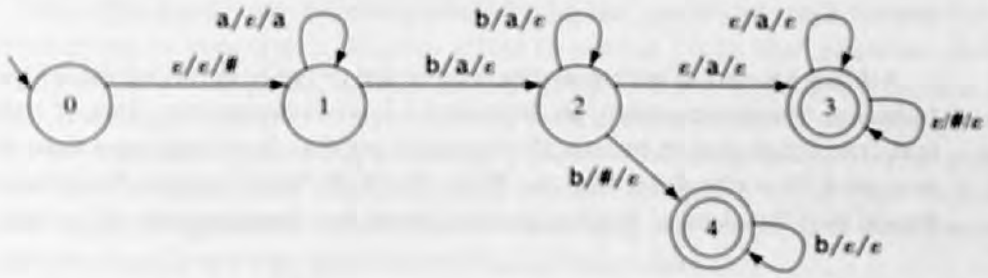
- A transition that should be taken only if the stack is empty competes against one or more moves that require a match of some string on the stack, and
- A transition that should be taken only if the input stream is empty competes against one or more moves that require a match against a specific input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



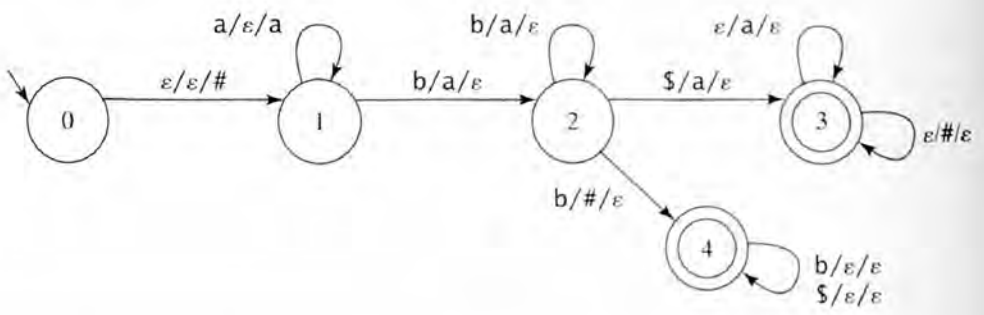
Now the transition back to state 2 no longer competes with the transition to state 4, which can only be taken when the # is the only symbol left on the stack. M' is still nondeterministic though, because the transition back to state 2 competes with the transition to state 3. We still don't have a way 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'':

EXAMPLE 12.10 (*Continued*)



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 *cfgtoPDAtopdown*(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 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 $S \rightarrow \gamma_1\gamma_2 \ldots \gamma_n$. 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 $\gamma_1\gamma_2 \ldots \gamma_n$ (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.

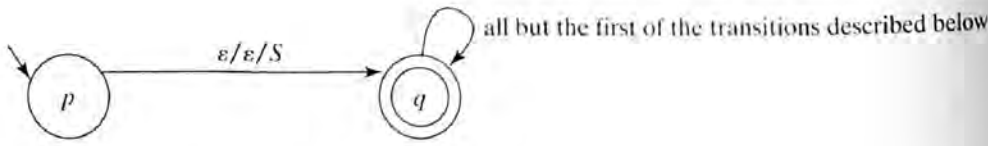


FIGURE 12.2 A PDA that parses top-down.

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

- The start-up transition $((p, \varepsilon, \varepsilon), (q, S))$, which pushes the start symbol onto the stack and goes to state q.
- For each rule $X \rightarrow \gamma_1\gamma_2 \ldots \gamma_n$ in R, the transition $((q, \varepsilon, X), (q, \gamma_1\gamma_2 \ldots \gamma_n))$, which replaces X by $\gamma_1\gamma_2 \ldots \gamma_n$. If $n = 0$ (i.e., the right-hand side of the rule is ε), then the transition is $((q, \varepsilon, X), (q, \varepsilon))$.
- 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.

So we can define:

cfgtoPDAtopdown (*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 *cfgtoPDAbottomup*(*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.

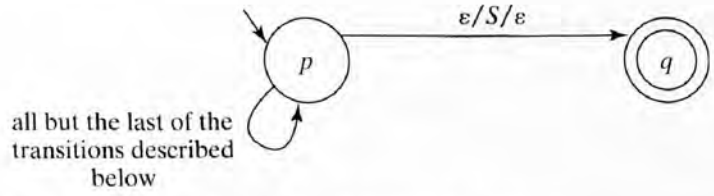


FIGURE 12.3 A PDA that parses bottom-up.

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.

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

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

So we can define:

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

EXAMPLE 12.11 Using *cfgtoPDAtopdown* and *cfgtoPDAbottomup*

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

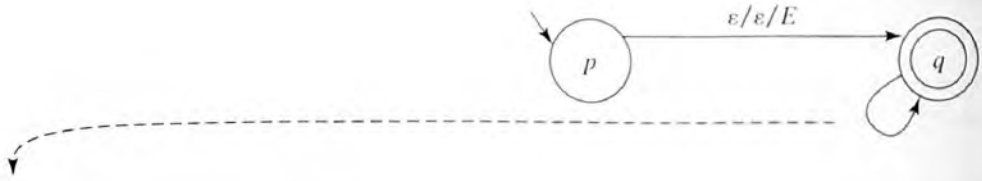
$$\begin{aligned} R = \quad & E \rightarrow E + T \\ & E \rightarrow T \\ & T \rightarrow T * F \\ & T \rightarrow F \end{aligned}$$

EXAMPLE 12.11 (*Continued*)

$$F \rightarrow E$$

$$F \rightarrow \texttt{id}\}.$$

We show two PDAs, M_a and M_b, that accept $\mathrm{E}_{\mathrm{xpr}}$. We can use the function *cfgtoPDAtopdown*(G) to build M_a =



(1) $(q, \varepsilon, E), (q, E + T)$

(2) $(q, \varepsilon, E), (q, T)$

(3) $(q, \varepsilon, T), (q, T * F)$

(4) $(q, \varepsilon, T), (q, F)$

(5) $(q, \varepsilon, F), (q, (E))$

(6) $(q, \varepsilon, F), (q, \texttt{id})$

(7) $(q, \texttt{id}, \texttt{id}), (q, \varepsilon)$

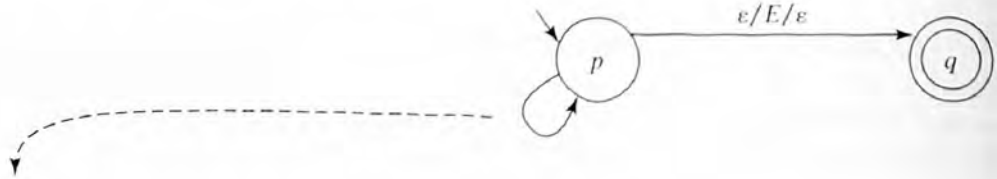
(8) $(q, (, ( ), (q, \varepsilon)$

(9) $(q,),)), (q, \varepsilon)$

(10) $(q, +, +), (q, \varepsilon)$

(11) $(q, *, *), (q, \varepsilon)$

We can use *cfgtoPDAbottomup*(G) to build M_b =



(1) $(p, \texttt{id}, \varepsilon), (p, \texttt{id})$

(2) $(p, (, \varepsilon), (p, ()$

(3) $(p,), \varepsilon), (p,))$

(4) $(p, +, \varepsilon), (p, +)$

(5) $(p, *, \varepsilon), (p, *)$

(6) $(p, \varepsilon, T + E), (p, E)$

(7) $(p, \varepsilon, T), (p, E)$

(8) $(p, \varepsilon, F * T), (p, T)$

(9) $(p, \varepsilon, F), (p, T)$

(10) $(p, \varepsilon,)E ( ), (p, F)$

(11) $(p, \varepsilon, \texttt{id}), (p, F)$

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 used in any other way in M. 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 $qq_1, qq_2, \ldots, qq_{k-1}$ be new state names. Then:

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

   $$((q_1, \varepsilon, \gamma_1), (qq_1, \varepsilon)), ((qq_1, \varepsilon, \gamma_2), (qq_2, \varepsilon)), \ldots,$$

   $$((qq_{k-1}, c, \gamma_n), (q_2, \gamma_P)).$$

/* 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.

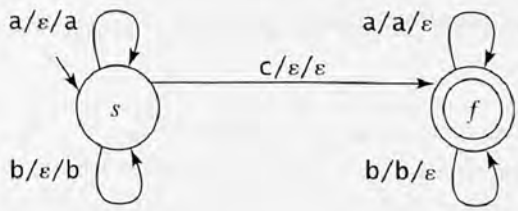
7. For every transition $t = ((q_1, c, \varepsilon), (q_2, \gamma))$ do:

   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 $((q_1, c, \alpha), (q_2, \gamma\alpha))$.

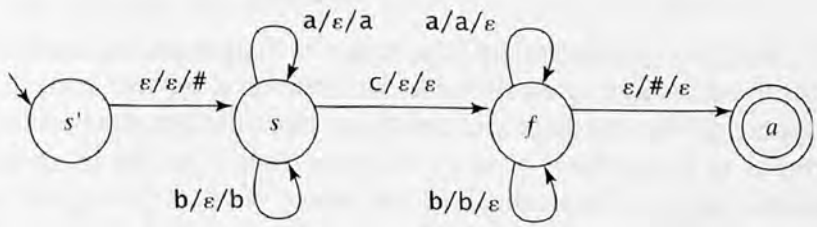
8. Return M'.

EXAMPLE 12.12 Converting to Restricted Normal Form

Let $\text{WcW}^R = \{wcw^R : w \in \{a, b\}^*\}$. 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:

((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)).

Building the Grammar

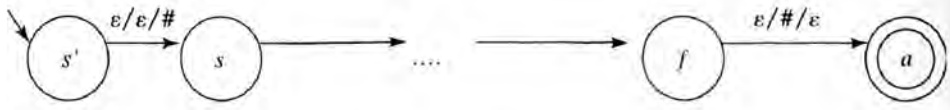
Since we have now shown that any PDA can be converted into an equivalent one in restricted normal form, we can show that, for any PDA M, there exists a context-free grammar that generates $L(M)$ by first converting M to restricted normal form and then constructing a grammar.

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:

$$<q_i, \gamma, q_j>.$$

The job of a nonterminal $<q_i, \gamma, q_j>$ 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 $<s, \#, a>$ 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:

$$S \rightarrow <s, \#, a>.$$

Now we need to describe the rules that will have $<s, \#, a>$ 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 $<f, \#, a>$, 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 $((q, c, \gamma), (r, \varepsilon))$, where $c \in \Sigma \cup \{\varepsilon\}$. We consider how such a transition can participate in a computation of M:

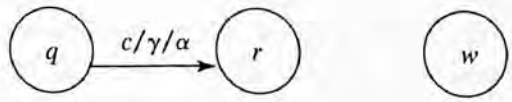
If this transition is taken, then M reads c, pops γ, and then moves to r. After doing that, it may follow any available paths from r to any next state w, where w may be q or r or any other state. So consider the nonterminal $<q, \gamma, w>$, for any state w. Its job is to generate all strings that drive 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 c 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:

$$<q, \gamma, w> \rightarrow c<r, \varepsilon, w>.$$

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 c, popping γ, going to r, and then somehow getting from r to w, leaving the stack just as it was. Since M reads c, G must generate it.

Every transition in M of the form $((q, c, \gamma), (r, \varepsilon))$ 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 c, 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 $<q, \gamma, w>$, 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 c 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:

$$<q, \gamma, w> \rightarrow c<r, \alpha, w>.$$

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 c, 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 $((q, c, \gamma), (r, \alpha))$ 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 c, 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 c 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:

$$<q, \gamma, w> \rightarrow c<r, \alpha, v><v, \beta, w>$$

Every transition in M of the form $((q, c, X), (r, \alpha\beta))$ 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:

$$<q, \varepsilon, q> \rightarrow \varepsilon.$$

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: 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 $S \rightarrow <s, \#, a>$.
   - **3.2.** For every transition $((q, c, \gamma), (r, \varepsilon))$ (i.e., every transition that pushes no symbols), and every state w, except s', in M do:
     Insert the rule $<q, \gamma, w> \rightarrow c<r, \varepsilon, w>$.
   - **3.3.** For every transition $((q, c, \gamma), (r, \alpha))$ (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 $<q, \gamma, w> \rightarrow c<r, \alpha, w>$.
   - **3.4.** For every transition $((q, c, \gamma), (r, \alpha\beta))$ (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 $<q, \gamma, w> \rightarrow c<r, \alpha, v><v, \beta, w>$.
   - **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 $<q, \varepsilon, q> \rightarrow \varepsilon$.
4. Set V_G to $\Sigma_M \cup \{$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 $((q_6, c, \gamma), (q_7, \alpha))$ 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 $<q_6, \gamma, q_8> \rightarrow c<q_7, \alpha, q_8>$. But $<q_7, \alpha, q_8>$ is unproductive since there are no strings that drive M from q_7 to q_8.

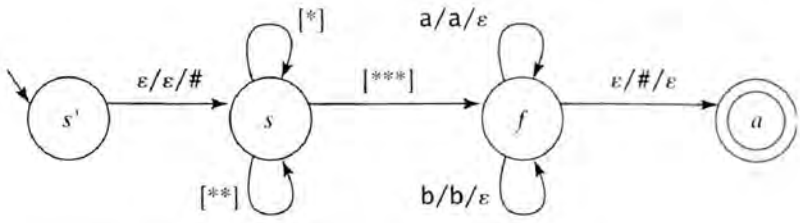
Finally, for an arbitrary PDA M, we define *PDAtoCFG*:

PDAtoCFG (M: 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':



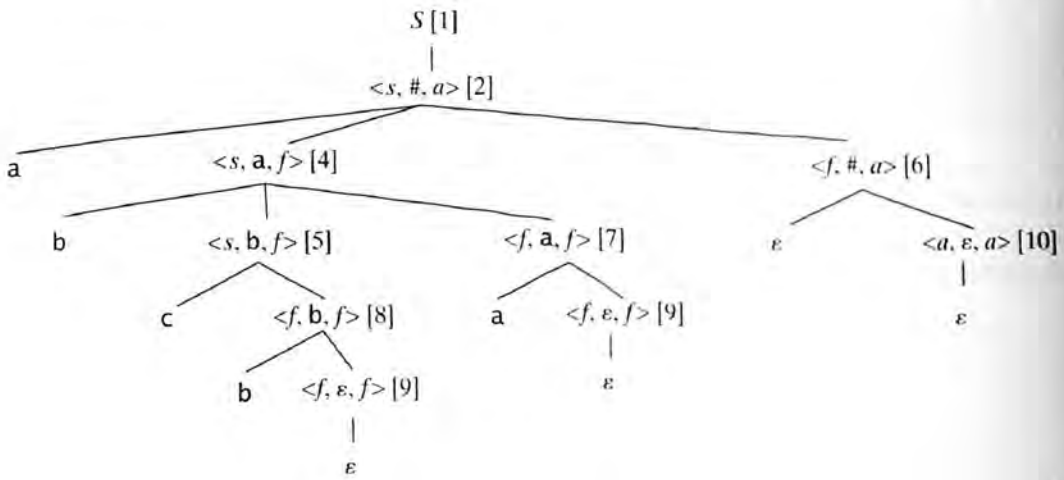
Each of the bracket-labeled arcs corresponds to:

[*] ((*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)), and

[***] ((*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.

		$S \rightarrow <s, \#, a>$	[1]
	$((s', \varepsilon, \varepsilon), (s, \#))$	no rules based on the transition from s''	
[*]	$((s, a, \#), (s, a\#))$	$<s, \#, s> \rightarrow a <s, a, s> <s, \#, s>$	[x]
		$<s, \#, s> \rightarrow a <s, a, f> <f, \#, s>$	[x]
		$<s, \#, s> \rightarrow a <s, a, a> <a, \#, s>$	[x]
		$<s, \#, f> \rightarrow a <s, a, s> <s, \#, f>$	[x]
		$<s, \#, f> \rightarrow a <s, a, f> <f, \#, f>$	[x]
		$<s, \#, f> \rightarrow a <s, a, a> <a, \#, f>$	[x]
		$<s, \#, a> \rightarrow a <s, a, s> <s, \#, a>$	[x]
		$<s, \#, a> \rightarrow a <s, a, f> <f, \#, a>$	[2]
		$<s, \#, a> \rightarrow a <s, a, a> <a, \#, a>$	[x]
	$((s, a, a), (s, aa))$	$<s, a, f> \rightarrow a<s, a, f> <f, a, f>$	[3]
	$((s, a, b), (s, ab))$	$<s, b, f> \rightarrow a <s, a, f> <f, b, f>$	[14]
[**]	$((s, b, \#), (s, b\#))$	$<s, \#, f> \rightarrow b <s, b, f> <f, \#, f>$	[15]
	$((s, b, a), (s, ba))$	$<s, a, f> \rightarrow b <s, b, f> <f, a, f>$	[4]
	$((s, b, b), (s, bb))$	$<s, b, f> \rightarrow b <s, b, f> <f, b, f>$	[16]
[***]	$((s, c, \#), (f, \#))$	$<s, \#, f> \rightarrow c <f, \#, f>$	[17]
	$((s, c, a), (f, a))$	$<s, a, f> \rightarrow c <f, a, f>$	[18]
	$((s, c, b), (f, b))$	$<s, b, f> \rightarrow c <f, b, f>$	[5]
	$((f, \varepsilon, \#), (a, \varepsilon))$	$<f, \#, a> \rightarrow \varepsilon <a, \varepsilon, a>$	[6]
	$((f, a, a), (f, \varepsilon))$	$<f, a, f> \rightarrow a <f, \varepsilon, f>$	[7]
	$((f, b, b), (f, \varepsilon))$	$<f, b, f> \rightarrow b <f, \varepsilon, f>$	[8]
		$<s, \varepsilon, s> \rightarrow \varepsilon$	[19]
		$<f, \varepsilon, f> \rightarrow \varepsilon$	[9]
		$<a, \varepsilon, a> \rightarrow \varepsilon$	[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:

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

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 $(1, a, \varepsilon) \vdash (2, a, a) \vdash (3, \varepsilon, \varepsilon)$ 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 computational path that has not yet halted. For example, on input aa, one such path is:
  (1, aa, ε) |- (2, aa, a) |- (1, aa, aa) |- (2, aa, aaa) |-
  (1, aa, aaaa) |- (2, aa, aaaaa) |-...
- 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.

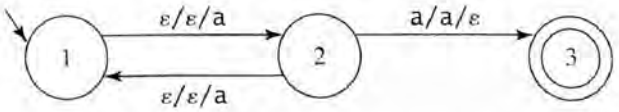


FIGURE 12.4 A PDA that may neither accept nor reject.

Problem 2 is especially critical. This 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.

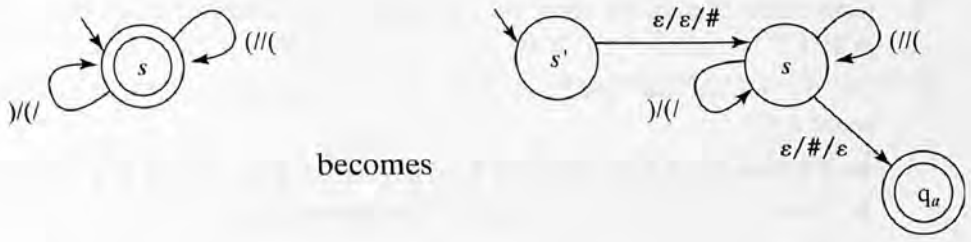
Given a PDA M that accepts by accepting state and empty stack, construct a new PDA M' that accepts by accepting state alone, where $L(M') = L(M)$. M' will have a single accepting state q_a. The only way for M' to get to q_a will be to land in an accepting state of M when the stack is logically empty. 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.

Exercises

1. Build a PDA to accept each of the following languages L:

a. BalDelim $= \{w :$ where w is a string of delimiters: (,), [,], {, }, that are properly balanced$\}$.

b. $\{a^i b^j : 2i = 3j + 1\}$.

c. $\{w \in \{a, b\}^* : \#_a(w) = 2 \cdot \#_b(w)\}$.

d. $\{a^n b^m : m \leq n \leq 2m\}$.

e. $\{w \in \{a, b\}^* : w = w^R\}$.

f. $\{a^i b^j c^k : i, j, k \geq 0$ and $(i \neq j$ or $j \neq k)\}$.

g. $\{w \in \{a, b\}^* :$ every prefix of w has at least as many a's as b's$\}$.

h. $\{a^n b^m a^n : n, m \geq 0$ and m is even$\}$.

i. $\{xc^n : x \in \{a, b\}^*, \#_a(x) = n$ or $\#_b(x) = n\}$.

j. $\{a^n b^m : m \geq n, m\text{-}n$ is even$\}$.

k. $\{a^m b^n c^p d^q : m, n, p, q \geq 0$ and $m + n = p + q\}$.

l. $\{b_i \# b_{i+1}{}^R : b_i$ is the binary representation of some integer i, $i \geq 0$, without leading zeros$\}$. (For example $101\#011 \in L$.)

m. $\{x^R \# y : x, y \in \{0,1\}^*$ and x is a substring of $y\}$.

n. $L_1{}^*$, where $L_1 = \{xx^R : x \in \{a,b\}^*\}$.

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 $L = \{ba^{m_1}ba^{m_2}ba^{m_3} \ldots ba^{m_n} : n \geq 2, m_1, m_2, \ldots, m_n \geq 0$, and $m_i \neq m_j$ for some $i, j\}$.

   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 $L_1 = \{ww^R : w \in \{a, b\}^*\}$ 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 $L = \{a^n b^+ a^m : n \geq 0$ 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 $L(M') = L(M)$.