Chapter 4

Context-Free Languages

Using Grammar Rules to Define a Language

- Regular languages and FAs are too simple for many purposes (Example: when we need to count something such as equal numbers of *a* and *b.*)
- We will use grammars, i.e., the set of rules for generating phrases and sentences.
  - Using *context-free grammars* allows us to describe more difficult languages
  - Much high-level programming language syntax can be expressed with context-free grammars
  - Context-free grammars with a very simple form provide another way to describe the regular languages
- Grammars can be ambiguous

For example, a string could be parsed using more than one chain of rules which can lead to different interpretations

Let's begin with an example …

- Consider the language $AnBn = \{a^n b^n \mid n \geq 0\}$, defined using the recursive definition:
  - $\Lambda \in AnBn$
  - For every $S \in AnBn$, $aSb \in AnBn$

- Think of S as a variable representing an **arbitrary element**, and write these rules as

$$S \rightarrow \Lambda$$

$$S \rightarrow aSb$$

  In the process of obtaining an element of $AnBn$, S can be replaced by either string.

  Generating with recursive definition: Λ, *ab*, *aabb*, *aaabbb*

  Generating with new rules: *S*, *aSb*, *aaSbb*, *aaaSbbb*, *aaabbb*

Representing a Chain of Grammar Rules

- If α and β are strings, and α contains at least one occurrence of S, then

  $\alpha \Rightarrow \beta$ means that β is obtained from α in one step, by using either $S \to \Lambda$ or $S \to aSb$

- Example of generating *aaabbb*

  $$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

  i.e., we describe a *derivation* of the string *aaabbb*

- We can simplify the rules by using the | symbol to mean "or", so that the rules become

  $$\boldsymbol{S \to \Lambda \mid aSb}$$

Another example of grammar

- Consider the language *Expr* of legal algebraic expressions:
  - $a \in Expr$
  - For every $x, y \in Expr$, $x+y \in Expr$, and $x*y \in Expr$
  - For every $x \in Expr$, $(x) \in Expr$

- Think of S as a variable representing an **arbitrary element**, and write these rules as …

Another example of grammar

- Consider the language *Expr* of legal algebraic expressions:
  - $a \in Expr$
  - For every $x,y \in Expr$, $x+y \in Expr$, and $x*y \in Expr$
  - For every $x \in Expr$, $(x) \in Expr$

- Think of S as a variable representing an **arbitrary element**, and write these rules as ...

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

- Examples of <u>**different**</u> derivations of $a+a*a$

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

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

We can use more than one variable

- Recursive definition of *Expr* is
  - $a \in Expr$
  - For every $x,y \in Expr$, $x+y \in Expr$, and $x*y \in Expr$
  - For every $x \in Expr$, $(x) \in Expr$

- The grammar rules are

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

i.e., you cannot expand it recursively

But what if we want to use more than one "atomic" expression? For example, if we need identifiers *a*,*b* and also constants 120, 1.6E-2, then we can add one more variable and more rules

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

$$A \rightarrow a \mid b \mid 120 \mid 1.6E\text{-}2$$

Palindromes and Nonpalindromes

- Palindromes

Palindromes and Nonpalindromes

- Palindromes

$$S \to \Lambda|a|b|aSa|bSb$$

- Nonpalindromes (NonPal). The last two rules can still work if S is a nonpalindrome. Let us define NonPal

Palindromes and Nonpalindromes

- Palindromes

$$S \to \Lambda|a|b|aSa|bSb$$

- Nonpalindromes (NonPal). The last two rules can still work if S is a nonpalindrome. Let us define NonPal

    - For every $A \in \{a,b\}^*$, aAb, and bAa are in NonPal.

    - For every $S \in$ NonPal, aSa, and bSb are in NonPal.

and the rules are

$$\begin{array}{rcl} & & \overset{1}{}\quad \overset{2}{}\quad \overset{3}{}\quad \overset{4}{} \\ S & \to & aSa|bSb|aAb|bAa \\ A & \to & Aa|Ab|\Lambda \\ & & \underset{5}{}\quad \underset{6}{}\quad \underset{7}{} \end{array}$$

A derivation of $abbaaba$ is

$$S \overset{1}{\Rightarrow} aSa \overset{2}{\Rightarrow} abSba \overset{4}{\Rightarrow} abbAaba \overset{5}{\Rightarrow} abbAaaba \overset{7}{\Rightarrow} abbaaba$$

Definition of CFG

- **A *context-free grammar* (CFG) is a 4-tuple G=(V, Σ, S, P), where V and Σ are <u>disjoint finite sets</u>, $S \in V$, and P is a finite set of formulas of the form**

  $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$

Σ - set of terminal symbols or terminals (such as letters, i.e., something that cannot be divided, and recursively extended)

Example: in $S \to aSb$, a and b are terminals

***V* -** set of variables or nonterminals

Example: in $S \to aSbA$, S, and A are variables

$S \in V$ - start variable

***P* –** grammar rules (or productions), i.e., a subset of all possible strings made of terminals and nonterminals

- We use $\rightarrow$ for productions in a grammar and $\Rightarrow$ for a step in a derivation

  Grammar rules:

$$\begin{array}{lcl} S & \rightarrow & aSa|bSb|aAb|bAa \\ A & \rightarrow & Aa|Ab|\Lambda \end{array}$$

A derivation of $abbaaba$ is

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abbAaba \Rightarrow abbAaaba \Rightarrow abbaaba$$

- The notations $\alpha \Rightarrow^n \beta$ and $\alpha \Rightarrow^* \beta$ refer to exactly *n* steps and zero or more steps, respectively

  Example: $S \Rightarrow^3 abbAaba$

- Sometimes we will write $\alpha \Rightarrow_G \beta$ to indicate that a derivation involves productions of grammar *G*.

<u>Note</u>: we just learned how to code rules as strings, i.e., strings code not only the input but also an algorithm (rules).

- **Definition: If $G = (V, \Sigma, S, P)$ is a CFG, the language generated by G is**

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

  where S is the *start* variable, and x is a string of *terminals*.

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

<u>Example</u>: CFG for language $L1a = \{x \in \{a,b\}^* \mid n_a(x)=1\}$

- **Definition: If $G = (V, \Sigma, S, P)$ is a CFG, the language generated by G is**

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

  where S is the *start* variable, and x is a string of *terminals*.

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

<u>Example</u>: CFG for language $L1a = \{x \in \{a,b\}^* \mid n_a(x)=1\}$

Easy to see that any x in $L1a$ is a non-null string, so

$x = yaz$, where $y,z \in L_b = \{s \in \{b\}^*\}$

- We represent L_b by the variable B

- The productions for L_b and S are **...**

- **Definition: If $G = (V, \Sigma, S, P)$ is a CFG, the language generated by G is**

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

where S is the *start* variable, and x is a string of *terminals*.

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

<u>Example</u>: CFG for language $L1a = \{x \in \{a,b\}^* \mid \ n_a(x)=1\}$

Easy to see that any x in $L1a$ is a non-null string, so

$x = yaz$, where $\quad y,z \in L_b = \{s \in \{b\}^*\}$

- We represent L_b by the variable B

- The productions for L_b are $\boldsymbol{B \to \Lambda \mid bB}$

- All we need now is production for $L1a$

- $\boldsymbol{S \to BaB}$

```
<expression>    ::= <expression> + <term> |
                        <term>
<term>          ::= <term> * <factor> |
                        <factor>
<factor>        ::= (  <expression> )
                        | <name> | <integer>
<name>          ::= <letter> | <name> <letter> |
                        <name> <digit>
<integer>       ::= <digit> | <integer> <digit>
<letter>        ::= A | B | ... |Z
<digit>         ::= 0 | 1 | 2 | ... | 9
```

Warm up exercises

What language is generated by CFG?

- $S \rightarrow aS \mid bS \mid \Lambda$

Warm up exercises

What language is generated by CFG?

- $S \rightarrow aS \mid bS \mid \Lambda$

  (a+b)*

- $S \rightarrow SaS \mid b$

Warm up exercises

What language is generated by CFG?

- $S \rightarrow aS \mid bS \mid \Lambda$

  (a+b)*

- $S \rightarrow SaS \mid b$

  (ba)*b

- $S \rightarrow SaS \mid b \mid \Lambda$

Warm up exercises

What language is generated by CFG?

- $S \rightarrow aS \mid bS \mid \Lambda$

  (a+b)*

- $S \rightarrow SaS \mid b$

  (ba)*b

- $S \rightarrow SaS \mid b \mid \Lambda$

  strings not containing bb

- $$\begin{aligned} S &\rightarrow aT \mid bT \mid \Lambda \\ T &\rightarrow aS \mid bS \end{aligned}$$

Warm up exercises

What language is generated by CFG?

- $S \rightarrow aS \mid bS \mid \Lambda$

  (a+b)*

- $S \rightarrow SaS \mid b$

  (ba)*b

- $S \rightarrow SaS \mid b \mid \Lambda$

  strings not containing bb

- $$\begin{array}{lcl} S & \rightarrow & aT \mid bT \mid \Lambda \\ T & \rightarrow & aS \mid bS \end{array}$$

  even length strings in (a+b)*

Warm up exercises

- The set of odd-length strings in (a+b)* with middle *a*.

- The set of odd-length strings in (a+b)* with middle *a*.

$$S \;\rightarrow\; aSa \mid aSb \mid bSa \mid bSb \mid a$$

- The set of even-length strings in (a+b)* with the two middle symbols equal.

- The set of odd-length strings in (a+b)* with middle *a*.

$$S \rightarrow aSa \mid aSb \mid bSa \mid bSb \mid a$$

- The set of even-length strings in (a+b)* with the two middle symbols equal.

$$S \rightarrow aSa \mid aSb \mid bSa \mid bSb \mid aa \mid bb$$

- The set of odd-length strings in (a+b)* whose first, middle, and last symbols are all the same.

Warm up exercises

- The set of odd-length strings in (a+b)* with middle *a*.

$$S \rightarrow aSa \mid aSb \mid bSa \mid bSb \mid a$$

- The set of even-length strings in (a+b)* with the two middle symbols equal.

$$S \rightarrow aSa \mid aSb \mid bSa \mid bSb \mid aa \mid bb$$

- The set of odd-length strings in (a+b)* whose first, middle, and last symbols are all the same.

$$\begin{aligned} S &\rightarrow aTa \mid bUb \\ T &\rightarrow aTa \mid aTb \mid bTa \mid bTb \mid a \\ U &\rightarrow aUa \mid aUb \mid bUa \mid bUb \mid b \end{aligned}$$

<u>Example</u>: CFG for language $AEqB = \{x \in \{a,b\}^* \mid n_a(x) = n_b(x)\}$

If x is a non-null string in $AEqB$ then either

$x = ay$, where $y \in L_b = \{z \in \{a,b\}^* \mid n_b(z) = n_a(z) + 1\}$, or

$x = by$, where $y \in L_a = \{z \in \{a,b\}^* \mid n_a(z) = n_b(z) + 1\}$

Example: CFG for language $AEqB = \{x \in \{a,b\}^* \mid n_a(x) = n_b(x)\}$

If x is a non-null string in $AEqB$ then either

$x = ay$, where $y \in L_b = \{z \in \{a,b\}^* \mid n_b(z) = n_a(z) + 1\}$, or

$x = by$, where $y \in L_a = \{z \in \{a,b\}^* \mid n_a(z) = n_b(z) + 1\}$

- We represent L_b by the variable B and L_a by the variable A
- The productions so far are $S \to \Lambda \mid aB \mid bA$
- All we need now are productions for A and B

Example: CFG for language $AEqB = \{x \in \{a,b\}^* \mid n_a(x) = n_b(x)\}$

If x is a non-null string in $AEqB$ then either

$x = ay$, where $y \in L_b = \{z \in \{a,b\}^* \mid n_b(z) = n_a(z) + 1\}$, or

$x = by$, where $y \in L_a = \{z \in \{a,b\}^* \mid n_a(z) = n_b(z) + 1\}$

- We represent L_b by the variable B and L_a by the variable A

- The productions so far are $S \to \Lambda \mid aB \mid bA$

- All we need now are productions for A and B

- If $y \in L_a$ starts with b, then the remainder is in $AEqB$
- If it starts with a, the rest has two more a's than b's
- Observation: if in z $n_a(z) = n_b(z) + 2$ then $z=z_1z_2$ such that $n_a(z_1) = n_b(z_1) + 1$ and
  $n_a(z_2) = n_b(z_2) + 1$ (same for b, and a).

Example: CFG for language $AEqB = \{x \in \{a,b\}^* \mid n_a(x) = n_b(x)\}$

If x is a non-null string in $AEqB$ then either

$x = ay$, where $y \in L_b = \{z \in \{a,b\}^* \mid n_b(z) = n_a(z) + 1\}$, or

$x = by$, where $y \in L_a = \{z \in \{a,b\}^* \mid n_a(z) = n_b(z) + 1\}$

- We represent L_b by the variable B and L_a by the variable A

- The productions so far are $S \to \Lambda \mid aB \mid bA$

- All we need now are productions for A and B

- If $y \in L_a$ starts with b, then the remainder is in $AEqB$
- If it starts with a, the rest has two more a's than b's
- Observation: if in z $n_a(z) = n_b(z) + 2$ then $z=z_1z_2$ such that $n_a(z_1) = n_b(z_1) + 1$ and $n_a(z_2) = n_b(z_2) + 1$ (same for b, and a).

The resulting grammar is $\boldsymbol{S \to \Lambda \mid aB \mid bA}$

$\boldsymbol{B \to bS \mid aBB}$

$\boldsymbol{A \to aS \mid bAA}$

Example: CFG for all binary strings x with an even $n_0(x)$.

If the first symbol is a 1, then even number of 0's remains

$$S \rightarrow 1S$$

If the first symbol is a 0, then go to the next zero; the remainder is a string with even $n_0(x)$

$$S \rightarrow 1S \mid 0A0S \mid \Lambda$$

$$A \rightarrow 1A \mid \Lambda$$

A language can have more than one CFG …

If the first symbol is a 0, the remainder is a language with odd number of 0's

$$S \rightarrow 1S \mid 0T \mid \Lambda$$

$$T \rightarrow 1T \mid 0S$$

Example: CFG for the regular language corresponding to the RE 00*11*.

We can represent this language as a concatenation of two languages

$$S \rightarrow CD$$
$$C \rightarrow 0C \mid 0$$
$$D \rightarrow 1D \mid 1$$

Example: CFG for the complement of $\{0^i1^j \mid i,j>0\}$

There is no obvious way to convert a grammar to its complement.

We can represent this language as three languages

$$S \rightarrow A \mid B \mid C \mid \Lambda$$

Produces all strings with 10

$$A \rightarrow D10D$$

$$D \rightarrow 0D \mid 1D \mid \Lambda$$

Only zeros

$$B \rightarrow 0B \mid 0$$

Only ones

$$C \rightarrow 1C \mid 1$$

We can create many different CFLs with the following theorem

- **<u>Theorem</u>: If $\boldsymbol{L_1}$ and $\boldsymbol{L_2}$ are CFLs over $\boldsymbol{\Sigma}$, then so are $\boldsymbol{L_1 \cup L_2}$, $\boldsymbol{L_1 L_2}$, and $\boldsymbol{L_1{}^*}$**
- Suppose G_1 and G_2 are CFGs that generate L_1 and L_2 respectively, and assume that they have no variables in common
- Suppose that S_1 and S_2 are the start variables. S_u, S_c and S_k , the start variables of the new grammars (for <u>u</u>nion, <u>c</u>oncatenation, and <u>K</u>leene), will be new variables.

We can create many different CFLs with the following theorem

- **Theorem: If L_1 and L_2 are CFLs over Σ, then so are $L_1 \cup L_2$, L_1L_2, and $L_1{}^*$**
- Suppose G_1 and G_2 are CFGs that generate L_1 and L_2 respectively, and assume that they have no variables in common
- Suppose that S_1 and S_2 are the start variables. S_u, S_c and S_k, the start variables of the new grammars (for union, concatenation, and Kleene), will be new variables.
  - G_u just adds the rules $S_u \to S_1 \mid S_2$ to G_1 and G_2
  - G_c just adds the rule $S_c \to S_1S_2$ to G_1 and G_2
  - G_k just adds the rules $S_k \to \Lambda \mid S_kS_1$ to G_1

Read and learn the proof of this theorem! See Theorem 4.9 in the textbook.

Union of two CFLs

Language	Grammar
$L_1 = \{x \in \{a,b\}^* \mid x = a^n b^n,\ n \in \mathbb{N}\}$	$G_1 = (V_1, \Sigma, S_1, P_1)$ $V_1 = \{S_1\}$ $\Sigma = \{a, b\}$ $P_1: \quad S_1 \ \rightarrow\ \Lambda \mid aS_1b$
$L_2 = \{x \in \{a,b\}^* \mid x = wr(w)\}$	$G_2 = (V_2, \Sigma, S_2, P_2)$ $V_2 = \{S_2\}$ $\Sigma = \{a, b\}$ $P_2: \quad S_2 \ \rightarrow\ \Lambda \mid aS_2a \mid bS_2b$
$L = L_1 \cup L_2$	$G = (V, \Sigma, S, P)$ $V = \{S, S_1, S_2\}$ $\Sigma = \{a, b\}$ $P: \quad S \ \rightarrow\ S_1 \mid S_2$ $\quad\quad S_1 \ \rightarrow\ \Lambda \mid aS_1b$ $\quad\quad S_2 \ \rightarrow\ \Lambda \mid aS_2a \mid bS_2b$

Concatenation of two CFLs

Language	Grammar
$L_1 = \{x \in \{a,b\}^* \mid x = a^n b^n,\ n \in \mathbb{N}\}$	$G_1 = (V_1, \Sigma, S_1, P_1)$ $V_1 = \{S_1\}$ $\Sigma = \{a,b\}$ $P_1: \quad S_1 \rightarrow \Lambda \mid aS_1b$
$L_2 = \{x \in \{a,b\}^* \mid x = wr(w)\}$	$G_2 = (V_2, \Sigma, S_2, P_2)$ $V_2 = \{S_2\}$ $\Sigma = \{a,b\}$ $P_2: \quad S_2 \rightarrow \Lambda \mid aS_2a \mid bS_2b$
$L = L_1L_2$	$G = (V, \Sigma, S, P)$ $V = \{S, S_1, S_2\}$ $\Sigma = \{a,b\}$ $P: \quad S \rightarrow S_1S_2$ $\quad\quad S_1 \rightarrow \Lambda \mid aS_1b$ $\quad\quad S_2 \rightarrow \Lambda \mid aS_2a \mid bS_2b$

Closure of CFL

Language	Grammar
$L_2 = \{x \in \{a,b\}^* \mid x = wr(w)\}$	$G_2 = (V_2, \Sigma, S_2, P_2)$ $V_2 = \{S_2\}$ $\Sigma = \{a, b\}$ $P_2 : \quad S_2 \; \rightarrow \; \Lambda \mid aS_2a \mid bS_2b$
$L = L_2^*$	$G = (V, \Sigma, S, P)$ $V = \{S, S_2\}$ $\Sigma = \{a, b\}$ $P : \quad S \; \rightarrow \; \Lambda \mid SS_2$ $\quad\quad S_2 \; \rightarrow \; \Lambda \mid aS_2a \mid bS_2b$

Intersection of two CFLs is not necessarily CFL

Language	Grammar
$L_1 = \{a^n b^n c^m \mid n, m \in \mathbb{N}\}$	$G_1 = (V_1, \Sigma, S_1, P_1)$ $V_1 = \{S, A, C\}$ $\Sigma = \{a, b, c\}$ $P_1: \quad S \rightarrow AC$ $A \rightarrow aAb \mid \Lambda$ $C \rightarrow cC \mid \Lambda$
$L_2 = \{a^n b^m c^m \mid n, m \in \mathbb{N}\}$	$G_2 = (V_2, \Sigma, S_2, P_2)$ $V_2 = \{S, A, B\}$ $\Sigma = \{a, b, c\}$ $P_2: \quad S \rightarrow AB$ $A \rightarrow aA \mid \Lambda$ $B \rightarrow bBc \mid \Lambda$
$L = L_1 \cap L_2 = \{a^n b^n c^n \mid n, m \in \mathbb{N}\}$	Not CFL

Complement of CFL is not necessarily CFL

Proof. Assume the complement of every CFL is a CFL. Let L_1 and L_2 be CFLs. Since CFLs are closed under union, and we are assuming they are closed under complement then

$$\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2,$$

that is a contradiction to our previous example. $\square$

Let us see if we can construct CFG from a regular expression.
Let $L \subseteq \{a,b\}^*$ be the language of $bba(ab)^* + (ab + ba^*b)^*ba$

- We can apply $\cup, \cdot,$ and $*$ rules from the theorem but it is lengthy (separate variables for $\{a\}, \{b\}, \ldots$).
- L is a union of L_1, and L_2, i.e. $S \rightarrow S_1 | S_2$.
- For L_1 we introduce $S_1 \rightarrow S_1ab | bba$
- For L_2 we also introduce variable T for $ab + ba^*b$, and the productions

$$\begin{array}{rcl} S_2 & \rightarrow & TS_2|ba \\ T & \rightarrow & ab|bUb \\ U & \rightarrow & \Lambda|aU \end{array}$$

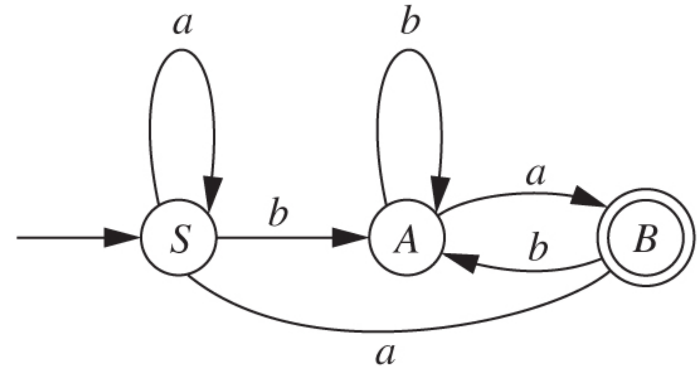
We don't need Lambda for * because there is a correct derivation

We can prove that every regular language can be generated by a context-free grammar.

Moreover, if you have an FA, you can create a context-free grammar.

- States correspond to variables
- For each transition we introduce a rule
- We also need to add termination rules

*L={a,b}*ba*



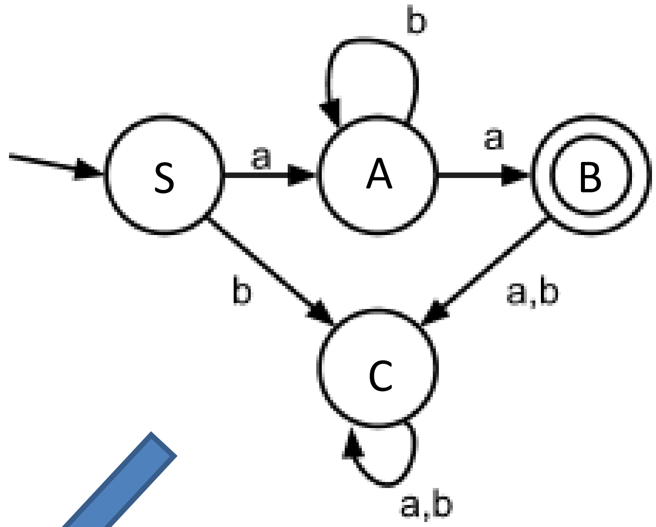
$$
\begin{aligned}
S &\rightarrow aS|bA \\
A &\rightarrow bA|aB \\
B &\rightarrow bA|aS|\Lambda
\end{aligned}
$$

- **Definition: <u>A context-free grammar is regular</u> (RG) if every production is of the form $A \to \sigma B$ or $A \to \Lambda$**
- **Theorem:** For every language $L \subseteq \Sigma^*$,

L is regular **if and only if** $L = L(G)$ for some regular grammar G
- Proof: <u>L is regular => L=$L(G)$ for some RG</u>
  - If L is regular, then there is an FA M=(Q, Σ, q_0, A, δ) that accepts it. Define G=(V, Σ, S, P) by
    - letting V be Q,
    - S the initial state q_0, and
    - P the set containing the production $T \to aU$ for every transition $\delta(T, a) = U$ in M, and
    - the production $T \to \Lambda$ for every accepting state T of M.
  - G is RG, and G accepts the same language as M

  For every $x = a_1a_2...a_n$ in L, the transitions on these symbols that start at q_0 end at an accepting state if and only if there is a derivation of x in G (show by induction on $|x| = n$). ← **<u>Prove at home!</u>**

Regular expression
*ab*a*



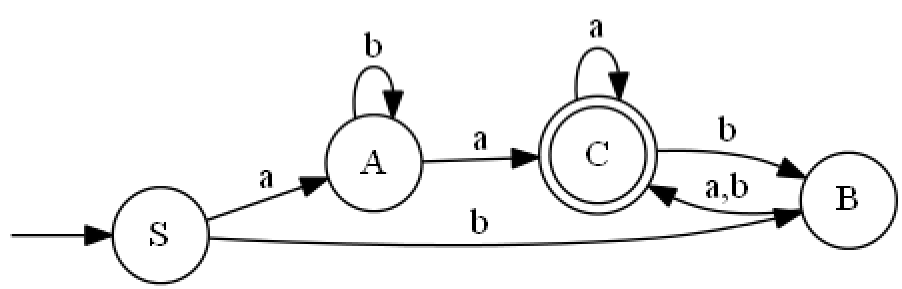
S → aA | bC
A → aB | bA
B → aC | bC | Λ
C → aC | bC

- Proof (part 2): *L* is regular <= *L=L(G)* for some RG

To prove the other direction we can start with a regular grammar G and reverse the construction to produce M.

M may be an NFA, but it still accepts $L(G)$, and it follows that $L(G)$ is regular.

$$
\begin{array}{ccc}
S & \rightarrow & aA|bB \\
A & \rightarrow & aC|bA \\
C & \rightarrow & aC|bB|\Lambda \\
B & \rightarrow & aC|bC
\end{array}
$$



- **Definition: <u>A context-free grammar is right-regular</u> (RRG) if every production is of the form $A \to B\sigma$ or $A \to \Lambda$**

- **Theorem: For every language $L \subseteq \Sigma^*$, L is regular if and only if $L = L(G)$ for some right-regular grammar G**

Proof: <u>*L* is regular <= *L=L(G)* for some RRG</u>

Grammar G contains rules of the form

$$A \to B\sigma \text{ or } A \to \Lambda,$$

i.e., all $x \in L(G)$ are reversed strings of the language generated by the reversed rules of the form $A \to \sigma B$. Therefore, $L(G) = r(L(G'))$, where G' are the reverse rules of G.

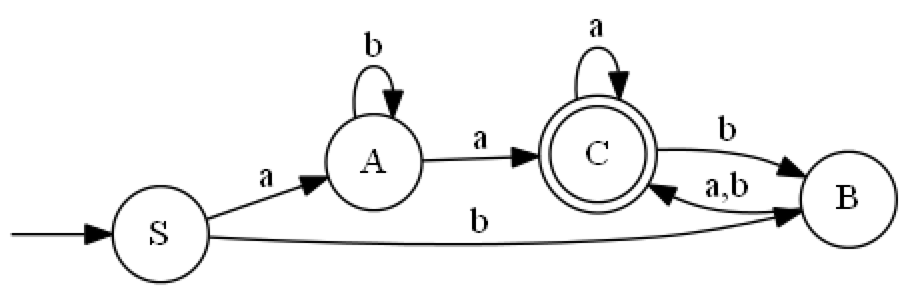
But we know that G' is regular, and so is $L(G')$, i.e., the reverse of L is regular. Can we prove that if L is regular then $r(L)$ is also regular?

reverse of L

Theorem. *If L is regular then $r(L)$ is also regular.*

Proof.

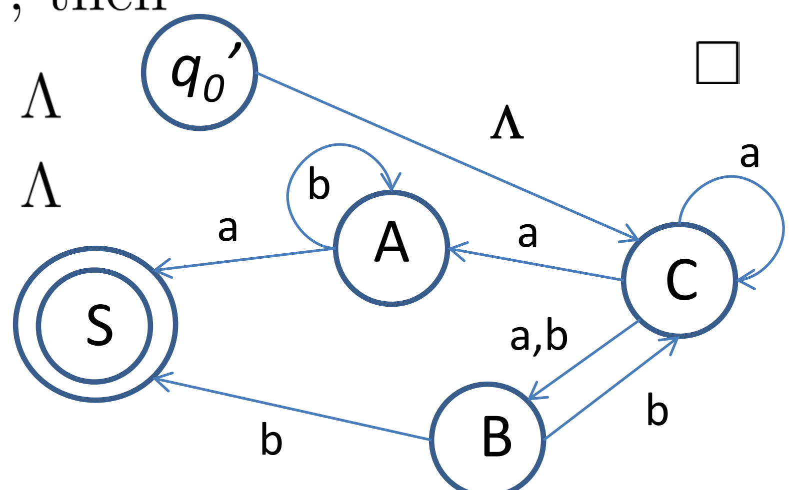
- If L is regular we can find its FA $M = (Q, \Sigma, q_0, A, \delta)$.
- We will construct NFA $M' = (Q', \Sigma, q_0', A', \delta')$ that recognizes $r(L)$. Then, M' will be converted into deterministic FA, and this completes the proof.



- $Q' = Q \cup \{q_0'\}$
- $A' = \{q_0\}$
- Define $\delta^{-1}(q, a) = \{q' \mid \delta(q', a) = q\}$, then

$$\delta'(q, \sigma) = \begin{cases} A & \text{if } q = q_0' \text{ and } \sigma = \Lambda \\ \delta^{-1}(q, \sigma) & \text{if } q \neq q_0' \text{ and } \sigma \neq \Lambda \end{cases}$$

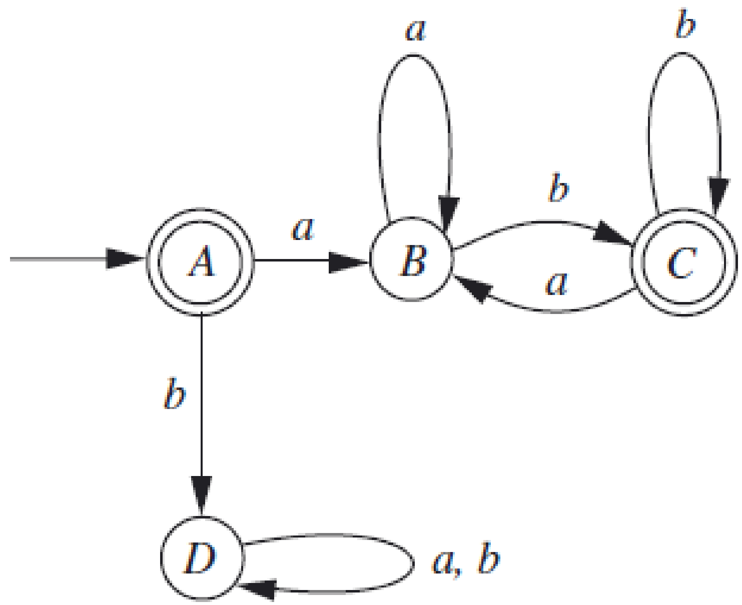
$\square$



- **Definition: <u>A context-free grammar is right-regular</u> (RRG) if every production is of the form $A \to B\sigma$ or $A \to \Lambda$**
- **Theorem: For every language $L \subseteq \Sigma^*$, L is regular if and only if $L = L(G)$ for some right-regular grammar G**
- Proof: <u>*L* is regular *=> L=L(G)* for some RRG</u>

L is regular $\Rightarrow r(L)$ is regular $\Rightarrow$ we can find (left) regular G' such that $G'(r(L))$ is regular $\Rightarrow$ we can reverse all rules of G' to get RRG G

Find a regular grammar generating the language *L(M)*

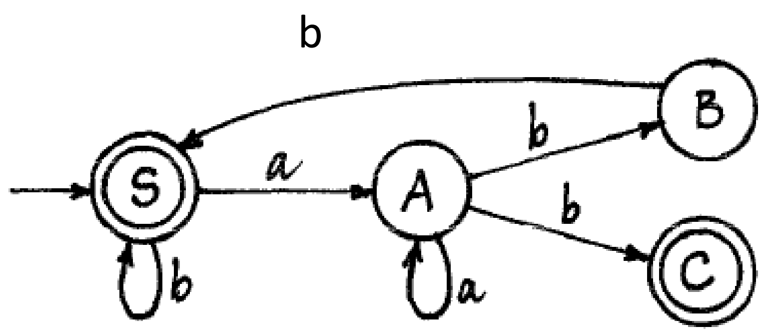


$A \rightarrow aB \mid \Lambda \qquad B \rightarrow aB \mid bC \qquad C \rightarrow aB \mid bC \mid \Lambda$. We could also add the productions $A \rightarrow bD \qquad D \rightarrow aD \mid bD$, but they are not necessary, because no string of terminals can be obtained from a sequence of steps in which the variable D appears.

Find an NFA for this grammar

$$S \to bS \mid aA \mid \Lambda \qquad A \to aA \mid bB \mid b \qquad B \to bS$$

In this example, we could accommodate the production $A \to b$ by making the state corresponding to B in our diagram accepting, since the only other production with B on the right side also has b preceding it. However, a more appropriate technique in general is to replace $A \to b$ by the two productions $A \to bC$ $\quad C \to \Lambda$, where C is a variable used only for this purpose. The resulting NFA looks like



Derivation Trees

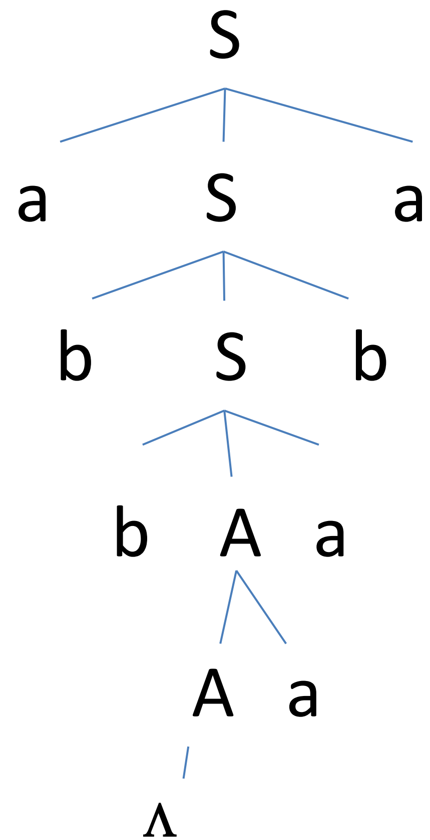
- The root node represents the start variable S
- Any interior node and its children represent a production $A \to \alpha$ used in the derivation; the node represents A, and the children, from left to right, represent the symbols in α (can include variables, and terminals).
- Each leaf node represents a terminal or Λ
- The string derived is read off from left to right, ignoring Λ's
- Every derivation has exactly one derivation tree, but a tree can represent more than one derivation

Grammar rules:

$$
\begin{aligned}
S &\rightarrow aSa|bSb|aAb|bAa \\
A &\rightarrow Aa|Ab|\Lambda
\end{aligned}
$$

A derivation of *abbaaba* is

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abbAaba \Rightarrow abbAaaba \Rightarrow abbaaba$$



Derivation Trees and Ambiguity

- So far we've been interested in *what* strings a CFG generates (e.g., S=> ….. => ababa)
- It is also useful to consider *how* a string is generated by a CFG
- A derivation may provide information about the structure of a string, and if a string has several possible derivations, one may be more appropriate than another
- We can draw trees to represent derivations

Derivation Trees and Ambiguity

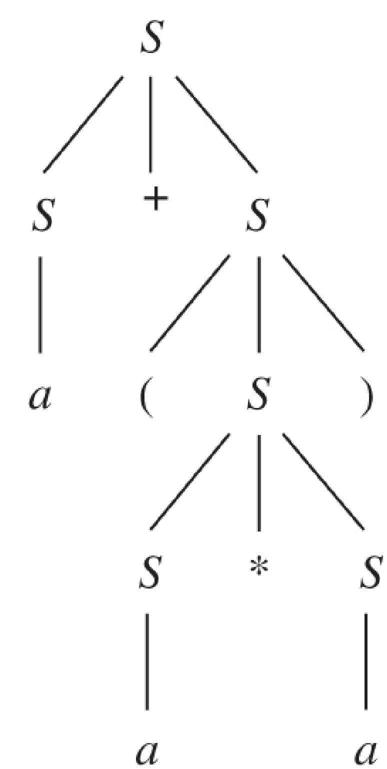
- Definition: **Ambiguity in a CFG**

A context-free grammar G is *ambiguous* if for at least one $x \in L(G)$, x has more than one derivation tree.

- Consider the language *Expr* of legal algebraic expressions:
  - $a \in Expr$
  - For every $x,y \in Expr$, $x+y \in Expr$, and $x*y \in Expr$
  - For every $x \in Expr$, $(x) \in Expr$

- CFG rules are $S \rightarrow a \mid S+S \mid S*S \mid (S)$

- Examples of **different** derivations of $a+(a*a)$ for the **same** derivation tree
  - $S \Rightarrow S+S \Rightarrow a+S \Rightarrow a+(S) \Rightarrow a+(S*S) \Rightarrow a+(S*a) \Rightarrow a+(a*a)$
  - $S \Rightarrow S+S \Rightarrow S+(S) \Rightarrow S+(S*S) \Rightarrow a+(S*S) \Rightarrow a+(a*S) \Rightarrow a+(a*a)$



Derivation Trees and Ambiguity

- In a derivation, at each step some production is applied to some occurrence of a variable

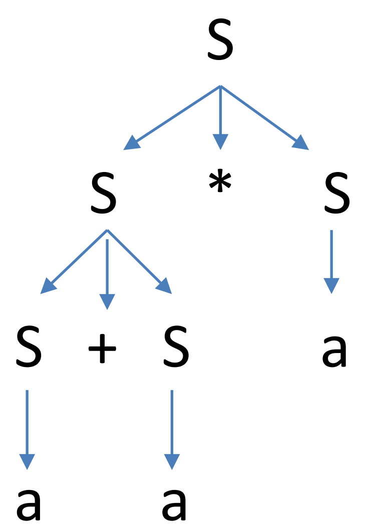
- Consider a derivation that starts $S \Rightarrow S + S$. We could apply a production to either the first or second of the S's, but the resulting trees would be the same

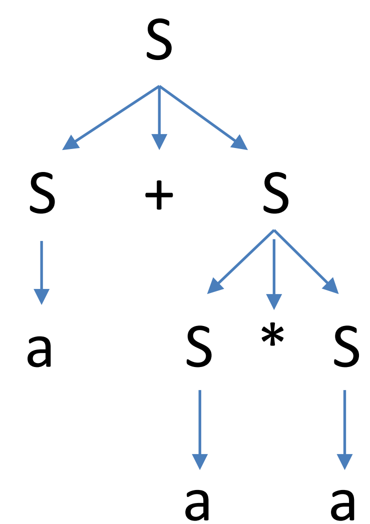
- <u>The order in which the tree was created is also important for evaluation.</u>

$$S \Rightarrow S+S \Rightarrow a+S \Rightarrow a+(S) \Rightarrow a+(S*S) \Rightarrow a+(S*a) \Rightarrow a+(a*a)$$

$$S \Rightarrow S+S \Rightarrow S+(S) \Rightarrow S+(S*S) \Rightarrow a+(S*S) \Rightarrow a+(a*S) \Rightarrow a+(a*a)$$

- Consider the language *Expr* of legal algebraic expressions:
  - $a \in Expr$
  - For every $x,y \in Expr$, $x+y \in Expr$, and $x^*y \in Expr$
  - For every $x \in Expr$, $(x) \in Expr$
- CFG rules are $S \rightarrow a \mid S+S \mid S^*S \mid (S)$
- Examples of **<u>different</u>** derivation trees for string a+a*a



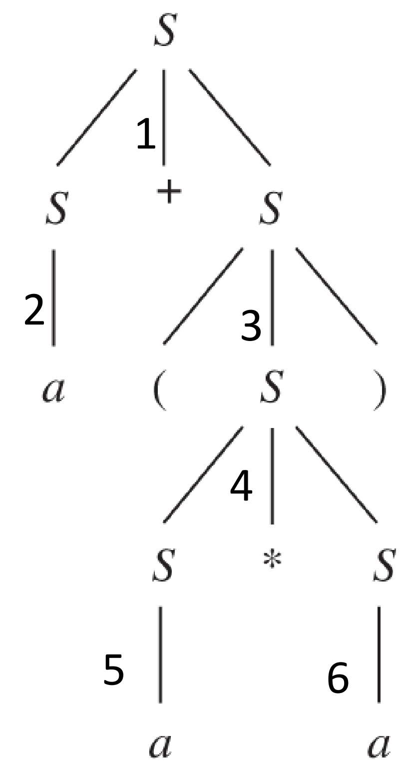


$$S \Rightarrow S^*S \Rightarrow S+S^*S \Rightarrow a+S^*S \Rightarrow a+a^*S \Rightarrow a+a^*a$$

$$S \Rightarrow S+S \Rightarrow S+S^*S \Rightarrow a+S^*S \Rightarrow a+a^*S \Rightarrow a+a^*a$$

Derivation Trees and Ambiguity

- **Definition: A derivation in a CFG is a *leftmost* derivation (LMD) if, at each step, a production is applied to the leftmost variable-occurrence in the current string**
  - A rightmost derivation is defined similarly



- CFG rules are $S \rightarrow a \mid S+S \mid S*S \mid (S)$
- Examples of **<u>different</u>** derivations of $a+(a*a)$
  - $S \Rightarrow S+S \Rightarrow a+S \Rightarrow a+(S) \Rightarrow a+(S*S) \Rightarrow a+(a*S) \Rightarrow a+(a*a)$
  - ~~$S \Rightarrow S+S \Rightarrow S+(S) \Rightarrow \ldots$~~

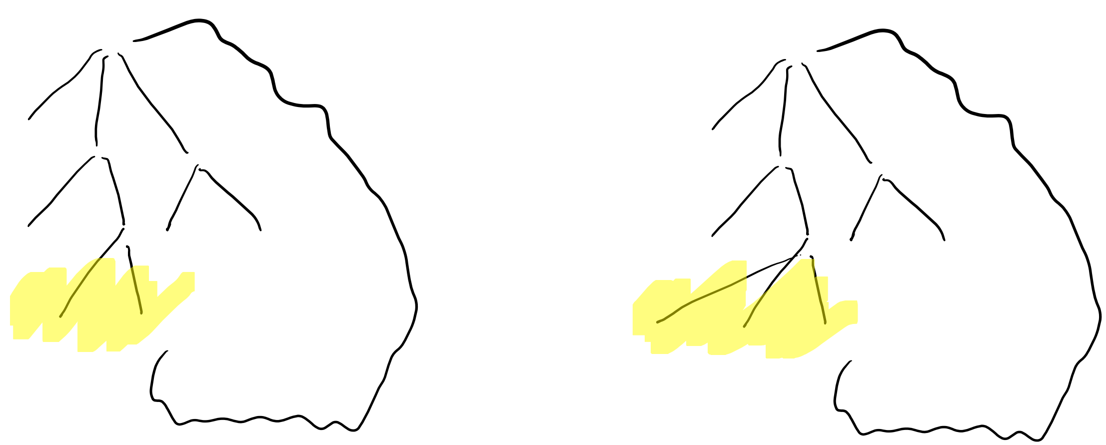
This derivation is not LMD

Theorem. *If G is CFG then for every $x \in L(G)$, these three statements are equivalent*

1. *x has more than one derivation tree.*
2. *x has more than one leftmost derivation.*
3. *x has more than one rightmost derivation.*

Proof. We will show $1 \Leftrightarrow 2$.

- Part I, $1 \Rightarrow 2$. Consider x with two different derivation trees $\Rightarrow$ these trees have two LMD that must be different because if they are not, then their trees are equal.



$\square$

Theorem. *If G is CFG then for every $x \in L(G)$, these three statements are equivalent*

1. *x has more than one derivation tree.*

2. *x has more than one leftmost derivation.*

3. *x has more than one rightmost derivation.*

Proof. We will show $1 \Leftrightarrow 2$.

- Part II, $1 \Leftarrow 2$. Consider x with two different LMD. Their corresponding trees are T_1, and T_2. Suppose that the first step where they are different is

  strings

  $$xA\beta \Rightarrow x\alpha_1\beta \text{ in LMD1, and } xA\beta \Rightarrow x\alpha_2\beta \text{ in LMD2}$$

  variable

  In both T_1, and T_2 there is a node corresponding to A, and these nodes have different children (because $\alpha_1 \neq \alpha_2$) which makes T_1, and T_2 different.

$\square$

Show that the CFG with productions

$$S \;\rightarrow\; a \mid Sa \mid bSS \mid SSb \mid SbS$$

is ambiguous.

Show that the CFG with productions

$$S \rightarrow a \mid Sa \mid bSS \mid SSb \mid SbS$$

is ambiguous.

By definition, grammar G is ambiguous if we can find at least one string with more than one derivation tree. The string $abaa$ has two leftmost derivations, one starting with

$$S \Rightarrow SbS \Rightarrow abS \Rightarrow abSa \Rightarrow abaa,$$

the other with

$$S \Rightarrow Sa \Rightarrow SbSa \Rightarrow abSa \Rightarrow abaa.$$

By using the previous theorem we can state that $abaa$ has more than one derivation tree.

Consider the context-free grammar with productions

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA \mid \Lambda \\ B &\rightarrow ab \mid bB \mid \Lambda \end{aligned}$$

Every derivation of a string in this grammar must begin with the production $S \rightarrow AB$. Clearly, any string derivable from A has only one derivation from A, and likewise for B. Therefore, the grammar is unambiguous.

True or false? Why?

Consider the context-free grammar with productions

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA \mid \Lambda \\ B &\rightarrow ab \mid bB \mid \Lambda \end{aligned}$$

Every derivation of a string in this grammar must begin with the production $S \rightarrow AB$. Clearly, any string derivable from A has only one derivation from A, and likewise for B. Therefore, the grammar is unambiguous.
True or false? Why?

False. Although for each string derivable from A there is only one recursive derivation, and similarly for B, there may be more than one choice for certain strings. For example, there is a derivation of ab in which a is derived from A and b from B, and there is another derivation in which Λ is derived from A and ab from B.

Derivation Trees and Ambiguity

- A classic example of ambiguity is the <u>dangling *else*</u>
- In C, an if-statement can be defined by

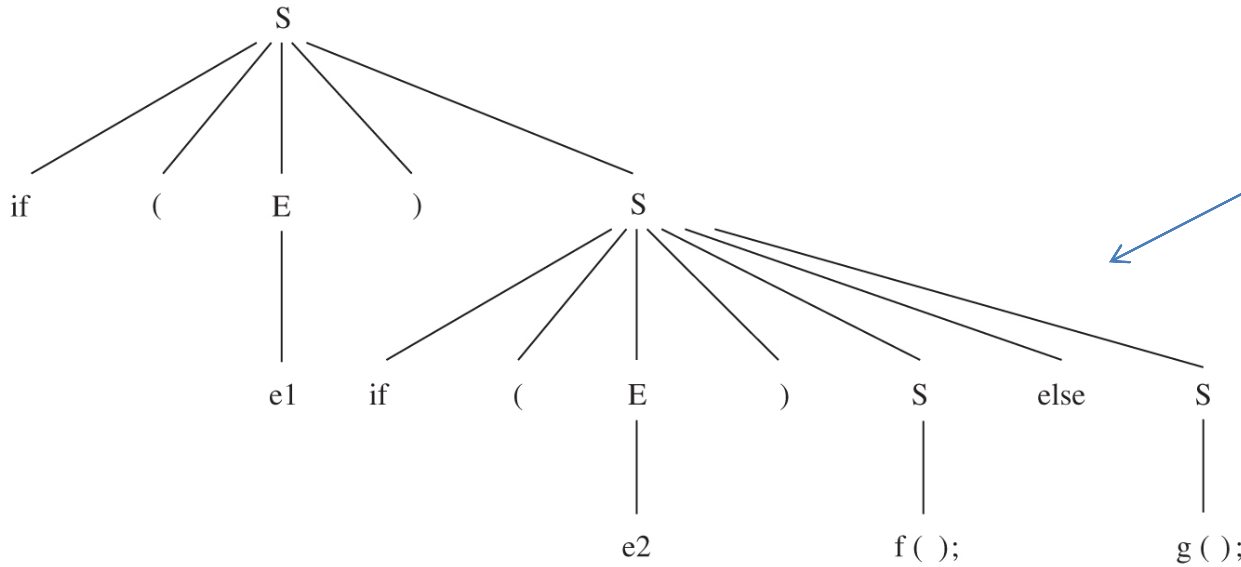
  $\boldsymbol{S \to \textbf{if}\ (\ E\ )\ S \mid \textbf{if}\ (\ E\ )\ S\ \textbf{else}\ S \mid OS}$

  (*OS*="other statements", E="expression", S="statement")
- Consider the statement string

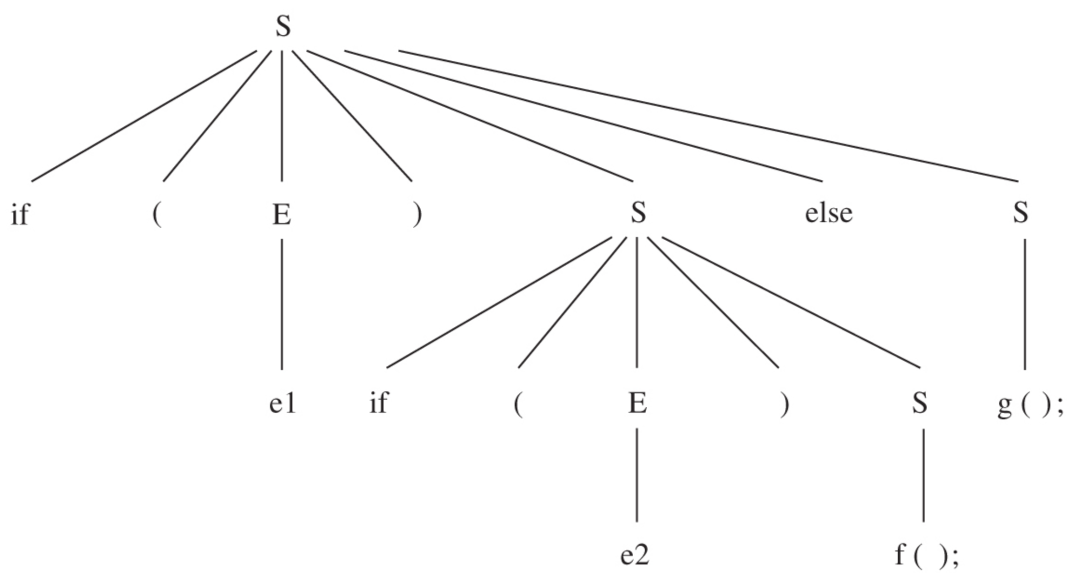
  if (e1) if (e2) f(); else g();

  - **In C, the *else* to belong to the second *if*, but this grammar does not rule out the other interpretation**
- The two derivation trees shown on the next slide demonstrate the two interpretations of a dangling *else*

preferred



(a)



(b)

Derivation Trees and Ambiguity

- Clearly the grammar

$$S \to \text{if } (E) \ S \ | \ \text{if } (E) \ S \text{ else } S \ | \ OS$$

is ambiguous, but there are equivalent grammars that allow only the correct interpretation

- Example:

$$S \to S_1 \mid S_2$$
$$S_1 \to \text{if } (E) \ S_1 \text{ else } S_1 \mid OS$$
$$S_2 \to \text{if } (E) \ S \mid \text{if } (E) \ S_1 \text{ else } S_2$$

Derivation Trees and Ambiguity

Eliminate ambiguity with the following grammar:

$$S \rightarrow S_1 \mid S_2$$
$$S_1 \rightarrow \text{if } (\, E \,)\; S_1 \text{ else } S_1 \mid OS$$
$$S_2 \rightarrow \text{if } (\, E \,)\; S \mid \text{if } (\, E \,)\; S_1 \text{ else } S_2$$

- S_1 represents a statement in which every *if* is matched by a corresponding *else*
- Every statement derived from S_2 contains at least one unmatched *if*.
- The only variable appearing before *else* in these rules is S_1; because the *else* cannot match any of the *if* s in the statement derived from S_1, it must match the *if* that appeared at the same time it did.

Derivation Trees and Ambiguity

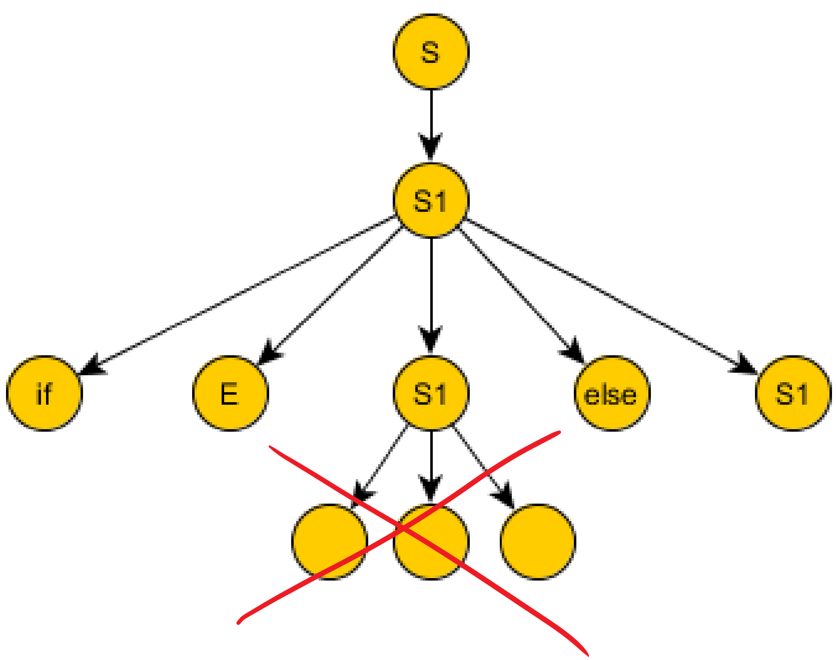
Eliminate ambiguity with the following grammar:

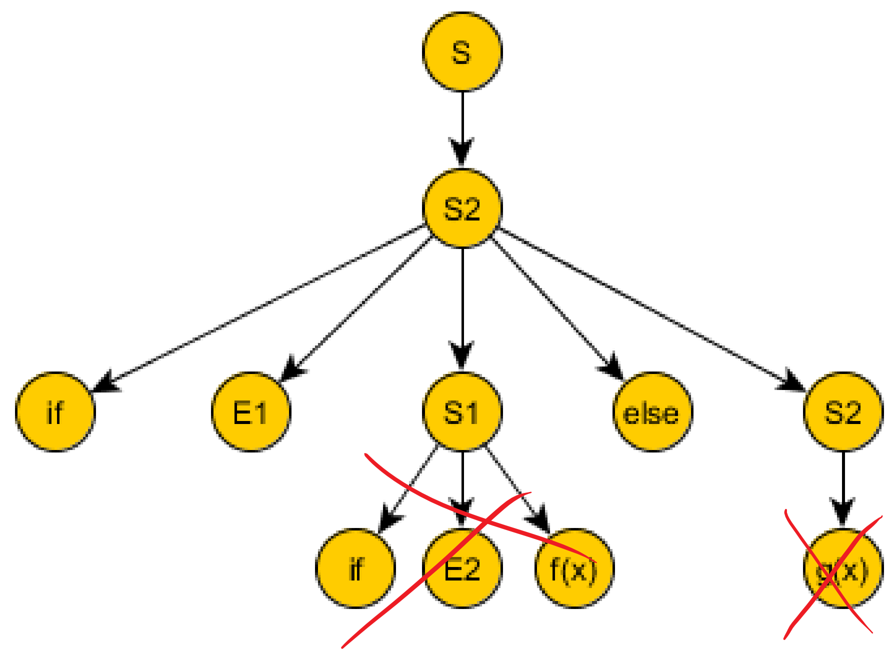
if (e1) if (e2) f(x); else g(x);

$$S \to S_1 \mid S_2$$

$$S_1 \to \text{if } (\ E\ )\ S_1 \text{ else } S_1 \mid OS$$

$$S_2 \to \text{if } (\ E\ )\ S \mid \text{if } (\ E\ )\ S_1 \text{ else } S_2$$



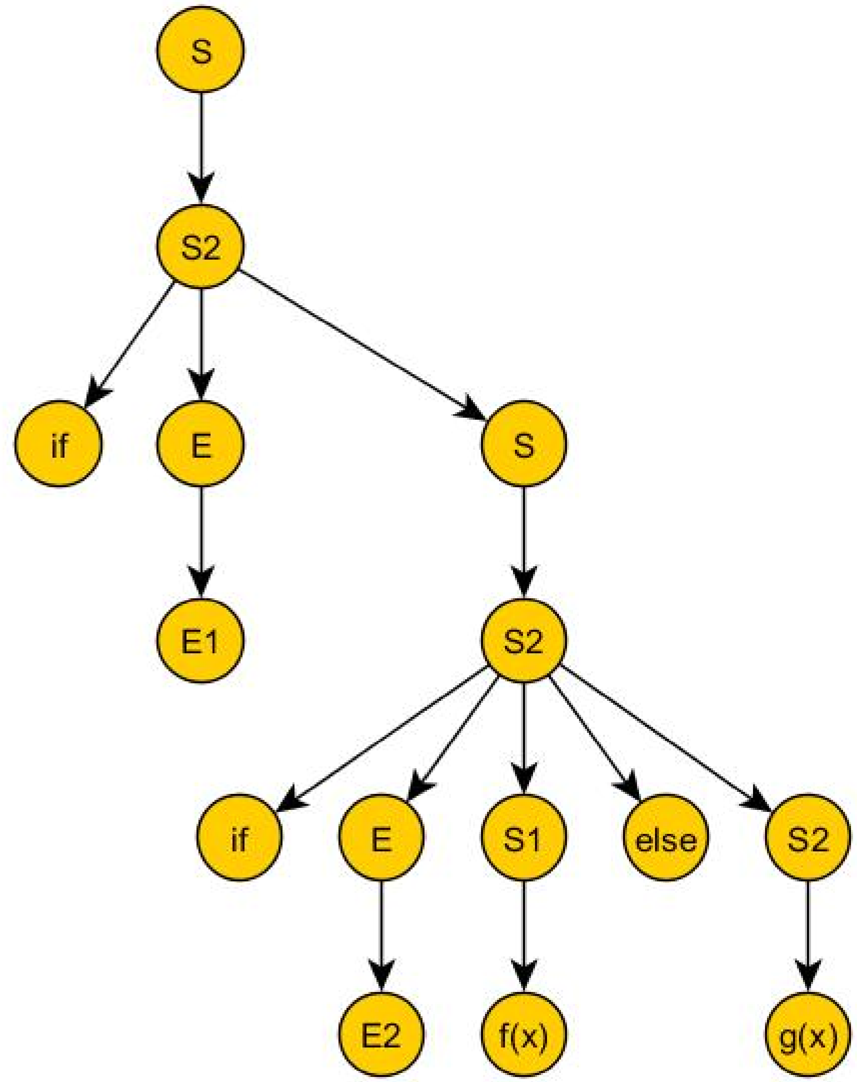


Derivation Trees and Ambiguity

Eliminate ambiguity with the following grammar:

if (e1) if (e2) f(x); else g(x);

$$S \rightarrow S_1 \mid S_2$$
$$S_1 \rightarrow \text{if } (\ E\ )\ S_1 \text{ else } S_1 \mid OS$$
$$S_2 \rightarrow \text{if } (\ E\ )\ S \mid \text{if } (\ E\ )\ S_1 \text{ else } S_2$$



Derivation Trees and Ambiguity

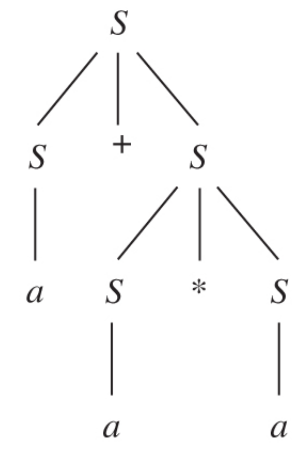
Consider the CFG $G: S \to S + S \mid S * S \mid (S) \mid a$

- G generates simple algebraic expressions
- One reason for ambiguity is unspecified precedence of + and *: a+a*a could be interpreted as (a+a)*a or as a+(a*a)
- In fact, $S \to S + S$ causes ambiguity by itself, because a+a+a could be interpreted as either (a+a)+a or a+(a+a). Similarly for $S \to S * S$
- We might try to correct both problems by using the productions $S \to S + T \mid T \quad T \to T + F \mid F$

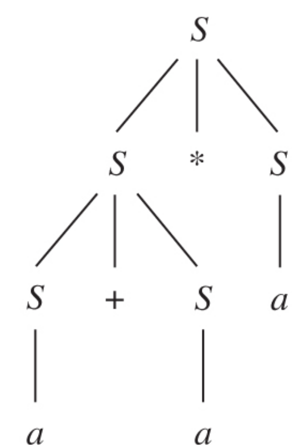
(think of T as "term" and F as "factor")

Derivation Trees and Ambiguity

Derivation trees for
$a + a * a$



(a)



(b)

- $*$ now has higher precedence than + (all the multiplications are performed *within* a term)
- By making the production $S \to S + T$, not $S \to T + S$, we make + associate to the left. Similarly for $*$
- We want parenthetical expressions to be evaluated first; this means we should consider such an expression to be part of a factor. The resulting unambiguous CFG generating $L(G)$ is

$$S \to S + T \mid T \qquad T \to T * F \mid F \qquad F \to (S) \mid a$$

(proofs of unambiguity and equivalence are both somewhat complicated)

Show that the context-free grammar is ambiguous and find an equivalent unambiguous grammar

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

The string aaa has two different leftmost derivations. An equivalent unambiguous grammar is

$$S \rightarrow aS \mid bS \mid \Lambda.$$

Is it possible to find regular ambiguous CFG?

Yes. For example,

$$\begin{aligned} S &\rightarrow aA \mid aB \\ A &\rightarrow \Lambda \\ B &\rightarrow \Lambda \end{aligned}$$

String a has two leftmost derivations.

Show that the context-free grammar is ambiguous and find an equivalent unambiguous grammar

$$\begin{aligned} S &\rightarrow ABA \\ A &\rightarrow aA \mid \Lambda \\ B &\rightarrow bB \mid \Lambda \end{aligned}$$

The string a can be derived $S \Rightarrow ABA \Rightarrow^2 aBA \Rightarrow^* a$ or $S \Rightarrow ABA \Rightarrow BA \Rightarrow A \Rightarrow^2 a$.

It's easy to see, that any string with at least one b has only one leftmost derivation.

Therefore, an equivalent unambiguous grammar is

$$\begin{aligned} S &\rightarrow A \mid ABA \\ A &\rightarrow aA \mid \Lambda \\ B &\rightarrow bB \mid b. \end{aligned}$$

Definition.

If L is a context-free language for which there exists an unambiguous grammar then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

Example: $L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}$, where n, m are nonnegative. This language is inherently ambiguous.
Easy to see that L is context-free. It is represented as

$$L = L_1 \cup L_2,$$

where L, L_1, and L_2 are generated by ...

$$\begin{array}{rcl} S & \to & S_1 \mid S_2 \end{array} \qquad \begin{array}{rcl} S_1 & \to & S_1 c \mid A \\ A & \to & aAb \mid \Lambda \end{array} \qquad \begin{array}{rcl} S_2 & \to & aS_2 \mid B \\ B & \to & bBc \mid \Lambda \end{array}$$

String $a^n b^n c^n$ has two distinct derivations, one starting with $S \Rightarrow S_1$, the other with $S \Rightarrow S_2$. It is not a proof but in some way L_1, and L_2 have conflicting requirements on equal number of a, b, and b, c.

Example of languages that cannot be inherently ambiguous: regular languages. Regular grammar can be ambiguous, but we can always eliminate it.

Question: Give an algorithm to check whether a CFG is ambiguous.

The problem is undecidable: there is no general algorithm to check whether a given context-free grammar is ambiguous. Some CFLs are inherently ambiguous.

This does not mean there aren't classes of grammars where an answer is possible.