CONTEXT-FREE LANGUAGES

In Chapter 1 we introduced two different, though equivalent, methods of describing languages: *finite automata* and *regular expressions*. We showed that many languages can be described in this way but that some simple languages, such as $\{0^n1^n | n \geq 0\}$, cannot.

In this chapter we introduce ***context-free grammars***, a more powerful method of describing languages. Such grammars can describe certain features that have a recursive structure which makes them useful in a variety of applications.

Context-free grammars were first used in the study of human languages. One way of understanding the relationship of terms such as *noun*, *verb*, and *preposition* and their respective phrases leads to a natural recursion because noun phrases may appear inside verb phrases and vice versa. Context-free grammars can capture important aspects of these relationships.

An important application of context-free grammars occurs in the specification and compilation of programming languages. A grammar for a programming language often appears as a reference for people trying to learn the language syntax. Designers of compilers and interpreters for programming languages often start by obtaining a grammar for the language. Most compilers and interpreters contain a component called a ***parser*** that extracts the meaning of a program prior to generating the compiled code or performing the interpreted execution. A number of methodologies facilitate the construction of a parser once a context-free grammar is available. Some tools even automatically generate the parser from the grammar.

The collection of languages associated with context-free grammars are called the ***context-free languages***. They include all the regular languages and many additional languages. In this chapter, we give a formal definition of context-free grammars and study the properties of context free languages. We also introduce ***pushdown automata***, a class of machines recognizing the context-free languages. Pushdown automata are useful because they allow us to gain additional insight into the power of context-free grammars.

2.1

CONTEXT-FREE GRAMMARS

The following is an example of a context-free grammar, which we'll call G_1.

$$\begin{aligned} A &\rightarrow 0A1 \\ A &\rightarrow B \\ B &\rightarrow \# \end{aligned}$$

A grammar consists of a collection of ***substitution rules***, also called ***productions***. Each rule appears as a line in the grammar and comprises a symbol and a string, separated by an arrow. The symbol is called a ***variable***. The string consists of variables and other symbols called ***terminals***. The variable symbols often are represented by capital letters. The terminals are analogous to the input alphabet and often are represented by lowercase letters, numbers, or special symbols. One variable is designated the ***start variable***. It usually occurs on the left-hand side of the topmost rule. For example, grammar G_1 contains three rules. G_1's variables are A and B, where A is the start variable. Its terminals are 0, 1, and #.

You use a grammar to describe a language by generating each string of that language in the following manner.

1. Write down the start variable. It is the variable on the left-hand side of the top rule, unless specified otherwise.
2. Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right-hand side of that rule.
3. Repeat step 2 until no variables remain.

For example, grammar G_1 generates the string 000#111. The sequence of substitutions to obtain a string is called a ***derivation***. A derivation of string 000#111 in grammar G_1 is

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

You may also represent the same information in a more pictorial way using a ***parse tree***. An example of a parse tree appears in the following figure.

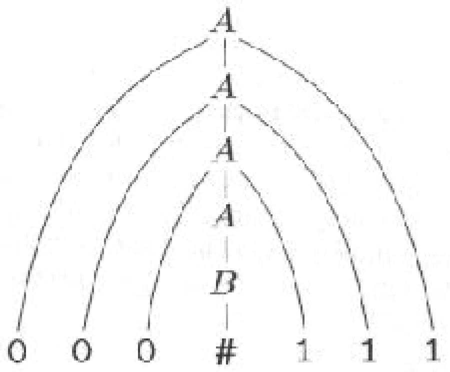


FIGURE 2.1
Parse tree for 000#111 in grammar G_1

All strings generated in this way constitute the ***language of the grammar***. We write $L(G_1)$ for the language of grammar G_1. Some experimentation with the grammar G_1 shows us that $L(G_1)$ is $\{0^n\#1^n \mid n \geq 0\}$. Any language that can be generated by some context-free grammar is called a ***context-free language*** (CFL). For convenience when presenting a context-free grammar, we abbreviate several rules with the same left-hand variable, such as $A \rightarrow 0A1$ and $A \rightarrow B$, into a single line $A \rightarrow 0A1 \mid B$, using the symbol " | " as an "or."

The following is a second example of a context-free grammar called G_2, which describes a fragment of the English language.

⟨SENTENCE⟩ → ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩
⟨NOUN-PHRASE⟩ → ⟨CMPLX-NOUN⟩ | ⟨CMPLX-NOUN⟩⟨PREP-PHRASE⟩
⟨VERB-PHRASE⟩ → ⟨CMPLX-VERB⟩ | ⟨CMPLX-VERB⟩⟨PREP-PHRASE⟩
⟨PREP-PHRASE⟩ → ⟨PREP⟩⟨CMPLX-NOUN⟩
⟨CMPLX-NOUN⟩ → ⟨ARTICLE⟩⟨NOUN⟩
⟨CMPLX-VERB⟩ → ⟨VERB⟩ | ⟨VERB⟩⟨NOUN-PHRASE⟩
⟨ARTICLE⟩ → a | the
⟨NOUN⟩ → boy | girl | flower
⟨VERB⟩ → touches | likes | sees
⟨PREP⟩ → with

Grammar G_2 has ten variables (the capitalized grammatical terms written inside brackets); 27 terminals (the standard English alphabet plus a space character); and eighteen rules. Strings in $L(G_2)$ include the following three examples.

```
a boy sees
the boy sees a flower
a girl with a flower likes the boy
```

Each of these strings has a derivation in grammar G_2. The following is a derivation of the first string on this list.

$$
\begin{aligned}
\langle\text{SENTENCE}\rangle &\Rightarrow \langle\text{NOUN-PHRASE}\rangle\langle\text{VERB-PHRASE}\rangle\\
&\Rightarrow \langle\text{CMPLX-NOUN}\rangle\langle\text{VERB-PHRASE}\rangle\\
&\Rightarrow \langle\text{ARTICLE}\rangle\langle\text{NOUN}\rangle\langle\text{VERB-PHRASE}\rangle\\
&\Rightarrow \texttt{a}\ \langle\text{NOUN}\rangle\langle\text{VERB-PHRASE}\rangle\\
&\Rightarrow \texttt{a boy}\ \langle\text{VERB-PHRASE}\rangle\\
&\Rightarrow \texttt{a boy}\ \langle\text{CMPLX-VERB}\rangle\\
&\Rightarrow \texttt{a boy}\ \langle\text{VERB}\rangle\\
&\Rightarrow \texttt{a boy sees}
\end{aligned}
$$

FORMAL DEFINITION OF A CONTEXT-FREE GRAMMAR

Let's formalize our notion of a context-free grammar (CFG).

DEFINITION 2.1

A ***context-free grammar*** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

If u, v, and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar, we say that uAv ***yields*** uwv, written $uAv \Rightarrow uwv$. Write $u \overset{*}{\Rightarrow} v$ if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v.$$

The ***language of the grammar*** is $\{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$.

In grammar G_1, $V = \{A, B\}$, $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{\#}\}$, $S = A$, and R is the collection of the three rules appearing on page 92. In grammar G_2,

$$
\begin{aligned}
V = \{&\langle\text{SENTENCE}\rangle, \langle\text{NOUN-PHRASE}\rangle, \langle\text{VERB-PHRASE}\rangle,\\
&\langle\text{PREP-PHRASE}\rangle, \langle\text{CMPLX-NOUN}\rangle, \langle\text{CMPLX-VERB}\rangle,\\
&\langle\text{ARTICLE}\rangle, \langle\text{NOUN}\rangle, \langle\text{VERB}\rangle, \langle\text{PREP}\rangle\},
\end{aligned}
$$

and $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z}, \text{“ ”}\}$. The symbol “ ” is the blank symbol, placed invisibly after each word (a, boy, etc.), so the words won't run together.

Often we specify a grammar by writing down only its rules. We can identify the variables as the symbols that appear on the left-hand side of the rules and the terminals as the remaining symbols. By convention, the start variable is the variable on the left-hand side of the first rule.

EXAMPLES OF CONTEXT-FREE GRAMMARS

EXAMPLE 2.2

Consider grammar $G_3 = (\{S\}, \{\mathtt{a}, \mathtt{b}\}, R, S)$. The set of rules, R, is

$$S \rightarrow \mathtt{a}S\mathtt{b} \mid SS \mid \varepsilon.$$

This grammar generates strings such as abab, aaabbb, and aababb. You can see more easily what this language is if you think of a as a left parenthesis "(" and b as a right parenthesis ")". Viewed in this way, $L(G_3)$ is the language of all strings of properly nested parentheses. ■

EXAMPLE 2.3

Consider grammar $G_4 = (V, \Sigma, R, \langle\text{EXPR}\rangle)$.

V is $\{\langle\text{EXPR}\rangle, \langle\text{TERM}\rangle, \langle\text{FACTOR}\rangle\}$ and Σ is $\{\mathtt{a}, \mathtt{+}, \mathtt{x}, \mathtt{(}, \mathtt{)}\}$. The rules are

$$\begin{aligned}\langle\text{EXPR}\rangle &\rightarrow \langle\text{EXPR}\rangle\mathtt{+}\langle\text{TERM}\rangle \mid \langle\text{TERM}\rangle\\ \langle\text{TERM}\rangle &\rightarrow \langle\text{TERM}\rangle\mathtt{x}\langle\text{FACTOR}\rangle \mid \langle\text{FACTOR}\rangle\\ \langle\text{FACTOR}\rangle &\rightarrow \mathtt{(}\,\langle\text{EXPR}\rangle\,\mathtt{)} \mid \mathtt{a}\end{aligned}$$

The two strings a+axa and (a+a)xa can be generated with grammar G_4. The parse trees are shown in the following figure.

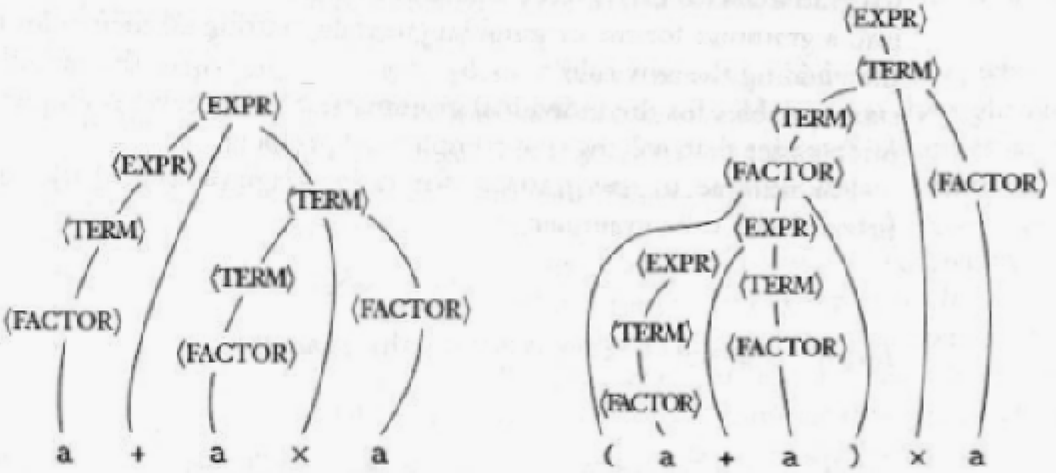


FIGURE 2.2
Parse trees for the strings a+axa and (a+a)xa

A compiler translates code written in a programming language into another form, usually one more suitable for execution. To do so the compiler extracts the meaning of the code to be compiled in a process called ***parsing***. One representation of this meaning is the parse tree for the code, in the context-free grammar for the programming language. We discuss an algorithm that parses context-free languages later in Theorem 7.14 and in Problem 7.38.

Grammar G_4 describes a fragment of a programming language concerned with arithmetic expressions. Observe how the parse trees in Figure 2.2 "group" the operations. The tree for a+a×a groups the × operator and its operands (the second two a's) together as one operand of the + operator. In the tree for (a+a)×a, the grouping is reversed. These groupings fit the standard precedence of multiplication before addition and the use of parentheses to override the standard precedence. Grammar G_4 is designed to capture these precedence relations.

■

DESIGNING CONTEXT-FREE GRAMMARS

As with the design of finite automata, discussed on page 41 in Section 1.1, the design of context-free grammars requires creativity. Indeed, context-free grammars are even trickier to construct than finite automata because we are more accustomed to programming a machine for specific tasks than we are to describing languages with grammars. The following techniques are helpful, singly or in combination, when you're faced with the problem of constructing a CFG.

First, many CFGs are the union of simpler CFGs. If you must construct a CFG for a CFL that you can break into simpler pieces, do so and then construct individual grammars for each piece. These individual grammars can be easily combined into a grammar for the original language by putting all their rules together and then adding the new rule $S \rightarrow S_1 \mid S_2 \mid \cdots \mid S_k$, where the variables S_i are the start variables for the individual grammars. Solving several simpler problems is often easier than solving one complicated problem.

For example, to get a grammar for the language $\{0^n1^n | n \geq 0\} \cup \{1^n0^n | n \geq 0\}$, first construct the grammar

$$S_1 \rightarrow 0S_11 \mid \varepsilon$$

for the language $\{0^n1^n | n \geq 0\}$ and the grammar

$$S_2 \rightarrow 1S_20 \mid \varepsilon$$

for the language $\{1^n0^n | n \geq 0\}$ and then add the rule $S \rightarrow S_1 \mid S_2$ to give the grammar

$$\begin{aligned} S &\rightarrow S_1 \mid S_2 \\ S_1 &\rightarrow 0S_11 \mid \varepsilon \\ S_2 &\rightarrow 1S_20 \mid \varepsilon \,. \end{aligned}$$

Second, constructing a CFG for a language that happens to be regular is easy if you can first construct a DFA for that language. You can convert any DFA into an equivalent CFG as follows. Make a variable R_i for each state q_i of the DFA. Add the rule $R_i \rightarrow aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA. Add the rule $R_i \rightarrow \varepsilon$ if q_i is an accept state of the DFA. Make R_0 the start variable of the grammar, where q_0 is the start state of the machine. Verify on your own that the resulting CFG generates the same language that the DFA recognizes.

Third, certain context-free languages contain strings with two substrings that are "linked" in the sense that a machine for such a language would need to remember an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other substring. This situation occurs in the language $\{0^n1^n | n \geq 0\}$ because a machine would need to remember the number of 0s in order to verify that it equals the number of 1s. You can construct a CFG to handle this situation by using a rule of the form $R \rightarrow uRv$, which generates strings wherein the portion containing the u's corresponds to the portion containing the v's.

Finally, in more complex languages, the strings may contain certain structures that appear recursively as part of other (or the same) structures. That situation occurs in the grammar that generates arithmetic expressions in Example 2.3. Any time the symbol a appears, an entire parenthesized expression might appear recursively instead. To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure may recursively appear.

AMBIGUITY

Sometimes a grammar can generate the same string in several different ways. Such a string will have several different parse trees and thus several different meanings. This result may be undesirable for certain applications, such as programming languages, where a given program should have a unique interpretation.

If a grammar generates the same string in several different ways, we say that the string is derived ***ambiguously*** in that grammar. If a grammar generates some string ambiguously we say that the grammar is ***ambiguous***.

For example, let's consider grammar G_5:

$$\langle\text{EXPR}\rangle \rightarrow \langle\text{EXPR}\rangle\texttt{+}\langle\text{EXPR}\rangle \mid \langle\text{EXPR}\rangle\texttt{x}\langle\text{EXPR}\rangle \mid \texttt{(}\,\langle\text{EXPR}\rangle\,\texttt{)} \mid \texttt{a}$$

This grammar generates the string a+axa ambiguously. The following figure shows the two different parse trees.

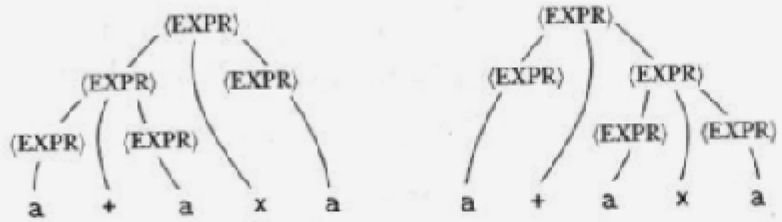


FIGURE 2.3
The two parse trees for the string `a+axa` in grammar G_5

This grammar doesn't capture the usual precedence relations and so may group the + before the × or vice versa. In contrast grammar G_4 generates exactly the same language, but every generated string has a unique parse tree. Hence G_4 is unambiguous, whereas G_5 is ambiguous.

Grammar G_2 on page 93 is another example of an ambiguous grammar. The sentence `the girl touches the boy with the flower` has two different derivations. In Exercise 2.8 you are asked to give the two parse trees and observe their correspondence with the two different ways to read that sentence.

Now we formalize the notion of ambiguity. When we say that a grammar generates a string ambiguously, we mean that the string has two different parse trees, not two different derivations. Two derivations may differ merely in the order in which they replace variables yet not in their overall structure. To concentrate on structure we define a type of derivation that replaces variables in a fixed order. A derivation of a string w in a grammar G is a ***leftmost derivation*** if at every step the leftmost remaining variable is the one replaced. The derivation on page 94 is a leftmost derivation.

DEFINITION 2.4

A string w is derived ***ambiguously*** in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ***ambiguous*** if it generates some string ambiguously.

Sometimes when we have an ambiguous grammar we can find an unambiguous grammar that generates the same language. Some context-free languages, however, can only be generated by ambiguous grammars. Such languages are called ***inherently ambiguous***. Problem 2.24 asks you to prove that the language $\{0^i1^j2^k \mid i = j \text{ or } j = k\}$ is inherently ambiguous.

CHOMSKY NORMAL FORM

When working with context-free grammars, it is often convenient to have them in simplified form. One of the simplest and most useful forms is called the Chomsky normal form. We will find Chomsky normal form useful when we are giving algorithms for working with context-free grammars in Chapters 4 and 7.

DEFINITION 2.5

A context-free grammar is in ***Chomsky normal form*** if every rule is of the form

$$A \rightarrow BC$$
$$A \rightarrow a$$

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition we permit the rule $S \rightarrow \varepsilon$, where S is the start variable.

THEOREM 2.6

Any context-free language is generated by a context-free grammar in Chomsky normal form.

PROOF IDEA We can convert any grammar G into Chomsky normal form. The conversion has several stages wherein rules that violate the conditions are replaced with equivalent ones that are satisfactory. First, we add a new start symbol. Then, we eliminate all ***ε rules*** of the form $A \rightarrow \varepsilon$. We also eliminate all ***unit rules*** of the form $A \rightarrow B$. In both cases the grammar is then patched up to be sure that it still generates the same language. Finally, we convert the remaining rules into the proper form.

PROOF First, we add a new start symbol S_0 and the rule $S_0 \rightarrow S$, where S was the original start symbol. This change guarantees that the start symbol doesn't occur on the right-hand side of a rule.

Second, we take care of all ε rules. We remove an ε-rule $A \rightarrow \varepsilon$, where A is not the start variable. Then for each occurrence of an A on the right-hand side of a rule, we add a new rule with that occurrence deleted. In other words, if $R \rightarrow uAv$ is a rule in which u and v are strings of variables and terminals, we add rule $R \rightarrow uv$. We do so for each *occurrence* of an A, so the rule $R \rightarrow uAvAw$ causes us to add $R \rightarrow uvAw$, $R \rightarrow uAvw$, and $R \rightarrow uvw$. If we have the rule $R \rightarrow A$, we add $R \rightarrow \varepsilon$ unless we had previously removed the rule $R \rightarrow \varepsilon$. We repeat these steps until we eliminate all ε rules not involving the start variable.

Third, we handle all unit rules. We remove a unit rule $A \rightarrow B$. Then, whenever a rule $B \rightarrow u$ appears, we add the rule $A \rightarrow u$ unless this was a unit rule previously removed. As before, u is a string of variables and terminals. We repeat these steps until we eliminate all unit rules.

Finally, we convert all remaining rules into the proper form. We replace each rule $A \rightarrow u_1u_2 \cdots u_k$ where $k \geq 3$ and each u_i is a variable or terminal symbol, with the rules $A \rightarrow u_1A_1$, $A_1 \rightarrow u_2A_2$, $A_2 \rightarrow u_3A_3$, … , $A_{k-2} \rightarrow u_{k-1}u_k$. The A_i's are new variables. If $k \geq 2$, we replace any terminal u_i in the preceding rule(s) with the new variable U_i and add the rule $U_i \rightarrow u_i$.

EXAMPLE 2.7

Let G_6 be the following CFG and convert it to Chomsky normal form by using the conversion procedure just given. The following series of grammars illustrates the steps in the conversion. Rules shown in bold have been just added. Rules shown in gray have just been removed.

1. The original CFG G_6 is shown on the left. The result of applying the first step to make a new start symbol appears on the right.

$$\begin{aligned} S &\to ASA \mid \mathtt{a}B \\ A &\to B \mid S \\ B &\to \mathtt{b} \mid \varepsilon \end{aligned}$$

$$\begin{aligned} \boldsymbol{S_0} &\boldsymbol{\to S} \\ S &\to ASA \mid \mathtt{a}B \\ A &\to B \mid S \\ B &\to \mathtt{b} \mid \varepsilon \end{aligned}$$

2. Remove ε rules $B \to \varepsilon$, shown on the left, and $A \to \varepsilon$, shown on the right.

$$\begin{aligned} S_0 &\to S \\ S &\to ASA \mid \mathtt{a}B \mid \boldsymbol{\mathtt{a}} \\ A &\to B \mid S \mid \boldsymbol{\varepsilon} \\ B &\to \mathtt{b} \mid \varepsilon \end{aligned}$$

$$\begin{aligned} S_0 &\to S \\ S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \mid \boldsymbol{S} \\ A &\to B \mid S \mid \varepsilon \\ B &\to \mathtt{b} \end{aligned}$$

3a. Remove unit rules $S \to S$, shown on the left, and $S_0 \to S$, shown on the right.

$$\begin{aligned} S_0 &\to S \\ S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \mid S \\ A &\to B \mid S \\ B &\to \mathtt{b} \end{aligned}$$

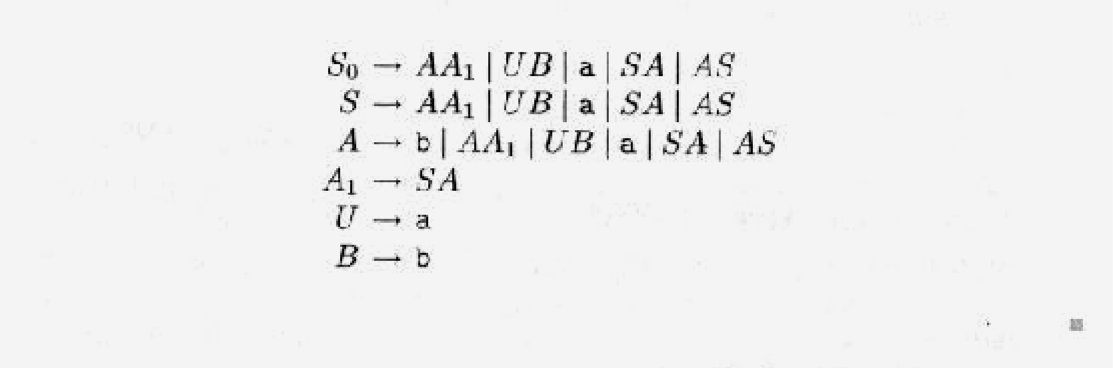
$$\begin{aligned} S_0 &\to S \mid \boldsymbol{ASA} \mid \boldsymbol{\mathtt{a}B} \mid \boldsymbol{\mathtt{a}} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \\ S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\ A &\to B \mid S \\ B &\to \mathtt{b} \end{aligned}$$

3b. Remove unit rules $A \to B$ and $A \to S$.

$$\begin{aligned} S_0 &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\ S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\ A &\to B \mid S \mid \boldsymbol{\mathtt{b}} \\ B &\to \mathtt{b} \end{aligned}$$

$$\begin{aligned} S_0 &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\ S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\ A &\to S \mid \mathtt{b} \mid \boldsymbol{ASA} \mid \boldsymbol{\mathtt{a}B} \mid \boldsymbol{\mathtt{a}} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \\ B &\to \mathtt{b} \end{aligned}$$

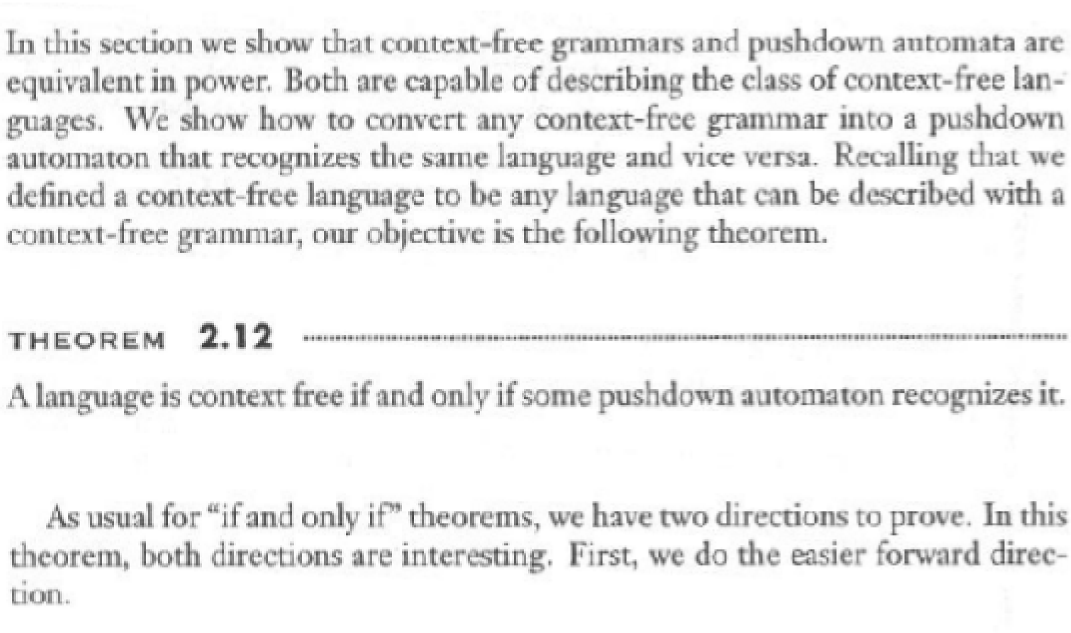
4. Convert the remaining rules into the proper form by adding additional variables and rules. The final grammar in Chomsky normal form is equivalent to G_6 and appears as follows. (Actually the procedure given in Theorem 2.6 produces several variables U_i along with several rules $U_i \to \mathtt{a}$. We simplified the resulting grammar by using a single variable U and rule $U \to \mathtt{a}$.)



$$\begin{aligned}
S_0 &\rightarrow AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
S &\rightarrow AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
A &\rightarrow \mathtt{b} \mid AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
A_1 &\rightarrow SA \\
U &\rightarrow \mathtt{a} \\
B &\rightarrow \mathtt{b}
\end{aligned}$$

■

PDA EQUIVALENCE WITH CONTEXT FREE GRAMMARS



In this section we show that context-free grammars and pushdown automata are equivalent in power. Both are capable of describing the class of context-free languages. We show how to convert any context-free grammar into a pushdown automaton that recognizes the same language and vice versa. Recalling that we defined a context-free language to be any language that can be described with a context-free grammar, our objective is the following theorem.

THEOREM 2.12

A language is context free if and only if some pushdown automaton recognizes it.

As usual for "if and only if" theorems, we have two directions to prove. In this theorem, both directions are interesting. First, we do the easier forward direction.

LEMMA 2.13

If a language is context free, then some pushdown automaton recognizes it.

PROOF IDEA Let A be a CFL. From the definition we know that A has a CFG, G, generating it. We show how to convert G into an equivalent PDA, which we call P.

The PDA P that we now describe will work by accepting its input w, if G generates that input, by determining whether there is a derivation for w. Recall that a derivation is simply the sequence of substitutions made as a grammar generates a string. Each step of the derivation yields an ***intermediate string*** of variables and terminals. We design P to determine whether some series of substitutions using the rules of G can lead from the start variable to w.

One of the difficulties in testing whether there is a derivation for w is in figuring out which substitutions to make. The PDA's nondeterminism allows it to guess the sequence of correct substitutions. At each step of the derivation one of the rules for a particular variable is selected nondeterministically and used to substitute for that variable.

The PDA P begins by writing the start variable on its stack. It goes through a series of intermediate strings, making one substitution after another. Eventually it may arrive at a string that contains only terminal symbols, meaning that it has derived a string using the grammar. Then P accepts if this string is identical to the string it has received as input.

Implementing this strategy on a PDA requires one additional idea. We need to see how the PDA stores the intermediate strings as it goes from one to another. Simply using the stack for storing each intermediate string is tempting. However, that doesn't quite work because the PDA needs to find the variables in the intermediate string and make substitutions. The PDA can access only the top symbol on the stack and that may be a terminal symbol instead of a variable. The way around this problem is to keep only *part* of the intermediate string on the stack: the symbols starting with the first variable in the intermediate string. Any terminal symbols appearing before the first variable are matched immediately with symbols in the input string. The following figure shows the PDA P.

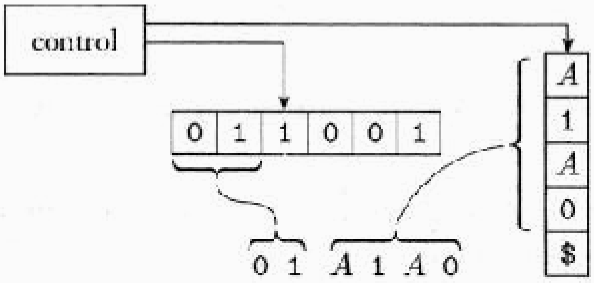


FIGURE 2.9
P representing the intermediate string $01A1A0$

The following is an informal description of P.

1. Place the marker symbol \$ and the start variable on the stack.
2. Repeat the following steps forever.
   - a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.
   - b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
   - c. If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read.

LEMMA 2.15

If a pushdown automaton recognizes some language, then it is context free.

PROOF IDEA We have a PDA P, and we want to make a CFG G that generates all the strings that P accepts. In other words, G should generate a string if that string causes the PDA to go from its start state to an accept state.

To achieve this outcome we design a grammar that does somewhat more. For each pair of states p and q in P the grammar will have a variable A_{pq}. This variable generates all the strings that can take P from p with an empty stack to q with an empty stack. Observe that such strings can also take P from p to q, regardless of the stack contents at p, leaving the stack at q in the same condition as it was at p.

First, we simplify our task by modifying P slightly to give it the following three features.

1. It has a single accept state, q_{accept}.
2. It empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack (a *pop* move), but does not do both at the same time.

Giving P features 1 and 2 is easy. To give it feature 3, we replace each transition that simultaneously pops and pushes with a two transition sequence that goes through a new state, and we replace each transition that neither pops nor pushes with a two transition sequence that pushes then pops an arbitrary stack symbol.

To design G so that A_{pq} generates all strings that take P from p to q, starting and ending with an empty stack, we must understand how P operates on these strings. For any such string x, P's first move on x must be a push, because every move is either a push or a pop and P can't pop an empty stack. Similarly the last move on x must be a pop, because the stack ends up empty.

Two possibilities occur during P's computation on x. Either the symbol popped at the end is the symbol that was pushed at the beginning, or not. If so, the stack is empty only at the beginning and end of P's computation on x. If not, the initially pushed symbol must get popped at some point before the end of x and thus the stack becomes empty at this point. We simulate the former possibility with the rule $A_{pq} \rightarrow aA_{rs}b$ where a is the input symbol read at the first move, b is the symbol read at the last move, r is the state following p, and s the state preceding q. We simulate the latter possibility with the rule $A_{pq} \rightarrow A_{pr}A_{rq}$, where r is the state when the stack becomes empty.

We have just proved that pushdown automata recognize the class of context-free languages. This proof allows us to establish a relationship between the regular languages and the context-free languages. Because every regular language is recognized by a finite automaton and every finite automaton is automatically a pushdown automaton that simply ignores its stack, we now know that every regular language is also a context-free language.

COROLLARY 2.18

Every regular language is context free.

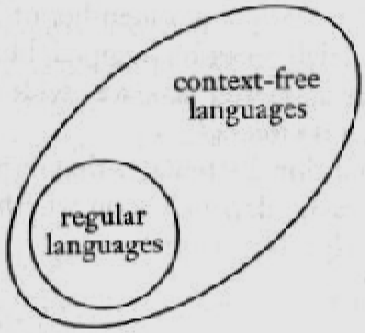


FIGURE 2.15
Relationship of the regular and context-free languages

2.3

NON-CONTEXT-FREE LANGUAGES

In this section we present a technique for proving that certain languages are not context free. Recall that in Section 1.4 we introduced the pumping lemma for showing that certain languages are not regular. Here we present a similar pumping lemma for context-free languages. It states that every context-free language has a special value called the ***pumping length*** such that all longer strings in the language can be "pumped." This time the meaning of ***pumped*** is a bit more complex. It means that the string can be divided into five parts so that the second and the fourth parts may be repeated together any number of times and the resulting string still remains in the language.

THE PUMPING LEMMA FOR CONTEXT-FREE LANGUAGES

THEOREM 2.19

Pumping lemma for context-free languages If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions:

1. For each $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

When s is being divided into $uvxyz$, condition 2 says that either v or y is not the empty string. Otherwise the theorem would be trivially true. Condition 3 states that the pieces v, x, and y together have length at most p. This technical condition sometimes is useful in proving that certain languages are not context free.

PROOF IDEA Let A be a CFL and let G be a CFG that generates it. We must show that any sufficiently long string s in A can be pumped and remain in A. The idea behind this approach is simple.

Let s be a very long string in A. (We make clear later what we mean by "very long.") Because s is in A, it is derivable from G and so has a parse tree. The parse tree for s must be very tall because s is very long. That is, the parse tree must contain some long path from the start variable at the root of the tree to one of the terminal symbols at a leaf. On this long path some variable symbol R must repeat because of the pigeonhole principle. As the following figure shows, this repetition allows us to replace the subtree under the second occurrence of R with the subtree under the first occurrence of R and still get a legal parse tree.

Therefore we may cut s into five pieces $uvxyz$ as the figure indicates, and we may repeat the second and fourth pieces and obtain a string still in the language. In other words, uv^ixy^iz is in A for any $i \geq 0$.

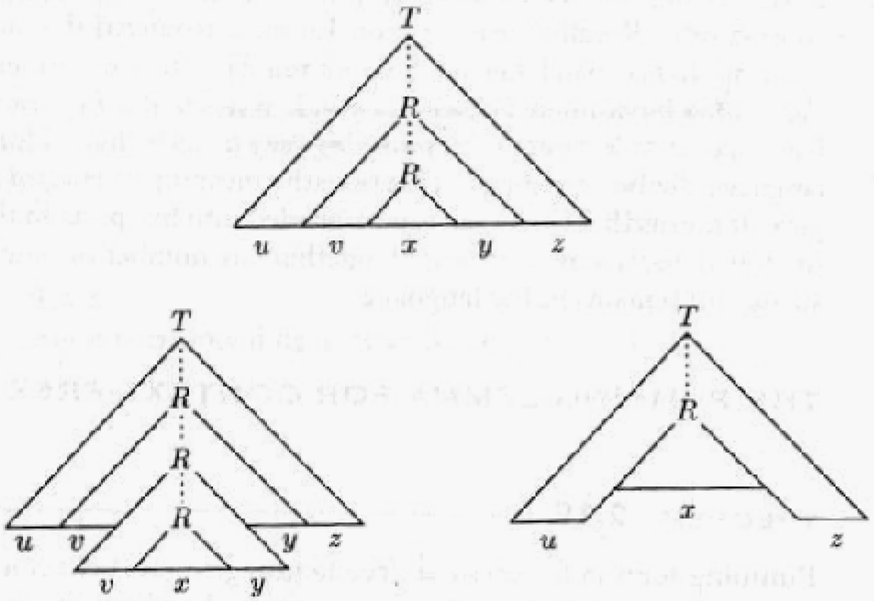


FIGURE 2.16
Surgery on parse trees

Let's now turn to the details to obtain all three conditions of the pumping lemma. We also show how to calculate the pumping length p.

PROOF Let G be a CFG for CFL A. Let b be the maximum number of symbols in the right-hand side of a rule. We may assume that $b \geq 2$. In any parse tree using this grammar we know that a node can have no more than b children. In other words at most b leaves are 1 step from the start variable; at most b^2 leaves are at most 2 steps from the start variable; and at most b^h leaves are at most h steps from the start variable. So, if the height of the parse tree is at most h, the length of the string generated is at most b^h.

Let $|V|$ be the number of variables in G. We set p to be $b^{|V|+2}$. Because $b \geq 2$, we know that $p > b^{|V|+1}$, so a parse tree for any string in A of length at least p requires height at least $|V|+2$.

Suppose that s is a string in A of length at least p. We now show how to pump s. Let τ be a parse tree for s. If s has several parse trees, we choose τ to be a parse tree that has the smallest number of nodes. As $|s| \geq p$, we know that τ has height at least $|V|+2$, so the longest path in τ has length at least $|V|+2$. This path must have at least $|V|+1$ variables because only the leaf is a terminal. With G having only $|V|$ variables, some variable R appears more than once on the path. For convenience later, we select R to be a variable that repeats among the lowest $|V|+1$ variables on this path.

We divide s into $uvxyz$ according to Figure 2.16. Each occurrence of R has a subtree under it, generating a part of the string s. The upper occurrence of R has a larger subtree and generates vxy, whereas the lower occurrence generates just x with a smaller subtree. Both of these subtrees are generated by the same variable, so we may substitute one for the other and still obtain a valid parse tree. Replacing the smaller by the larger repeatedly gives parse trees for the strings uv^ixy^iz at each $i > 1$. Replacing the larger by the smaller generates the string uxz. That establishes condition 1 of the lemma. We now turn to conditions 2 and 3.

To get condition 2 we must be sure that both v and y are not ε. If they were, the parse tree obtained by substituting the smaller subtree for the larger would have fewer nodes than τ does and would still generate s. This result isn't possible because we had already chosen τ to be a parse tree for s with the smallest number of nodes. That is the reason for selecting τ in this way.

In order to get condition 3 we need to be sure that vxy has length at most p. In the parse tree for s the upper occurrence of R generates vxy. We chose R so that both occurrences fall within the bottom $|V|+1$ variables on the path, and we chose the longest path in the parse tree, so the subtree where R generates vxy is at most $|V|+2$ high. A tree of this height can generate a string of length at most $b^{|V|+2} = p$.

EXAMPLE 2.20

Use the pumping lemma to show that the language $B = \{a^nb^nc^n | n \geq 0\}$ is not context free.

We assume that B is a CFL and obtain a contradiction. Let p be the pumping length for B that is guaranteed to exist by the pumping lemma. Select the string $s = a^pb^pc^p$. Clearly s is a member of B and of length at least p. The pumping lemma states that s can be pumped, but we show that it cannot. In other words, we show that no matter how we divide s into $uvxyz$, one of the three conditions of the lemma is violated.

First, condition 2 stipulates that either v or y is nonempty. Then we consider one of two cases, depending on whether substrings v and y contain more than one type of alphabet symbol.

1. When both v and y contain only one type of alphabet symbol, v does not contain both a's and b's or both b's and c's, and the same holds for y. In this case the string uv^2xy^2z cannot contain equal numbers of a's, b's, and c's. Therefore it cannot be a member of B. That violates condition 1 of the lemma and is thus a contradiction.
2. When either v or y contain more than one type of symbol uv^2xy^2z may

contain equal numbers of the three alphabet symbols but won't contain them in the correct order. Hence it cannot be a member of B and a contradiction occurs.

One of these cases must occur. Because both cases result in a contradiction, a contradiction is unavoidable. So the assumption that B is a CFL must be false. Thus we have proved that B is not a CFL. ∎

EXAMPLE 2.21

Let $C = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid 0 \leq i \leq j \leq k\}$. We use the pumping lemma to show that C is not a CFL. This language is similar to language B in Example 2.20, but proving that it is not context free is a bit more complicated.

Assume that C is a CFL and obtain a contradiction. Let p be the pumping length given by the pumping lemma. We use the string $s = \mathtt{a}^p\mathtt{b}^p\mathtt{c}^p$ that we used earlier, but this time we must "pump down" as well as "pump up." Let $s = uvxyz$ and again consider the two cases that occurred in Example 2.20.

1. When both v and y contain only one type of alphabet symbol, v does not contain both a's and b's or both b's and c's, and the same holds for y. Note that the reasoning used previously in case 1 no longer applies. The reason is that C contains strings with unequal numbers of a's, b's, and c's as long as the numbers are not decreasing. We must analyze the situation more carefully to show that s cannot be pumped. Observe that because v and y contain only one type of alphabet symbol, one of the symbols a, b, or c doesn't appear in v or y. We further subdivide this case into three subcases according to which symbol does not appear.
   - **a.** The a's do not appear. Then we try pumping down to obtain the string $uv^0xy^0z = uxz$. That contains the same number of a's as s does, but it contains fewer b's or fewer c's. Therefore it is not a member of C, and a contradiction occurs.
   - **b.** The b's do not appear. Then either a's or c's must appear in v or y because both can't be the empty string. If a's appear, the string uv^2xy^2z contains more a's than b's, so it is not in C. If c's appear, the string uv^0xy^0z contains more b's than c's, so it is not in C. Either way a contradiction occurs.
   - **c.** The c's do not appear. Then the string uv^2xy^2z contains more a's or more b's than c's, so it is not in C, and a contradiction occurs.
2. When either v or y contain more than one type of symbol, uv^2xy^2z will not contain the symbols in the correct order. Hence it cannot be a member of C, and a contradiction occurs.

Thus we have shown that s cannot be pumped in violation of the pumping lemma and that C is not context free. ∎

EXAMPLE 2.22

Let $D = \{ww|w \in \{0,1\}^*\}$. Use the pumping lemma to show that D is not a CFL. Assume that D is a CFL and obtain a contradiction. Let p be the pumping length given by the pumping lemma.

This time choosing string s is less obvious. One possibility is the string 0^p10^p1. It is a member of D and has length greater than p, so it appears to be a good candidate. But this string *can* be pumped by dividing it as follows, so it is not adequate for our purposes.

$$\overbrace{\underbrace{000\cdots000}_{u}\;\underbrace{0}_{v}\;\underbrace{1}_{x}}^{0^p1}\;\overbrace{\underbrace{0}_{y}\;\underbrace{000\cdots0001}_{z}}^{0^p1}$$

Let's try another candidate for s. Intuitively, the string $0^p1^p0^p1^p$ seems to capture more of the "essence" of the language D than the previous candidate did. In fact, we can show that this string does work, as follows.

We show that the string $s = 0^p1^p0^p1^p$ cannot be pumped. This time we use condition 3 of the pumping lemma to restrict the way that s can be divided. It says that we can pump s by dividing $s = uvxyz$, where $|vxy| \leq p$.

First, we show that the substring vxy must straddle the midpoint of s. Otherwise, if the substring occurs only in the first half of s, pumping s up to uv^2xy^2z moves a 1 into the first position of the second half, and so it cannot be of the form ww. Similarly, if vxy occurs in the second half of s, pumping s up to uv^2xy^2z moves a 0 into the last position of the first half, and so it cannot be of the form ww.

But if the substring vxy straddles the midpoint of s, when we try to pump s down to uxz it has the form $0^p1^i0^j1^p$, where i and j cannot both be p. This string is not of the form ww. Thus s cannot be pumped, and D is not a CFL.