6. CONTEXT-FREE GRAMMARS

6.1 Examples and Definitions

- Many languages can be described by recursive definitions.
- A *context-free grammar* is a related way of recursively defining a language.
- **Example 6.1:** Using Grammar Rules to Describe a Language

  The language $L = \{a, b\}^*$ can be defined recursively as follows:

  1. $\Lambda \in L$.
  2. For any $S \in L$, $Sa \in L$.
  3. For any $S \in L$, $Sb \in L$.
  4. No other strings are in L.

  If S is thought of as a *variable* representing an arbitrary element of L, these rules can be rewritten as follows:

  $S \rightarrow \Lambda$
  $S \rightarrow Sa$
  $S \rightarrow Sb$

  The symbol $\rightarrow$ is used to indicate that a variable is replaced by a string.

  For two strings α and β, the notation $\alpha \Rightarrow \beta$ means that β can be obtained by applying one of these rules to a single variable in the string α.

  Using this notation, we can write

  $S \Rightarrow Sa \Rightarrow Sba \Rightarrow Sbba \Rightarrow \Lambda bba = bba$

  to describe the sequence of steps used to obtain, or *derive,* the string *bba*.

  The derivation comes to an end when S is replaced by a string of alphabet symbols (in this case Λ).

  The symbol | ("or") can be used to combine rules:

  $S \rightarrow \Lambda \mid Sa \mid Sb$

Examples and Definitions (Continued)

An alternative definition of L:

1. $\Lambda \in L$.
2. $a \in L$.
3. $b \in L$.
4. For every x and y in L, $xy \in L$.
5. No other strings are in L.

Using the new notation, the “grammar rules” could be written as follows:

$S \to \Lambda \mid a \mid b \mid SS$

With this approach there is more than one way to obtain the string *bba*. Two derivations are shown below:

$S \Rightarrow SS \Rightarrow bS \Rightarrow bSS \Rightarrow bbS \Rightarrow bba$
$S \Rightarrow SS \Rightarrow Sa \Rightarrow SSa \Rightarrow bSa \Rightarrow bba$

- **Example 6.2:** The Language $\{a^n b^n \mid n \geq 0\}$

  The grammar rules

  $S \to aSb \mid \Lambda$

  are another way of describing the language L defined as follows:

  1. $\Lambda \in L$.
  2. For any $x \in L$, $axb \in L$.
  3. Nothing else is in L.

  The language L is easily seen to be the nonregular language $\{a^n b^n \mid n \geq 0\}$.

Examples and Definitions (Continued)

- **Example 6.3:** Palindromes

  The language *pal* of palindromes over the alphabet $\{a, b\}$ can be defined recursively as follows:

  1. $\Lambda, a, b \in pal$.
  2. For any $S \in pal$, aSa and bSb are in *pal*.
  3. No other strings are in *pal*.

  pal can also be described by a context-free grammar with the following rules:

  $S \rightarrow \Lambda \mid a \mid b \mid aSa \mid bSb$

  Let N be the complement of *pal* (the set of all nonpalindromes over $\{a, b\}$).

  N also obeys rule 2: For any nonpalindrome x, both axa and bxb are nonpalindromes. However, a recursive definition of N cannot be as simple as this definition of *pal*, because there is no finite set of basis elements.

  A nonpalindrome has a central portion that starts with one symbol and ends with the opposite symbol; the string between those two can be anything. Using this observation, N can be defined recursively:

  1. For any $A \in \{a, b\}^*$, aAb and bAa are in N.
  2. For any $S \in N$, aSa and bSb are in N.
  3. No other strings are in N.

  A context-free grammar describing N can be obtained by introducing a second variable A, representing an arbitrary element of $\{a, b\}^*$:

  $S \rightarrow aAb \mid bAa \mid aSa \mid bSb$
  $A \rightarrow \Lambda \mid Aa \mid Ab$

  A derivation of the nonpalindrome *abbaaba*:

  $S \Rightarrow aSa \Rightarrow abSba \Rightarrow abbAaba \Rightarrow abbAaaba \Rightarrow abb\Lambda aaba = abbaaba$

  It is common, and often necessary, to include several variables in a context-free grammar describing a language L.

  There will still be one special variable that represents an arbitrary string in L. It is customary to denote this one by S (the *start* variable).

Definition of a Context-Free Grammar

- **Definition 6.1:** A context-free grammar (CFG) is a 4-tuple $G = (V, \Sigma, S, P)$, where V and Σ are disjoint finite sets, S is an element of V, and P is a finite set of formulas of the form $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.

  The elements of V are called *variables,* or *nonterminal* symbols, and those of the alphabet Σ are called *terminal* symbols, or *terminals*. S is called the *start symbol;* and the elements of P are called *grammar rules*, or *productions*.

- Suppose $G = (V, \Sigma, S, P)$ is a CFG. The symbol $\Rightarrow$ is used for steps in a derivation. Sometimes it is useful to indicate explicitly that the derivation is with respect to G, and in this case we write $\Rightarrow_G$. Writing

  $\alpha \Rightarrow_G \beta$

  means that the string β can be obtained from the string α by replacing some variable that appears on the left side of a production in G by the corresponding right side, or that

  $\alpha = \alpha_1 A \alpha_2$
  $\beta = \alpha_1 \gamma \alpha_2$

  and one of the productions in G is $A \to \gamma$. In this case, we say that α derives β, or β is derived from α, in one step.

- More generally, we write

  $\alpha \Rightarrow_G^* \beta$

  (and shorten it to $\alpha \Rightarrow^* \beta$ if it is clear what grammar is involved) if α derives β in zero or more steps; in other words, either $\alpha = \beta$, or there exist an integer $k \geq 1$ and strings $\alpha_0, \alpha_1, \ldots, \alpha_k$, with $\alpha_0 = \alpha$ and $\alpha_k = \beta$, so that $\alpha_i \Rightarrow_G \alpha_{i+1}$ for every i with $0 \leq i \leq k - 1$.

- **Definition 6.2:** Let $G = (V, \Sigma, S, P)$ be a CFG. The language generated by G is

  $L(G) = \{x \in \Sigma^* \mid S \Rightarrow_G^* x\}$

  A language L is a *context-free language* (CFL) if there is a CFG G so that $L = L(G)$.

Examples of Context-Free Grammars

- **Example 6.4:** The Language of Algebraic Expressions

  The following context-free grammar generates algebraic expressions that can be formed from the four binary operators +, –, *, and /, left and right parentheses, and the single identifier a:

  $S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid a$

  The string $a + (a * a) / a - a$ can be obtained from the following derivation:

  $$\begin{aligned} S &\Rightarrow S - S \Rightarrow S + S - S \Rightarrow a + S - S \Rightarrow a + S / S - S \\ &\Rightarrow a + (S) / S - S \Rightarrow a + (S * S) / S - S \Rightarrow a + (a * S) / S - S \\ &\Rightarrow a + (a * a) / S - S \Rightarrow a + (a * a) / a - S \Rightarrow a + (a * a) / a - a \end{aligned}$$

  There are many other derivations as well, including the following:

  $$\begin{aligned} S &\Rightarrow S / S \Rightarrow S + S / S \Rightarrow a + S / S \Rightarrow a + (S) / S \\ &\Rightarrow a + (S * S) / S \Rightarrow a + (a * S) / S \Rightarrow a + (a * a) / S \\ &\Rightarrow a + (a * a) / S - S \Rightarrow a + (a * a) / a - S \Rightarrow a + (a * a) / a - a \end{aligned}$$

  The first derivation seems more natural than the second, because it indicates that the original expression is the difference of two other expressions. The second derivation interprets the expression as a quotient.

  This context-free grammar does not incorporate standard conventions having to do with the precedence of operators and left-to-right associativity.

  It is often desirable to select, if possible, a CFG in which a string can have only one derivation (except for differences between the order in which variables are chosen for replacement). Section 6.4 discusses this issue in more detail.

Examples of Context-Free Grammars (Continued)

- **Example 6.5:** The Syntax of Programming Languages

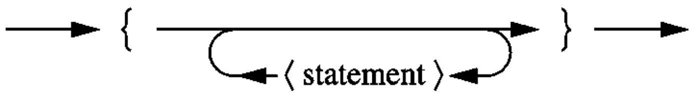
  To a large extent, context-free grammars can be used to describe the overall syntax of programming languages.

  Sample grammar rules for statements in the C language:

  <statement> → … | <*if*-statement> | <*for*-statement> | …
  <*if*-statement> → if (<expression>) <statement>
  <*for*-statement> → for (<expression>; <expression>; <expression>) <statement>

  An *if* statement or *for* statement may contain a single "inner" statement. To allow multiple statements, the "inner" statement will need to be *compound*.

  A compound statement consists of zero or more statements enclosed within {}. The syntax of a compound statement could be specified by grammar rules or by a *syntax diagram:*



Examples of Context-Free Grammars (Continued)

- **Example 6.6:** Grammar Rules for English

  Context-free grammars can be used to describe the syntax of simple English sentences. The following rules describe the syntax of many sentences:

  ```
  <declarative sentence> → <subject phrase> <verb phrase> <object> |
                           <subject phrase> <verb phrase>
  ```

  Unfortunately, many sentences generated using these rules will be syntactically correct but semantically meaningless.

  Consider the productions

  ```
  <declarative sentence> → <subject> <verb> <object>
               <subject> → <proper noun>
           <proper noun> → John | Jane
                  <verb> → reminded
                <object> → <proper noun> | <reflexive pronoun>
     <reflexive pronoun> → himself | herself
  ```

  These rules generate sentences such as “John reminded herself” and “Jane reminded himself.” These could be eliminated by introducing productions such as

  ```
  <declarative sentence> → <masculine noun> <verb> <masculine reflexive pronoun>
  ```

  A slightly more subtle problem is “Jane reminded Jane.” There is no obvious way to prohibit this sentence without also prohibiting “Jane reminded John.” Distinguishing between these sentences requires using *context*, which is exactly what a context-free grammar does not allow.

6.2 More Examples

- Proving that a CFG generates a language requires showing that (a) every string in the language can be derived from the grammar and (b) no other string can be derived from the grammar.

  One of these statements is often easy to show, while the other is much harder.

- **Example 6.7:** A CFG for $\{x \mid n_0(x) = n_1(x)\}$

  Consider the language

  $L = \{x \in \{0, 1\}^* \mid n_0(x) = n_1(x)\}$

  Clearly, $\Lambda \in L$. Given a string x in L, a longer string in L can be obtained by adding one 0 and one 1. One possibility is to add one symbol at each end, which suggests the productions

  $S \to \Lambda \mid 0S1 \mid 1S0$

  Strings that begin and end with the same symbol cannot be obtained from these productions, however. Adding the production $S \to SS$ solves this problem:

  $S \to \Lambda \mid 0S1 \mid 1S0 \mid SS$

  Let G be the CFG containing these productions. It is easy to see that $L(G) \subseteq L$. The converse, $L \subseteq L(G)$, will need to be proved.

  Let $d(x)$ be defined as follows:

  $d(x) = n_0(x) - n_1(x)$

  The goal is to prove that, for any string x with $d(x) = 0$, $x \in L(G)$. The proof is by mathematical induction on $|x|$.

More Examples (Continued)

Basis step. Assume that $|x| = 0$ and $d(x) = 0$. Then $x \in L(G)$ because $S \to \Lambda$ is a production in G.

Induction hypothesis. $k \geq 0$ and for any y with $|y| \leq k$ and $d(y) = 0$, $y \in L(G)$.

Statement to be shown in induction step. If $|x| = k + 1$ and $d(x) = 0$, then $x \in L(G)$.

Proof of induction step. *Case 1:* If x begins with 0 and ends with 1, then $x = 0y1$ for some string y satisfying $d(y) = 0$. By the induction hypothesis, $y \in L(G)$. Therefore x can be derived as follows:

$S \Rightarrow 0S1 \Rightarrow^* 0y1 = x$

Case 2: If x begins with 1 and ends with 0, the production $S \to 1S0$ is used to begin the derivation.

Case 3: x begins and ends with the same symbol. Since $d(x) = 0$, x has length at least 2. Suppose that $x = 0y0$ for some string y. A derivation of x would have to start with the production $S \to SS$.

In order to prove that there is such a derivation, it is necessary to show that $x = wz$, where w and z are shorter strings that can both be derived from S—in other words, x has a prefix w so that $0 < |w| < |x|$ and $d(w) = 0$.

Consider $d(w)$ for prefixes w of x. The shortest nonnull prefix is 0, and $d(0) = 1$; the longest prefix shorter than x is $0y$, and $d(0y) = -1$. Furthermore, the d-value of a prefix changes by 1 each time an extra symbol is added. It follows that there must be a prefix w, longer than 0 and shorter than $0y$, with $d(w) = 0$.

The case when $x = 1y1$ is almost the same.

More Examples (Continued)

- **Example 6.8:** Another CFG for $\{x \mid n_0(x) = n_1(x)\}$

  Again, let $L = \{x \in \{0, 1\}^* \mid n_0(x) = n_1(x)\}$.

  One way to obtain an element of L is to add a single symbol to a string that has one extra occurrence of the opposite symbol. Every element of L can be obtained this way and in fact can be obtained by adding this extra symbol at the beginning.

  Let the variables A and B represent strings with an extra 0 and an extra 1, and let these languages be denoted by L_0 and L_1:

  $L_0 = \{x \in \{0, 1\}^* \mid n_0(x) = n_1(x) + 1\} = \{x \in \{0, 1\}^* \mid d(x) = 1\}$
  $L_1 = \{x \in \{0, 1\}^* \mid n_1(x) = n_0(x) + 1\} = \{x \in \{0, 1\}^* \mid d(x) = -1\}$

  where d is the function defined by $d(x) = n_0(x) - n_1(x)$.

  Clearly the following productions will be needed:

  $S \rightarrow 0B \mid 1A \mid \Lambda$

  It is also easy to find one production for each of the variables A and B. If a string in L_0 begins with 0, or if a string in L_1 begins with 1, then the remainder is an element of L. Thus, it is appropriate to add the productions

  $A \rightarrow 0S \quad B \rightarrow 1S$

  What remains are the strings in L_0 that start with 1 and the strings in L_1 that start with 0. In the first case, if $x = 1y$ and $x \in L_0$, then y has two more 0's than 1's. If it y could be written as the concatenation of two strings, each with one extra 0, then the A-productions could be completed by adding $A \rightarrow 1AA$, and B could be handled similarly.

  If $d(x) = 1$ and $x = 1y$, then Λ is a prefix of y with $d(\Lambda) = 0$, and y itself is a prefix of y with $d(y) = 2$. Therefore, there is some intermediate prefix w of y with $d(w) = 1$, and $y = wz$ where $w, z \in L_0$.

  It is fairly easy to prove by induction that the following grammar generates L:

  $S \rightarrow 0B \mid 1A \mid \Lambda$
  $A \rightarrow 0S \mid 1AA$
  $B \rightarrow 1S \mid 0BB$

More Examples (Continued)

- The following theorem provides three simple ways of obtaining new CFLs from languages that are known to be context-free.

- **Theorem 6.1.** If L_1 and L_2 are context-free languages, then the languages $L_1 \cup L_2$, L_1L_2, and L_1* are also CFLs.

  Proof: The proof is constructive: Starting with CFGs

  $G_1 = (V_1, \Sigma, S_1, P_1)$ and $G_2 = (V_2, \Sigma, S_2, P_2)$

  generating L_1 and L_2, respectively, we show how to construct a new CFG for each of the three cases.

  A grammar $G_u = (V_u, \Sigma, S_u, P_u)$ generating $L_1 \cup L_2$. First rename the elements of V_2 if necessary so that $V_1 \cap V_2 = \varnothing$, and then define

  $V_u = V_1 \cup V_2 \cup \{S_u\}$
  $P_u = P_1 \cup P_2 \cup \{S_u \rightarrow S_1 \mid S_2\}$

  where S_u is a new symbol not in V_1 or V_2.

  If x is in either L_1 or L_2, then $S_u \Rightarrow^* x$ in the grammar G_u, because a derivation can be started with either $S_u \rightarrow S_1$ or $S_u \rightarrow S_2$ and continued with the derivation of x in G_1 or G_2. Therefore,

  $L_1 \cup L_2 \subseteq L(G_u)$

  If x is derivable from S_u in G_u, the first step in any derivation must be

  $S_u \Rightarrow S_1$ or $S_u \Rightarrow S_2$

  In the first case, all subsequent productions used must be productions in G_1, because no variables in V_2 are involved, and thus $x \in L_1$; in the second case, $x \in L_2$. Therefore,

  $L(G_u) \subseteq L_1 \cup L_2$

More Examples (Continued)

A grammar $G_c = (V_c, \Sigma, S_c, P_c)$ generating L_1L_2. Relabel variables if necessary so that $V_1 \cap V_2 = \varnothing$, and define

$V_c = V_1 \cup V_2 \cup \{S_c\}$
$P_c = P_1 \cup P_2 \cup \{S_c \to S_1S_2\}$

If $x \in L_1L_2$, then $x = x_1x_2$, where $x_i \in L_i$ for each i. It is then possible to derive x in G_c as follows:

$S_c \Rightarrow S_1S_2 \Rightarrow^* x_1S_2 \Rightarrow^* x_1x_2 = x$

Conversely, if x can be derived from S_c, then since the first step in the derivation must be $S_c \Rightarrow S_1S_2$, x must be derivable from S_1S_2. Therefore, $x = x_1x_2$, where for each i, x_i can be derived from S_i in G_i. Since $V_1 \cap V_2 = \varnothing$, being derivable from S_i in G_c means being derivable from S_i in G_i, and so $x \in L_1L_2$.

A grammar $G^* = (V, \Sigma, S, P)$ generating L_1^*. Let $V = V_1 \cup \{S\}$ where $S \notin V_1$. The language L_1^* contains strings of the form $x = x_1x_2\ldots x_k$, where each $x_i \in L_1$. Since each x_i can be derived from S_1, to derive x from S it is enough to be able to derive a string of k S_1's. This can be accomplished by defining P as follows:

$P = P_1 \cup \{S \to S_1S \mid \Lambda\}$

The proof that $L_1^* \subseteq L(G^*)$ is straightforward. If $x \in L(G^*)$, then either $x = \Lambda$ or x can be derived from some string of the form S_1^k in G^*. In the second case, the only productions in G^* beginning with S_1 are those in G_1, and therefore

$x \in L(G_1)^k \subseteq L(G_1)^*$

- **Corollary 6.1.** Every regular language is a CFL.

  Proof: According to Definition 3.1, regular languages over Σ are the languages obtained from $\varnothing$, $\{\Lambda\}$, and $\{a\}$ ($a \in \Sigma$) by using the operations of union, concatenation, and Kleene *. Each of the primitive languages $\varnothing$, $\{\Lambda\}$, and $\{a\}$ is clearly a context-free language. The corollary therefore follows from Theorem 6.1, using the principle of structural induction.

More Examples (Continued)

- **Example 6.9:** A CFG Equivalent to a Regular Expression

  Let L be the language corresponding to the regular expression

  (011 + 1)*(01)*

  Productions that generate the language {011, 1}:

  $A \rightarrow 011 \mid 1$

  Productions that generate the language {011, 1}* (using B as the start symbol):

  $B \rightarrow AB \mid \Lambda$
  $A \rightarrow 011 \mid 1$

  Productions that generate the language {01}* (using C as the start symbol):

  $C \rightarrow DC \mid \Lambda$
  $D \rightarrow 01$

  The final set of productions:

  $S \rightarrow BC$
  $B \rightarrow AB \mid \Lambda$
  $A \rightarrow 011 \mid 1$
  $C \rightarrow DC \mid \Lambda$
  $D \rightarrow 01$

- Starting with any regular expression, an equivalent CFG can be obtained using the techniques illustrated in this example.

- The next section shows that any regular language L can also be described by a CFG whose productions all have a very simple form, and that such a CFG can be obtained easily from an FA accepting L.

More Examples (Continued)

- **Example 6.10:** A CFG for $\{x \mid n_0(x) \neq n_1(x)\}$

  Consider the language

  $L = \{x \in \{0, 1\}^* \mid n_0(x) \neq n_1(x)\}$

  There is no general technique for finding a grammar generating the complement of a given CFL—which may in some cases not be a context-free language at all. However, L can be expressed as the union of the following languages:

  $L_0 = \{x \in \{0, 1\}^* \mid n_0(x) > n_1(x)\}$
  $L_1 = \{x \in \{0, 1\}^* \mid n_1(x) > n_0(x)\}$

  If a CFG G_0 generating the language L_0 can be found, then it should also be possible to find a CFG G_1 generating the language L_1.

  Clearly $0 \in L_0$, and for any $x \in L_0$, both $x0$ and $0x$ are in L_0. This suggests the productions

  $S \rightarrow 0 \mid S0 \mid 0S$

  Adding a 1 to an element of L_0 will not always produce an element of L_0; however, concatenating two strings in L_0 produces a string with at least two more 0's than 1's, and then adding a single 1 will still yield an element of L_0. The 1 could be added at the left, at the right, or between the two strings:

  $S \rightarrow 1SS \mid SS1 \mid S1S$

  It is not hard to see that any string derived by using the productions

  $S \rightarrow 0 \mid S0 \mid 0S \mid 1SS \mid SS1 \mid S1S$

  is an element of L_0.

  In the converse direction it is possible to do a little better: if G_0 is the grammar with productions

  $S \rightarrow 0 \mid 0S \mid 1SS \mid SS1 \mid S1S$

  every string in L_0 can be derived in G_0.

More Examples (Continued)

The proof is by induction on the length of the string. As in previous examples, let $d(x) = n_0(x) - n_1(x)$. The basis step, for a string in L_0 of length 1, is straightforward. Suppose that $k \geq 1$ and that any x for which $|x| \leq k$ and $d(x) > 0$ can be derived in G_0; and consider a string x for which $|x| = k + 1$ and $d(x) > 0$.

Consider the case in which $x = 0y0$ for some string y. (The other cases—x starts with 1 and x ends with 1—are left as an exercise.) If x contains only 0's, it can be derived from S using the productions $S \rightarrow 0 \mid 0S$; assume, therefore, that x contains at least one 1. The goal is to show that x has the form

$x = w1z$ for some w and z with $d(w) > 0$ and $d(z) > 0$

Then, by the induction hypothesis, both w and z can be derived from S, so that x can be derived by starting with the production

$S \rightarrow S1S$

Suppose that x contains n 1's, where $n \geq 1$. For each i with $1 \leq i \leq n$, let w_i be the prefix of x up to but not including the ith 1, and z_i the suffix of x that follows this 1. In other words, for each i, $x = w_i1z_i$ where the 1 is the ith 1 in x.

If $d(w_n) > 0$, then let $w = w_n$ and $z = z_n$. The string z_n is 0^j for some $j > 0$ because x ends with 0.

Otherwise, $d(w_n) \leq 0$. In this case select the *first* i with $d(w_i) \leq 0$, say $i = m$. Since x begins with 0, $d(w_1)$ must be > 0, which implies that $m \geq 2$. Thus, $d(w_{m-1}) > 0$ and $d(w_m) \leq 0$. Because w_m has only one more 1 than w_{m-1}, $d(w_{m-1})$ can be no more than 1. Therefore, $d(w_{m-1}) = 1$.

Since $x = w_{m-1}1z_{m-1}$, and $d(x) > 0$, it follows that $d(z_{m-1}) > 0$. This means that the desired result can be obtained by letting $w = w_{m-1}$ and $z = z_{m-1}$.

The context-free grammar G generating L is:

$S \rightarrow A \mid B$
$A \rightarrow 0 \mid 0A \mid 1AA \mid AA1 \mid A1A$
$B \rightarrow 1 \mid 1B \mid 0BB \mid BB0 \mid B0B$

More Examples (Continued)

- **Example 6.11:** Another Application of Theorem 6.1

  Let $L = \{0^i 1^j 0^k \mid j > i + k\}$. Observe that

  $0^i 1^{i+k} 0^k = 0^i 1^i 1^k 0^k$

  The only difference between this and a string x in L is that x has at least one extra 1 in the middle:

  $x = 0^i 1^i 1^m 1^k 0^k$ (for some $m > 0$)

  Therefore, it is possible to write $L = L_1 L_2 L_3$, where

  $L_1 = \{0^i 1^i \mid i \geq 0\}$
  $L_2 = \{1^m \mid m > 0\}$
  $L_3 = \{1^k 0^k \mid k \geq 0\}$

  L_1 is essentially the language in Example 6.2, L_3 is the same with the symbols 0 and 1 reversed, and L_2 can be generated by the productions

  $B \to 1B \mid 1$

  The final CFG $G = (V, \Sigma, S, P)$ incorporating these pieces is shown below.

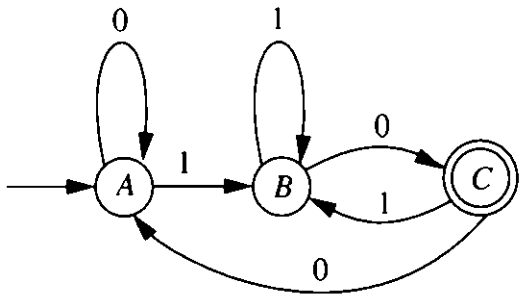
  $V = \{S, A, B, C\}$ $\quad \Sigma = \{0, 1\}$
  $P = \{S \to ABC$
  $\quad A \to 0A1 \mid \Lambda$
  $\quad B \to 1B \mid 1$
  $\quad C \to 1C0 \mid \Lambda\}$

  A derivation of $01^4 0^2 = (01)(1)(1^2 0^2)$:

  $S \Rightarrow ABC \Rightarrow 0A1BC \Rightarrow 0\Lambda 1BC \Rightarrow 011C$
  $\Rightarrow 0111C0 \Rightarrow 01111C00 \Rightarrow 01111\Lambda 00 = 0111100$

6.3 Regular Grammars

- The proof of Theorem 6.1 provides an algorithm for constructing a CFG corresponding to a given regular expression.
- A CFG for a regular language L can also be constructed from an FA accepting L. The productions have a very simple form, closely related to the moves of the FA.
- Consider the following FA, which accepts the language $L = \{0, 1\}^*\{10\}$:



A trace of the FA's actions for the string 110001010:

Substring processed so far	***State***
Λ	A
1	B
11	B
110	C
1100	A
11000	A
110001	B
1100010	C
11000101	B
110001010	C

If the lines of this table are listed consecutively, separated by $\Rightarrow$, the result looks like a derivation:

$$A \Rightarrow 1B \Rightarrow 11B \Rightarrow 110C \Rightarrow 1100A \Rightarrow 11000A \Rightarrow 110001B$$
$$\Rightarrow 1100010C \Rightarrow 11000101B \Rightarrow 110001010C$$

Regular Grammars (Continued)

The grammar can be obtained by specifying the variables to be the states of the FA and starting with the productions

$A \to 1B$
$B \to 1B$
$B \to 0C$
$C \to 0A$
$A \to 0A$
$C \to 1B$

These include every production of the form

$P \to aQ$

where

$P \overset{a}{\to} Q$

is a transition in the FA. The start symbol is A, the initial state of the FA. Adding the production $B \to 0$ allows C to be removed from the last string:

$11000101B \Rightarrow 110001010$

This production has the form

$P \to a$

where

$P \overset{a}{\to} F$

is a transition from P to an accepting state F.

- This construction works for any FA, except that the resulting grammar has no Λ-productions and therefore cannot generate Λ.

Regular Grammars (Continued)

- **Definition 6.3:** A grammar $G = (V, \Sigma, S, P)$ is *regular* if every production takes one of the two forms

  $B \rightarrow aC$
  $B \rightarrow a$

  for variables B and C and terminals a.

- **Theorem 6.2.** For any language $L \subseteq \Sigma^*$, L is regular if and only if there is a regular grammar G so that $L(G) = L - \{\Lambda\}$.

  Proof: First, suppose that L is regular and let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA accepting L. Define the grammar $G = (V, \Sigma, S, P)$ by letting $V = Q$, $S = q_0$, and

  $P = \{B \rightarrow aC \mid \delta(B, a) = C\} \cup \{B \rightarrow a \mid \delta(B, a) = F$ for some $F \in A)$

  Suppose that $x = a_1a_2\ldots a_n$ is accepted by M, and $n \geq 1$. Then there is a sequence of transitions

  $$\rightarrow q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \ldots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q_n$$

  where $q_n \in A$. By definition of G, we have the corresponding derivation

  $$S = q_0 \Rightarrow a_1q_1 \Rightarrow a_1a_2q_2 \Rightarrow \ldots \Rightarrow a_1a_2\ldots a_{n-1}q_{n-1} \Rightarrow a_1a_2\ldots a_{n-1}a_n$$

  Similarly, if x is generated by G, it is clear that x is accepted by M.

  Conversely, suppose $G = (V, \Sigma, S, P)$ is a regular grammar generating L. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA in which Q is the set $V \cup \{f\}$, q_0 is the start symbol S, and A is the set $\{f\}$. For any $q \in V$ and $a \in \Sigma$,

  $$\delta(q, a) = \begin{cases} \{p \mid \text{the production } q \rightarrow ap \text{ is in } P\} & \text{if } q \rightarrow a \text{ is not in } P \\ \{p \mid q \rightarrow ap \text{ is in } P\} \cup \{f\} & \text{if } q \rightarrow a \text{ is in } P \end{cases}$$

  There are no transitions out of f.

Regular Grammars (Continued)

If $x = a_1a_2\ldots a_{n-1}a_n \in L(G)$, then there is a derivation of the form

$S \Rightarrow a_1q_1 \Rightarrow a_1a_2q_2 \Rightarrow \ldots \Rightarrow a_1a_2\ldots a_{n-1}q_{n-1} \Rightarrow a_1a_2\ldots a_{n-1}a_n$

and according to the definition of M, there is a corresponding sequence of transitions

$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} f$

which implies that x is accepted by M.

On the other hand, if $x = a_1a_2\ldots a_n$ is accepted by M, then $|x| \geq 1$ because f is the only accepting state of M. The transitions causing x to be accepted look like

$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} f$

These transitions correspond to a derivation of x in the grammar, and it follows that $x \in L(G)$.

- Sometimes the term *regular* is applied to grammars that do not restrict the form of the productions so severely.

- It can be shown that a language is regular if and only if it can be generated, except possibly for Λ, by a grammar in which all productions have the form

  $B \to xC$
  $B \to x$

  where B and C are variables and x is a nonnull string of terminals. Grammars of this type are also called *right linear.*

6.4 Derivation Trees and Ambiguity

- A natural way of exhibiting the structure of a derivation is to draw a *derivation tree*, or *parse tree*.

- At the root of the tree is the variable with which the derivation begins.

- Interior nodes correspond to variables that appear in the derivation. The children of the node corresponding to A represent the symbols in a string α for which the production $A \to \alpha$ is used in the derivation.

- In the case of a production $A \to \Lambda$, the node labeled A has the single child Λ.

- Concatenating the symbols at the leaf nodes of a derivation tree will show the string that was derived. (Nodes that correspond to Λ can be ignored, because Λ disappears when concatenated with other symbols.)

- Derivation trees can also be constructed in which the root is a variable other than the start symbol of the grammar, and the string being derived still contains some variables as well as terminal symbols.

Derivation Trees (Continued)

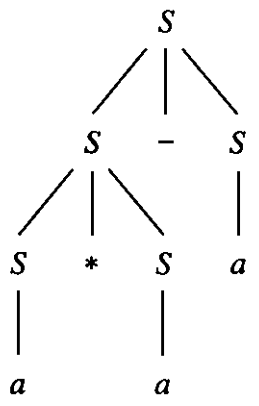
- Consider the CFG with the following productions:

  $S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid a$

  The derivation

  $S \Rightarrow S - S \Rightarrow S * S - S \Rightarrow a * S - S \Rightarrow a * a - S \Rightarrow a * a - a$

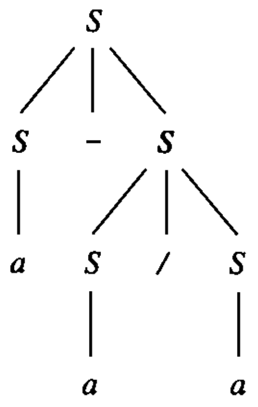
  has the following derivation tree:



  The derivation

  $S \Rightarrow S - S \Rightarrow S - S / S \Rightarrow a - S / S \Rightarrow a - a / S \Rightarrow a - a / a$

  has the following derivation tree:



  In general, any derivation of a string in a CFL has exactly one corresponding derivation tree.

Derivation Trees (Continued)

- There is a problem with $S \to SS \mid a$, because the derivation $S \Rightarrow SS \Rightarrow SSS \Rightarrow^* aaa$ corresponds to two different derivation trees, depending on which S is replaced in the second step.

- Therefore, specifying a derivation means giving not only the sequence of strings but also the position in each string at which the next substitution occurs.

- Derivation trees allow us to ignore the order in which productions are used. The derivations

  $S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + a$

  and

  $S \Rightarrow S + S \Rightarrow S + a \Rightarrow a + a$

  are identical except for the order in which productions are applied. Since both derivations correspond to the same derivation tree, they are essentially the same.

- It is easier to compare derivations if they are normalized, by requiring that each follow the same rule as to which variable to replace first when there is a choice.

Leftmost Derivations

- A derivation is *leftmost* if the variable used in each step is always the leftmost variable of the ones in the current string.

- There is a one-to-one correspondence between leftmost derivations and derivation trees.

- Clearly every leftmost derivation corresponds to a single derivation tree.

- On the other hand, the derivation trees corresponding to two different leftmost derivations are different. Consider the first step at which the derivations differ. Suppose that this step is

  $xA\beta \Rightarrow x\alpha_1\beta$

  in one derivation and

  $xA\beta \Rightarrow x\alpha_2\beta$

  in the other. Here x is a string of terminals, A is a variable, and $\alpha_1 \neq \alpha_2$. The two derivation trees must both have a node labeled A, and the portions of the two trees to the left of this node must be identical. These two nodes have different children, however, and the trees cannot be the same.

- The same reasoning applies to rightmost derivations.

Ambiguity

- **Definition 6.4:** A context-free grammar G is *ambiguous* if there is at least one string in $L(G)$ having two or more distinct derivation trees (or, equivalently, two or more distinct leftmost derivations).

- **Example 6.12:** Ambiguity in the CFG in Example 6.4

  The algebraic-expression CFG in Example 6.4 has the productions

  $S \to S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid a$

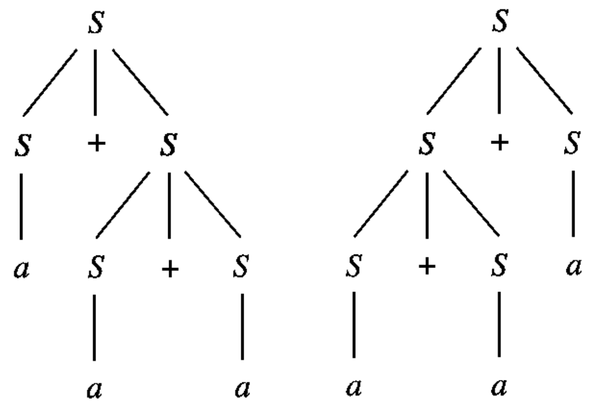
  This grammar is ambiguous, which can be demonstrated using only the productions $S \to S + S$ and $S \to a$. The string $a + a + a$ has leftmost derivations

  $S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + S + S \Rightarrow a + a + S \Rightarrow a + a + a$

  and

  $S \Rightarrow S + S \Rightarrow S + S + S \Rightarrow a + S + S \Rightarrow a + a + S \Rightarrow a + a + a$

  These derivations lead to the following derivation trees:



  The expression is interpreted as $a + (a + a)$ in one case, and $(a + a) + a$ in the other.

- Every CFG containing a production of the form $A \to A\alpha A$ is ambiguous. However, there are more subtle ways in which ambiguity occurs. Characterizing the ambiguous context-free grammars in any nontrivial way turns out to be difficult or impossible.

The “Dangling Else” Problem

- **Example 6.13:** The “Dangling Else”

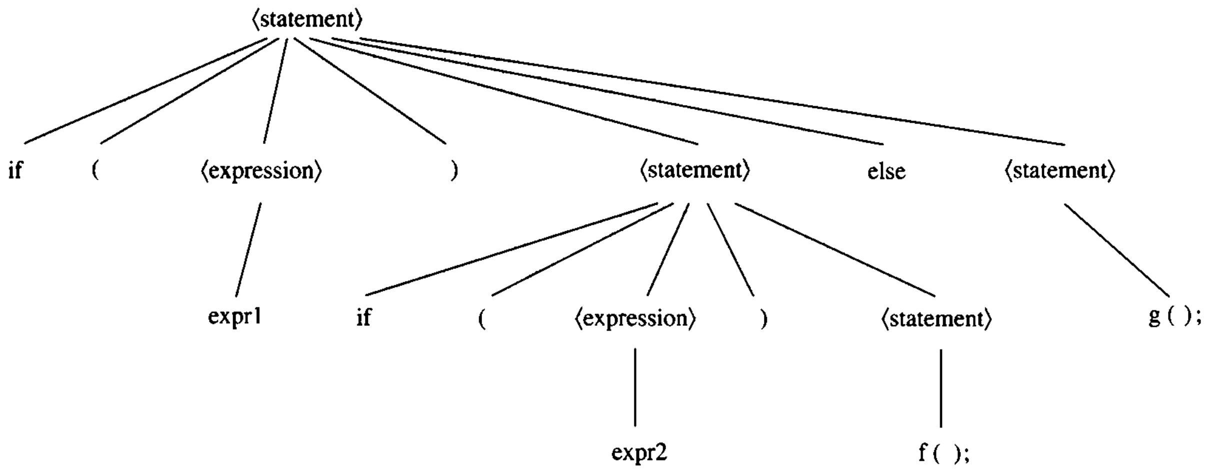
  Consider the productions

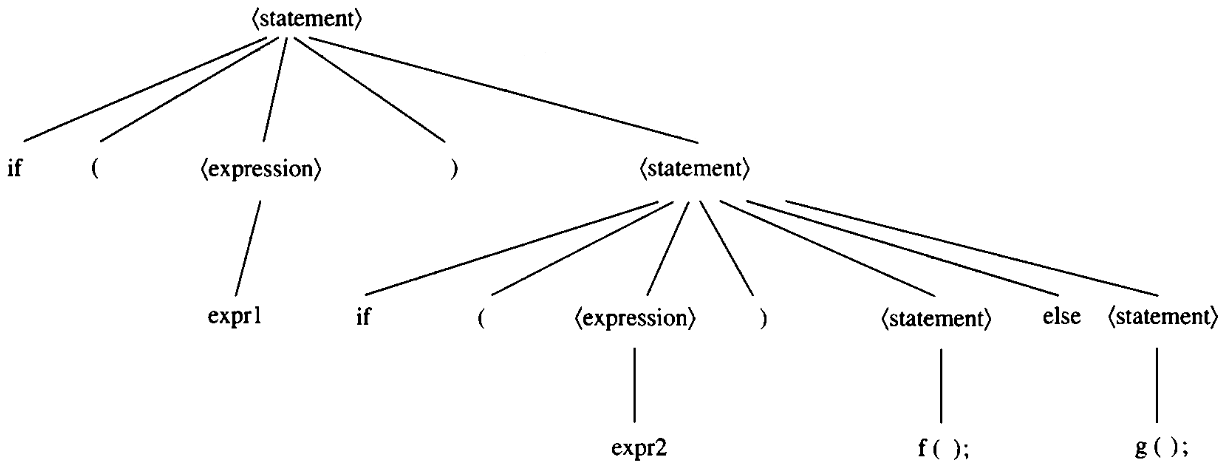
  <statement> → if (<expression>) <statement> |
  if (<expression>) <statement> else <statement> |
  <otherstatement>

  describing the *if* statement in the C language. The statement

```
if (expr1) if (expr2) f(); else g();
```

  can be derived in two ways, as shown by the following derivation trees:





The “Dangling Else” Problem (Continued)

A C compiler should interpret the statement the second way. The programmer can also use braces to remove the ambiguity in the statement. Writing

```
if (expr1) {if (expr2) f();} else g();
```

forces the first interpretation, whereas

```
if (expr1) {if (expr2) f(); else g();}
```

forces the second.

The following grammar rules generate the same strings as the original rules and can be shown to be unambiguous:

```
<statement> → <st1> | <st2>
      <st1> → if (<expression>) <st1> else <st1> | <otherstatement>
      <st2> → if (<expression>) <statement> |
              if (<expression>) <st1> else <st2>
```

<st1> represents a statement in which every *if* is matched by a corresponding *else*, while any statement derived from <st2> contains at least one unmatched *if*. The only variable appearing before *else* in these formulas is <st1>; since the *else* cannot match any of the *if*’s in the statement derived from <st1>, it must match the *if* that appeared in the formula with the *else*.

Languages such as Modula-2 avoid the dangling else problem by requiring an explicit END symbol at the end of each *if* statement:

```
<statement> → IF <expression> THEN <statementsequence> END |
              IF <expression) THEN <statementsequence>
              ELSE <statementsequence> END |
              <otherstatement>
```

6.5 An Unambiguous CFG for Algebraic Expressions

- Some context-free languages are inherently ambiguous, in the sense that they can be produced only by ambiguous grammars.

- Ambiguity is normally a property of a grammar, however. If a CFG is ambiguous, it is often possible to find an equivalent unambiguous CFG.

- Consider the problem of removing ambiguity from the algebraic-expression grammar of Example 6.4. For simplicity, productions involving operators other than + and * will be removed:

  $S \rightarrow S + S \mid S * S \mid (S) \mid a$

- To remove ambiguity from the grammar, it will be rewritten to incorporate the usual rules for operator precedence and associativity.

- The rule $S \rightarrow S + S$ will need to be replaced by either $S \rightarrow S + T$ or $S \rightarrow T + S$, where T stands for *term*, an expression that cannot itself be expressed as a sum. $S \rightarrow S + T$ is the right choice, since it reflects the fact that + is left associative.

- An expression can also consist of a single term, so the production $S \rightarrow T$ will also be needed:

  $S \rightarrow S + T \mid T$

- A term can be a product of *factors*, leading to the following productions:

  $T \rightarrow T * F \mid F$

- A parenthesized expression cannot be expressed directly as either a sum or a product, and it therefore seems most appropriate to consider it a factor, leading to the following productions:

  $F \rightarrow (S) \mid a$

An Unambiguous CFG for Algebraic Expressions (Continued)

- **Theorem 6.3.** Let G be the context-free grammar with productions

  $S \to S + S \mid S * S \mid (S) \mid a$

  and let $G1$ be the context-free grammar with productions

  $S1 \to S1 + T \mid T$
  $T \to T * F \mid F$
  $F \to (S1) \mid a$

  Then $L(G) = L(G1)$.

  Proof: **First, to show $L(G1) \subseteq L(G)$.** The proof is by induction on the length of a string in $L(G1)$. The basis step is to show that $a \in L(G)$, and this is clear.

  In the induction step, assume that $k \geq 1$ and that every y in $L(G1)$ satisfying $|y| \leq k$ is in $L(G)$. The goal is to show that if $x \in L(G1)$ and $|x| = k + 1$, then $x \in L(G)$. Since $x \neq a$, any derivation of x in $G1$ must begin in one of three ways:

  $S1 \Rightarrow S1 + T$
  $S1 \Rightarrow T \Rightarrow T * F$
  $S1 \Rightarrow T \Rightarrow F \Rightarrow (S1)$

  Consider the first case (the other two are similar). If x has a derivation beginning $S1 \Rightarrow S1 + T$, then $x = y + z$, where $S1 \Rightarrow_{G1}^* y$ and $T \Rightarrow_{G1}^* z$. Since $S1 \Rightarrow_{G1}^* T$, it follows that $S1 \Rightarrow_{G1}^* z$.

  Since $|y|$ and $|z|$ must be $\leq k$, the induction hypothesis implies that y and z are both in $L(G)$. Since G contains the production $S \to S + S$, the string $y + z$ is derivable from S in G, and therefore $x \in L(G)$.

  Second, to show $L(G) \subseteq L(G1)$. Again, the proof is by induction on $|x|$. The basis step is straightforward. Assume that $k \geq 1$ and that for every $y \in L(G)$ with $|y| \leq k$, $y \in L(G1)$; the goal is to show that if $x \in L(G)$ and $|x| = k + 1$, then $x \in L(G1)$.

  Case 1: x has a derivation in G beginning $S \Rightarrow (S)$. In this case $x = (y)$, for some y in $L(G)$, and it follows from the inductive hypothesis that $y \in L(G1)$. Therefore, x can be derived in $G1$ by starting the derivation $S1 \Rightarrow T \Rightarrow F \Rightarrow (S1)$ and then deriving y from $S1$.

An Unambiguous CFG for Algebraic Expressions (Continued)

Case 2: x has a derivation in G that begins $S \Rightarrow S + S$. Then the induction hypothesis says that $x = y + z$, where y and z are both in $L(G1)$. Let

$$x = x_1 + x_2 + \dots + x_n$$

where each $x_i \in L(G1)$ and n is as large as possible. It must be the case that $n \geq 2$. Because of the way n is defined, none of the x_i's can have a derivation in $G1$ that begins $S1 \Rightarrow S1 + T$, therefore, every x_i can be derived from T in $G1$. Let

$$y = x_1 + x_2 + \dots + x_{n-1} \qquad z = x_n$$

Then y can be derived from $S1$, since $S1 \Rightarrow_{G1}{}^* T + T + \dots + T$ ($n - 1$ terms), and z can be derived from T. It follows that $x \in L(G1)$, since a derivation of x can start with the production $S1 \rightarrow S1 + T$.

Case 3: Every derivation of x in G begins $S \Rightarrow S * S$. Then for some y and z in $L(G)$, $x = y * z$. This time let

$$x = x_1 * x_2 * \dots * x_n$$

where each $x_i \in L(G)$ and n is as large as possible. Then by the inductive hypothesis, each $x_i \in L(G1)$. No x_i can have a derivation in $G1$ that begins $S1 \Rightarrow T \Rightarrow T * F$. If this were true, x_i would be of the form $y_i * z_i$ for some y_i , $z_i \in L(G1)$; since $L(G1) \subseteq L(G)$, this would contradict the maximal property of n.

Suppose that some x_i had a derivation in $G1$ beginning $S1 \Rightarrow S1 + T$. Then $x_i = y_i + z_i$ for some y_i , $z_i \in L(G1) \subseteq L(G)$. In this case,

$$x = x_1 * x_2 * \dots * x_{i-1} * y_i + z_i * x_{i+1} * \dots * x_n$$

This is impossible too. If u and v are the substrings before and after the +, respectively, then $u, v \in L(G)$, and therefore $x = u + v$. This means that x could be derived in G using a derivation that begins $S \Rightarrow S + S$, which was assumed not to be the case.

The conclusion is that each x_i is derivable from F in $G1$. Now, let

$$y = x_1 * x_2 * \dots * x_{n-1} \qquad z = x_n$$

The string y is derivable from T in $G1$, since $F * F * \dots * F$ ($n - 1$ factors) is. Therefore, x can be derived from $S1$ in $G1$ by starting the derivation $S1 \Rightarrow T \Rightarrow T * F$, and so $x \in L(G1)$.

An Unambiguous CFG for Algebraic Expressions (Continued)

- In order to show that $G1$ is unambiguous, it will be helpful to concentrate on the parentheses in a string, temporarily ignoring the other terminal symbols.

- **Definition 6.5:** A string of left and right parentheses is *balanced* if it has equal numbers of left and right parentheses, and no prefix has more right than left.

  The *mate* of a left parenthesis in a balanced string is the first right parenthesis following it for which the string containing those two and everything in between is balanced.

  If x is a string containing parentheses and other symbols, and the parentheses within x form a balanced string, a symbol σ in x is *within parentheses* if σ appears between some left parenthesis and its mate.

- Note that the string of parentheses in any string obtained from $S1$ in the grammar $G1$ is balanced. Also, the parentheses between and including the pair produced by a single application of the production $F \rightarrow (S1)$ form a balanced string.

- Now suppose that $x \in L(G1)$, and $(_0$ is any left parenthesis in x. In any leftmost derivation of x, it is easy to see that the right parenthesis produced at the same time as $(_0$ is the mate of $(_0$.

- The conclusion is that when a symbol is *within parentheses,* these must be the two parentheses produced by the production $F \rightarrow (S1)$.

An Unambiguous CFG for Algebraic Expressions (Continued)

- **Theorem 6.4.** The context-free grammar $G1$ with productions

  $S1 \rightarrow S1 + T \mid T$
  $T \rightarrow T * F \mid F$
  $F \rightarrow (S1) \mid a$

  is unambiguous.

  Proof: The goal is to show that every string x in $L(G1)$ has only one leftmost derivation from $S1$. The proof will be by mathematical induction on $|x|$. It will actually be easier to prove a stronger statement: For any x derivable from one of the variables $S1$, T, or F, x has only one leftmost derivation from that variable.

  For the basis step, we observe that a can be derived from any of the three variables, and that in each case there is only one derivation.

  In the induction step, assume that $k \geq 1$ and that for every y derivable from $S1$, T, or F for which $|y| = k$, y has only one leftmost derivation from that variable. The goal is to show the same result for a string x with $|x| = k + 1$.

  Case 1: x contains at least one + not within parentheses. Since the only +'s in strings derivable from T or F are within parentheses, x can be derived only from $S1$, and any derivation of x must begin $S1 \Rightarrow S1 + T$, where this + is the last + in x that is not within parentheses.

  Therefore, any leftmost derivation of x from $S1$ has the form

  $S1 \Rightarrow S1 + T \Rightarrow^* y + T \Rightarrow^* y + z$

  where the last two steps represent leftmost derivations of y from $S1$ and z from T, respectively, and the + is still the last one not within parentheses

  The induction hypothesis says that y has only one leftmost derivation from $S1$ and z has only one from T. Therefore, x has only one leftmost derivation from $S1$.

An Unambiguous CFG for Algebraic Expressions (Continued)

*Case 2: x contains no + outside parentheses but at least one * outside parentheses.* This time x can be derived only from $S1$ or T; any derivation from $S1$ must begin $S1 \Rightarrow T \Rightarrow T * F$; and any derivation from T must begin $T \Rightarrow T * F$. In either case, the * must be the last one in x that is not within parentheses.

As in the first case, the subsequent steps of any leftmost derivation must be

$T * F \Rightarrow^* y * F \Rightarrow^* y + z$

consisting first of a leftmost derivation of y from T and then of a leftmost derivation of z from F.

The induction hypothesis says that there is only one possible way for these derivations to proceed, and so there is only one leftmost derivation of x from $S1$ or T.

*Case 3: x contains no +'s or *'s outside parentheses.* The only derivation of x from $S1$ begins $S1 \Rightarrow T \Rightarrow F \Rightarrow (S1)$, and the only derivation from T or F begins the same way with the first one or two steps omitted. Therefore, $x = (y)$, where $S1 \Rightarrow^* y$.

By the induction hypothesis, y has only one leftmost derivation from $S1$, and it follows that x has only one from each of the three variables.

6.6 Simplified Forms and Normal Forms

- It is often useful to improve a grammar by eliminating “Λ-productions,” which have the form $A \to \Lambda$, and “unit productions,” in which one variable is simply replaced by another.

- In some situations, it can also be important to standardize productions so that they have a certain “normal form.”

- To illustrate how these improvements might be useful, suppose that a grammar contains no Λ-productions or unit productions, and consider a derivation containing the step

  $\alpha \Rightarrow \beta$

  If there are no Λ-productions, then β must be at least as long as α; if there are no unit productions, α and β can be of equal length only if this step consists of replacing a variable by a single terminal.

- If l and t represent the length of the current string and the number of terminals in the current string, respectively, then $l + t$ must increase at each step of the derivation. Since the value of $l + t$ is 1 for the string S and $2k$ for a string x of length k in the language, a derivation of x can have no more than $2k - 1$ steps.

- This observation provides an (inefficient) algorithm for determining whether a given string x is generated by the grammar: If $|x| = k$, try all possible sequences of $2k - 1$ productions, and see if any of them produces x.

- *Note:* It is not possible to eliminate Λ-productions from a grammar if the grammar generates Λ itself. However, it will be possible to show that for any context-free language L, $L - \{\Lambda\}$ can be generated by a CFG with no Λ-productions.

Eliminating Λ-Productions

- **Example 6.14:** Eliminating Λ-productions from a CFG

  Let G be the context-free grammar with productions

  $S \to ABCBCDA$
  $A \to CD$
  $B \to Cb$
  $C \to a \mid \Lambda$
  $D \to bD \mid \Lambda$

  Consider the production $S \to ABCBCDA$, which we write temporarily as

  $S \to ABC_1BC_2DA$

  The variables C_1, C_2, and D all begin Λ-productions, and each can also be used to derive a nonnull string. In a derivation none, any, or all of these three can be replaced by Λ.

  Without Λ-productions, it will be necessary to add productions of the form $S \to \alpha$, where α is a string obtained from $ABCBCDA$ by deleting some or all of $\{C_1, C_2, D\}$. In other words, the productions

  $$\begin{aligned} S \to\ & ABBC_2DA \mid ABC_1BDA \mid ABC_1BC_2A \mid \\ & ABBDA \mid ABBC_2A \mid ABC_1BA \mid \\ & ABBA \end{aligned}$$

  will need to be added to the grammar.

  Although A does not begin a Λ-production, Λ can be derived from A (as can other nonnull strings). Therefore, it will be necessary to add versions of the the production $A \to CD$ in which C or D are left out (but not both).

  The fact that A derives Λ will affect the production for S. Adding subscripts to the occurrences of A gives the following result:

  $S \to A_1BC_1BC_2DA_2$

  It will be necessary to add productions in which the right side is obtained by leaving out some subset of $\{A_1, A_2, C_1, C_2, D\}$. There are 32 subsets, so 31 new productions will need to be added to the grammar.

Eliminating Λ-Productions (Continued)

In general, if $X \to \alpha$ is a production containing variables from which Λ can be derived, then it will be necessary to add all the productions $X \to \alpha'$, where α' is obtained from α by deleting some of these variables.

This procedure might produce new Λ-productions—if so, they are ignored—and it might produce productions of the form $X \to X$, which also contribute nothing to the grammar and can be omitted.

For this particular grammar, the final version will have 40 productions, including the 32 S-productions already mentioned and the ones that follow:

$A \to CD \mid C \mid D$
$B \to Cb \mid b$
$C \to a$
$D \to bD \mid b$

- **Definition 6.6:** A *nullable* variable in a CFG $G = (V, \Sigma, S, P)$ is defined as follows.

  1. Any variable A for which P contains the production $A \to \Lambda$ is nullable.
  2. If P contains the production $A \to B_1B_2\ldots B_n$ and $B_1, B_2, \ldots, B_n$ are nullable variables, then A is nullable.
  3. No other variables in V are nullable.

- **Algorithm FindNull (Finding the nullable variables in a CFG (V, Σ, S, P))**

```
$N_0 = \{A \in V \mid P$ contains the production $A \to \Lambda\}$;
$i = 0$;
do
    $i = i + 1$;
    $N_i = N_{i-1} \cup \{A \mid P$ contains $A \to \alpha$ for some $\alpha \in N_{i-1}^* \}$
while $N_i \neq N_{i-1}$;
$N_i$ is the set of nullable variables.
```

- When the algorithm is applied in Example 6.14, the set N_0 is $\{C, D\}$. The set N_1 also contains A, as a result of the production $A \to CD$.

Eliminating Λ-Productions (Continued)

- **Algorithm 6.1 (Finding an equivalent CFG with no Λ-productions)** Given a CFG $G = (V, \Sigma, S, P)$, construct a CFG $G1 = (V, \Sigma, S, P1)$ with no Λ-productions as follows.

  1. Initialize $P1$ to be P.
  2. Find all nullable variables in V, using Algorithm FindNull.
  3. For every production $A \rightarrow \alpha$ in P, add to $P1$ every production that can be obtained from this one by deleting from α one or more of the occurrences of nullable variables in α.
  4. Delete all Λ-productions from $P1$. Also delete any duplicates, as well as productions of the form $A \rightarrow A$.

- **Theorem 6.5.** Let $G = (V, \Sigma, S, P)$ be any context-free grammar, and let $G1$ be the grammar obtained from G by Algorithm 6.l. Then $G1$ has no Λ-productions, and $L(G1) = L(G) - \{\Lambda\}$.

  Proof: That $G1$ has no Λ-productions is obvious. We show a statement that is slightly stronger than that in the theorem. For any variable $A \in V$, and any non-null $x \in \Sigma^*$,

  $A \Rightarrow_G^* x$ if and only if $A \Rightarrow_{G1}^* x$

  The notation $A \Rightarrow_G^k x$ will mean that there is a k-step derivation of x from A in G.

  The first goal is to show that for any $n \geq 1$, if $A \Rightarrow_G^n x$, then $A \Rightarrow_{G1}^* x$. The proof is by mathematical induction on n.

  For the basis step, suppose $A \Rightarrow_G^1 x$. Then $A \rightarrow x$ is a production in P. Since $x \neq \Lambda$, this production is also in $P1$, and so $A \Rightarrow_{G1}^* x$.

  In the induction step, assume that $k \geq 1$ and that any string other than Λ derivable from A in k or fewer steps in G is derivable from A in $G1$. The goal is to show that if $x \neq \Lambda$ and $A \Rightarrow_G^{k+1} x$, then $A \Rightarrow_{G1}^* x$.

Eliminating Λ-Productions (Continued)

Suppose that the first step in a $(k + 1)$-step derivation of x in G is $A \to X_1X_2\ldots X_n$, where each X_i is either a variable or a terminal. Then $x = x_1x_2\ldots x_n$, where each x_i is either equal to X_i or derivable from X_i in k or fewer steps in G.

Any X_i for which the corresponding x_i is Λ is a nullable variable in G. If these X_i's are deleted from the string $X_1X_2\ldots X_n$, there are still some left, since $x \neq \Lambda$ and the resulting production is an element of $P1$. Furthermore, the induction hypothesis says that for each X_i remaining in the right side of this production, $X_i \Rightarrow_{G1}^* x_i$. Therefore, $A \Rightarrow_{G1}^* x$.

The goal is now to show the converse, that for any $n \geq 1$, if $A \Rightarrow_{G1}^n x$, then $A \Rightarrow_G^* x$; again the proof is by induction on n.

If $A \Rightarrow_{G1}^1 x$, then $A \to x$ is a production in $P1$. This means that $A \to \alpha$ is a production in P, where x is obtained from α by deleting zero or more nullable variables. It follows that $A \Rightarrow_G^* x$, because a derivation can start with the production $A \to \alpha$ and proceed by deriving Λ from each of the nullable variables that was deleted to obtain x.

Suppose that $k \geq 1$ and that any string other than Λ derivable from A in k or fewer steps in $G1$ is derivable from A in G. The goal is to show the same result for a string x for which $A \Rightarrow_{G1}^{k+1} x$.

Let the first step of a $(k + 1)$-step derivation of x in $G1$ be $A \to X_1X_2\ldots X_n$, where each X_i is either a variable or a terminal. Then $x = x_1x_2\ldots x_n$ where each x_i is either equal to X_i or derivable from X_i in k or fewer steps in $G1$.

By the induction hypothesis, $X_i \Rightarrow_G^* x_i$ for each i. By definition of $G1$, there is a production $A \to \alpha$ in P so that $X_1X_2\ldots X_n$ can be obtained from α by deleting certain nullable variables. Since $A \Rightarrow_G^* X_1X_2\ldots X_n$, it is possible to derive x from A in G by first deriving $X_1X_2\ldots X_n$ and then deriving each x_i from the corresponding X_i.

- If the context-free grammar G is unambiguous, then the grammar $G1$ produced by Algorithm 6.1 is also.

Eliminating Unit Productions

- Eliminating unit productions is similar to eliminating Λ-productions.

- To ensure that eliminating unit productions does not also eliminate strings in the language, it will be necessary to add the production $A \to \alpha$ whenever $B \to \alpha$ is a nonunit production and $A \Rightarrow^* B$.

- The algorithm for eliminating unit productions assumes that Algorithm 6.1 has already been used, so that the grammar has no Λ-productions. Thus, one variable can be derived from another only by a sequence of unit productions.

- For any variable A, the set of "A-derivable" variables (essentially, variables B other than A for which $A \Rightarrow^* B$) is defined as follows:

  1. If $A \to B$ is a production, B is A-derivable.
  2. If C is A-derivable, $C \to B$ is a production, and $B \neq A$, then B is A-derivable.
  3. No other variables are A-derivable.

  Note that a variable A is A-derivable only if $A \to A$ is actually a production.

- **Algorithm 6.2 (Finding an equivalent CFG with no unit productions)** Given a context-free grammar $G = (V, \Sigma, S, P)$ with no Λ-productions, construct a grammar $G1 = (V, \Sigma, S, P1)$ having no unit productions as follows.

  1. Initialize $P1$ to be P.
  2. For each $A \in V$, find the set of A-derivable variables.
  3. For every pair (A, B) such that B is A-derivable, and every nonunit production $B \to \alpha$, add the production $A \to \alpha$ to $P1$ if it is not already present in $P1$.
  4. Delete all unit productions from $P1$.

Eliminating Unit Productions (Continued)

- **Theorem 6.6.** Let G be any CFG without Λ-productions, and let $G1$ be the CFG obtained from G by Algorithm 6.2. Then $G1$ contains no unit productions, and $L(G1) = L(G)$.

- It can be shown that if G is unambiguous, then $G1$ is also.

- **Example 6.15:** Eliminating Unit Productions

  Let G be the algebraic-expression grammar with the following productions:

  $S \rightarrow S + T \mid T$
  $T \rightarrow T * F \mid F$
  $F \rightarrow (S) \mid a$

  The S-derivable variables are T and F, and F is T-derivable.

  In step 3 of Algorithm 6.2, the productions $S \rightarrow T * F \mid (S) \mid a$ and $T \rightarrow (S) \mid a$ are added to $P1$.

  The resulting grammar when unit productions are deleted:

  $S \rightarrow S + T \mid T * F \mid (S) \mid a$
  $T \rightarrow T * F \mid (S) \mid a$
  $F \rightarrow (S) \mid a$

Chomsky Normal Form

- In addition to eliminating specific types of productions, it may also be useful to impose restrictions upon the form of the remaining productions. Several “normal forms” have been introduced, including Chomsky normal form.

- **Definition 6.7:** A context-free grammar is in *Chomsky normal form* (CNF) if every production is of one of these two types:

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

  where A, B, and C are variables and a is a terminal symbol.

- Transforming a grammar $G = (V, \Sigma, S, P)$ into Chomsky normal form may be done in three steps.

- *Step 1:* Apply Algorithms 6.1 and 6.2 to obtain a CFG $G1 = (V, \Sigma, S, P1)$ having neither Λ-productions nor unit productions so that $L(G1) = L(G) - \{\Lambda\}$.

- *Step 2:* Obtain a grammar $G2 = (V2, \Sigma, S, P2)$, generating the same language as $G1$, so that every production in $P2$ is either of the form

  $A \rightarrow B_1B_2\ldots B_k$

  where $k \geq 2$ and each B_i is a variable in $V2$, or of the form

  $A \rightarrow a$

  for some $a \in \Sigma$.

- Every production in $P1$ that is not already of the form $A \rightarrow a$ looks like $A \rightarrow \alpha$ for some string α of length at least 2. For every terminal a appearing in such a string α, introduce a new variable X_a and a new production $X_a \rightarrow a$, and replace a by X_a in all the productions where it appears (except those of the form $A \rightarrow a$).

- For example, the productions $A \rightarrow aAb$ and $B \rightarrow ab$ would be replaced by $A \rightarrow X_aAX_b$ and $B \rightarrow X_aX_b$, and the productions $X_a \rightarrow a$ and $X_b \rightarrow b$ would be added.

Chomsky Normal Form (Continued)

- *Step 3:* The right side of every production in $G2$ is either a single terminal or a string of two or more variables. The last step is to replace each production having more than two variables on the right by an equivalent set of productions, each one having exactly two variables on the right.

- For example, the production

  $A \rightarrow BCDBCE$

  would be replaced by

  $A \rightarrow BY_1$
  $Y_1 \rightarrow CY_2$
  $Y_2 \rightarrow DY_3$
  $Y_3 \rightarrow BY_4$
  $Y_4 \rightarrow CE$

  The new variables Y_1, Y_2, Y_3, Y_4 are specific to this production and would be used nowhere else.

- **Theorem 6.7.** For any context-free grammar $G = (V, \Sigma, S, P)$, there is a CFG $G' = (V', \Sigma, S, P')$ in Chomsky normal form so that $L(G') = L(G) - \{\Lambda\}$.

Chomsky Normal Form (Continued)

- **Example 6.16:** Converting a CFG to Chomsky Normal Form

  Let G be the grammar with productions

  $S \rightarrow AACD$
  $A \rightarrow aAb \mid \Lambda$
  $C \rightarrow aC \mid a$
  $D \rightarrow aDa \mid bDb \mid \Lambda$

  G can be converted to CNF in four steps.

  1. Eliminating Λ-productions. The nullable variables are A and D, and Algorithm 6.1 produces the grammar with productions

  $S \rightarrow AACD \mid ACD \mid AAC \mid CD \mid AC \mid C$
  $A \rightarrow aAb \mid ab$
  $C \rightarrow aC \mid a$
  $D \rightarrow aDa \mid bDb \mid aa \mid bb$

  2. Eliminating unit productions. Here it is necessary to add the productions

  $S \rightarrow aC \mid a$

  and delete $S \rightarrow C$.

  3. Restricting the right sides of productions to single terminals or strings of two or more variables. This step yields the productions

  $S \rightarrow AACD \mid ACD \mid AAC \mid CD \mid AC \mid X_aC \mid a$
  $A \rightarrow X_aAX_b \mid X_aX_b$
  $C \rightarrow X_aC \mid a$
  $D \rightarrow X_aDX_a \mid X_bDX_b \mid X_aX_a \mid X_bX_b$
  $X_a \rightarrow a$
  $X_b \rightarrow b$

Chomsky Normal Form (Continued)

4. The final step to CNF. There are six productions whose right sides are too long. Applying the algorithm produces the grammar with productions

$S \to AT_1 \quad T_1 \to AT_2 \quad T_2 \to CD$
$S \to AU_1 \quad U_1 \to CD$
$S \to AV_1 \quad V_1 \to AC$
$S \to CD \mid AC \mid X_aC \mid a$
$A \to X_aW_1 \quad W_1 \to AX_b$
$A \to X_aX_b$
$C \to X_aC \mid a$
$D \to X_aY_1 \quad Y_1 \to DX_a$
$D \to X_bZ_1 \quad Z_1 \to DX_b$
$D \to X_aX_b \mid X_bX_b$
$X_a \to a \quad X_b \to b$