CSE 211 (Theory of Computation)

Context Free Languages

Dr. Muhammad Masroor Ali

Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

January 2023

Version: 3.2, Last modified: July 19, 2023

Context-Free Grammars

Sipser, 2.1, p-102

- Grammar, G_1.
- $$A \rightarrow 0A1$$
  $$A \rightarrow B$$
  $$B \rightarrow \#$$

Context-Free Grammars

Sipser, 2.1, p-102

- Grammar, G_1.
- $$A \rightarrow 0A1$$
  $$A \rightarrow B$$
  $$B \rightarrow \#$$
- - Substitution rules, also called productions, or production rules.
  - Variables or non-terminal symbols.
  - Terminals or terminal symbols.
  - Start variable or start symbol.

Context-Free Grammars

Sipser, 2.1, p-102

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

- Grammar G_1 generates the string $000\#111$.
- The sequence of substitutions to obtain a string is called a derivation.

Context-Free Grammars

Sipser, 2.1, p-102

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

- A derivation of string $000\#111$ in grammar G_1 is

$$A \xRightarrow{A \rightarrow 0A1} 0A1$$
$$\xRightarrow{A \rightarrow 0A1} 00A11$$
$$\xRightarrow{A \rightarrow 0A1} 000A111$$
$$\xRightarrow{A \rightarrow B} 000B111$$
$$\xRightarrow{B \rightarrow \#} 000\#111$$

Context-Free Grammars — *continued*

Sipser, 2.1, p-102

$$\begin{aligned} A &\xRightarrow{A\to 0A1} 0A1 \\ &\xRightarrow{A\to 0A1} 00A11 \\ &\xRightarrow{A\to 0A1} 000A111 \\ &\xRightarrow{A\to B} 000B111 \\ &\xRightarrow{B\to \#} 000\#111 \end{aligned}$$

- You may also represent the same information pictorially with a parse tree.

- You may also represent the same information pictorially with a parse tree.

$$
\begin{aligned}
A &\xRightarrow{A \to 0A1} 0A1 \\
&\xRightarrow{A \to 0A1} 00A11 \\
&\xRightarrow{A \to 0A1} 000A111 \\
&\xRightarrow{A \to B} 000B111 \\
&\xRightarrow{B \to \#} 000\#111
\end{aligned}
$$

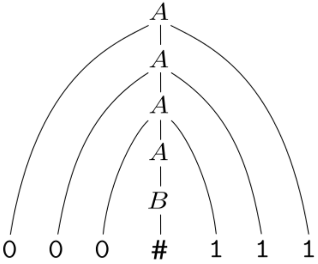


FIGURE 2.1
Parse tree for 000#111 in grammar G_1

Context-Free Grammars

Hopcroft, Motwani, and Ullman, 5.1.1, p-170

- Let us consider the language of (binary) palindromes,

$$
\begin{aligned}
P &\rightarrow \epsilon \\
P &\rightarrow 0 \\
P &\rightarrow 1 \\
\\
P &\rightarrow 0P0 \\
P &\rightarrow 1P1
\end{aligned}
$$

Context-Free Grammars

Hopcroft, Motwani, and Ullman, 5.1.1, p-170

- Let us consider the language of (binary) palindromes,

$$\left.\begin{array}{lcl} P & \rightarrow & \epsilon \\ P & \rightarrow & 0 \\ P & \rightarrow & 1 \end{array}\right\} \textit{Base cases}$$

$$\left.\begin{array}{lcl} P & \rightarrow & 0P0 \\ P & \rightarrow & 1P1 \end{array}\right\} \textit{Recursive cases}$$

Context-Free Grammars

Hopcroft, Motwani, and Ullman, 5.1.1, p-170

- Let us consider the language of (binary) palindromes,

$$\left.\begin{array}{lcl} P & \rightarrow & \epsilon \\ P & \rightarrow & 0 \\ P & \rightarrow & 1 \end{array}\right\} \textit{Base cases}$$

$$\left.\begin{array}{lcl} P & \rightarrow & 0P0 \\ P & \rightarrow & 1P1 \end{array}\right\} \textit{Recursive cases}$$

- Can be succinctly written as,

$$P \rightarrow \epsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1$$

Context-Free Grammars

Sipser, 2.1, p-102

$$
\begin{aligned}
\langle \text{SENTENCE} \rangle &\rightarrow \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \\
\langle \text{NOUN-PHRASE} \rangle &\rightarrow \langle \text{CMPLX-NOUN} \rangle \mid \langle \text{CMPLX-NOUN} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{VERB-PHRASE} \rangle &\rightarrow \langle \text{CMPLX-VERB} \rangle \mid \langle \text{CMPLX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{PREP-PHRASE} \rangle &\rightarrow \langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \\
\langle \text{CMPLX-NOUN} \rangle &\rightarrow \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \\
\langle \text{CMPLX-VERB} \rangle &\rightarrow \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle \\
\langle \text{ARTICLE} \rangle &\rightarrow \texttt{a} \mid \texttt{the} \\
\langle \text{NOUN} \rangle &\rightarrow \texttt{boy} \mid \texttt{girl} \mid \texttt{flower} \\
\langle \text{VERB} \rangle &\rightarrow \texttt{touches} \mid \texttt{likes} \mid \texttt{sees} \\
\langle \text{PREP} \rangle &\rightarrow \texttt{with}
\end{aligned}
$$

Context-Free Grammars

Sipser, 2.1, p-102

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

Formal Definition of a Context-Free Grammar

Sipser, Definition 2.2, p-104

Definition 2.2

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

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

Formal Definition of a Context-Free Grammar — *continued*

Sipser, Definition 2.2, p-104

- If u, v, and w are strings of variables and terminals.
- And $A \to w$ is a rule of the grammar.
- We say that uAv yields uwv, written $uAv \Rightarrow uwv$.

Formal Definition of a Context-Free Grammar — *continued*

Sipser, Definition 2.2, p-104

- Say that u derives v, written $u \stackrel{*}{\Rightarrow} v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$.
- The language of the grammar is $\left\{ w \in \Sigma^* \mid S \stackrel{*}{\Rightarrow} w \right\}$.

Example

Sipser, Example 2.3, p-105

- $G_3 = (\{S\}, \{a, b\}, R, S)$.
- The set of rules, R, is $S \rightarrow aSb \mid SS \mid \epsilon$.

Example

Sipser, Example 2.3, p-105

- $G_3 = (\{S\}, \{a,b\}, R, S)$.
- The set of rules, R, is $S \to aSb \mid SS \mid \epsilon$.
- This grammar generates strings such as $abab$, $aaabbb$, and $aababb$.
- You can see more easily what this language is if you think of a as a left parenthesis "(" and b as a right parenthesis ")".
- Viewed in this way, $L(G_3)$ is the language of all strings of properly nested parentheses.
- Observe that the right-hand side of a rule may be the empty string ϵ.

Example

Sipser, Example 2.4, p-105

- $G_4 = (V, \Sigma, R, \texttt{<EXPR>})$.
- V is $\{\texttt{<EXPR>}, \texttt{<TERM>}, \texttt{<FACTOR>}\}$,
- and Σ is $\{a, +, \times, (,)\}$.
- The rules are,

$$
\begin{aligned}
\texttt{<EXPR>} &\rightarrow \texttt{<EXPR>} + \texttt{<TERM>} \mid \texttt{<TERM>} \\
\texttt{<TERM>} &\rightarrow \texttt{<TERM>} \times \texttt{<FACTOR>} \mid \texttt{<FACTOR>} \\
\texttt{<FACTOR>} &\rightarrow (\texttt{<EXPR>}) \mid a
\end{aligned}
$$

Formal Definition of a Context-Free Grammar

Sipser, Figure 2.5, p-105

<EXPR> → <EXPR> + <TERM> | <TERM>

<TERM> → <TERM> × <FACTOR> | <FACTOR>

<FACTOR> → (<EXPR>) | a

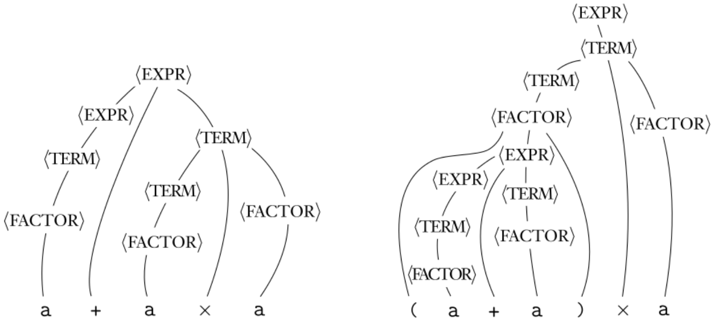


FIGURE 2.5
Parse trees for the strings a+a×a and (a+a)×a

Designing CFGs

Adapted from `https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf`

- Like designing DFAs, NFAs, and regular expressions, designing CFGs is a craft.

Designing CFGs

Adapted from https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf

- When thinking about CFGs:
  - Think recursively:
    - Build up bigger structures from smaller ones.
  - Have a construction plan:
    - Know in what order you will build up the string.
  - Store information in nonterminals:
    - Have each nonterminal correspond to some useful piece of information.

Designing Context-Free Grammars

Sipser, 2.1, p-106

- $L = \{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}$

- To get a grammar for the language $L = \{0^n1^n \mid n \geq 0\} \cup \{1^n0^n \mid n \geq 0\}$
- First construct the grammar $S_1 \to 0S_11 \mid \epsilon$ for the language $\{0^n1^n \mid n \geq 0\}$.
- Then construct the grammar $S_2 \to 1S_20 \mid \epsilon$ for the language $\{1^n0^n \mid n \geq 0\}$.
- Then add the rule $S \to S_1 \mid S_2$ to give the grammar

$$
\begin{aligned}
S &\to S_1 \mid S_2 \\
S_1 &\to 0S_11 \mid \epsilon \\
S_2 &\to 1S_20 \mid \epsilon.
\end{aligned}
$$

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{0^n 1^{2n} \mid n \geq 0\}$

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{0^n 1^{2n} \mid n \geq 0\}$

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{0^n 1^{2n} \mid n \geq 0\}$
- The grammar forces every 0 to match to 11.

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{0^n 1^{2n} \mid n \geq 0\}$
- The grammar forces every 0 to match to 11.
- The context-free grammar for L is,

$$S \rightarrow 0S11 \mid \epsilon$$

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{0^n 1^m \mid m, n \geq 0,\ 2n \leq m \leq 3n\}$

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{0^n 1^m \mid m, n \geq 0,\ 2n \leq m \leq 3n\}$

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{0^n 1^m \mid m, n \geq 0,\ 2n \leq m \leq 3n\}$
- The grammar forces every 0 to match to 11 or 111.

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{0^n 1^m \mid m, n \geq 0,\ 2n \leq m \leq 3n\}$
- The grammar forces every 0 to match to 11 or 111.
- The context-free grammar for L is

$$S \rightarrow 0S11 \mid 0S111 \mid \epsilon$$

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{0^n 1^m \mid m, n \geq 0,\ n \neq m\}$

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{0^n 1^m \mid m, n \geq 0,\ n \neq m\}$

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{0^n 1^m \mid m, n \geq 0,\ n \neq m\}$
- Let $L_1 = \{0^n 1^m \mid m, n \geq 0,\ n > m\}$
- Let $L_2 = \{0^n 1^m \mid m, n \geq 0,\ n < m\}$

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{0^n 1^m \mid m, n \geq 0,\ n \neq m\}$
- Let $L_1 = \{0^n 1^m \mid m, n \geq 0,\ n > m\}$
- Let $L_2 = \{0^n 1^m \mid m, n \geq 0,\ n < m\}$
- Then, if S_1 generates L_1, and S_2 generates L_2, our grammar will be,

$$S \rightarrow S_1 \mid S_2$$

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- L_1 is just the language of strings 0^n1^n with one or more extra 0's in front.
- So,

$$S_1 \to 0S_1 \mid 0E$$
$$E \to 0E1 \mid \epsilon$$

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- L_1 is just the language of strings 0^n1^n with one or more extra 0's in front.
- So,

$$S_1 \to 0S_1 \mid 0E$$
$$E \to 0E1 \mid \epsilon$$

- L_2 is just the language of strings 0^n1^n with one or more extra 1's in the end.
- So,

$$S_2 \to S_21 \mid E1$$
$$E \to 0E1 \mid \epsilon$$

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

$$L = \{0^n 1^m \mid m, n \geq 0,\ n \neq m\}$$

- Finally, our desired grammar is,

$$\begin{aligned} S &\rightarrow S_1 \mid S_2 \\ S_1 &\rightarrow 0S_1 \mid 0E \\ S_2 &\rightarrow S_2 1 \mid E1 \\ E &\rightarrow 0E1 \mid \epsilon \end{aligned}$$

Designing Context-Free Grammars

http://www.andrew.cmu.edu/user/ko/pdfs/lecture-7.pdf

- $L = \{w \mid w \in \{a, b\}^*, n_a(w) = n_b(w)\}$.

Designing Context-Free Grammars

http://www.andrew.cmu.edu/user/ko/pdfs/lecture-7.pdf

- $L = \{w \mid w \in \{a,b\}^*, n_a(w) = n_b(w)\}$

- $L = \{w \mid w \in \{a, b\}^*, n_a(w) = n_b(w)\}$
- The grammar generates the basis strings of ϵ, ab and ba.

- If w is a string in this grammar, awb will belong to this grammar.
- awb will be generated from by using the rule $S \to aSb$.

- $L = \{w \mid w \in \{a, b\}^*, n_a(w) = n_b(w)\}$
- The grammar generates the basis strings of ϵ, ab and ba.

- If w is a string in this grammar, bwa will belong to this grammar.
- bwa will be generated from by using the rule $S \to bSa$.

- $L = \{w \mid w \in \{a, b\}^*, n_a(w) = n_b(w)\}$
- The grammar generates the basis strings of ϵ, ab and ba.

- If w is a string in this grammar, ww will belong to this grammar.
- w will be generated from by using the rule $S \to SS$.

Designing Context-Free Grammars

http://www.andrew.cmu.edu/user/ko/pdfs/lecture-7.pdf

- $L = \{w \mid w \in \{a, b\}^*, n_a(w) = n_b(w)\}$
- $S \rightarrow aSb \mid bSa \mid SS \mid \epsilon$

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{w \mid w \in \{0,1\}^*$ and of even length$\}$.

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{w \mid w \in \{0,1\}^* \text{ and of even length}\}$
- The grammar generates the basis strings of ϵ, 00, 01, 10, and 11.

- If w is a string in this grammar, $0w0$ will belong to this grammar.
- $0w0$ will be generated by using the rule $S \rightarrow 0S0$.

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{w \mid w \in \{0,1\}^*$ and of even length$\}$
- The grammar generates the basis strings of ϵ, 00, 01, 10, and 11.

- If w is a string in this grammar, $0w1$ will belong to this grammar.
- $0w1$ will be generated by using the rule $S \to 0S1$.

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{w \mid w \in \{0, 1\}^*$ and of even length$\}$
- The grammar generates the basis strings of ϵ, 00, 01, 10, and 11.

- If w is a string in this grammar, $1w0$ will belong to this grammar.
- $1w0$ will be generated by using the rule $S \rightarrow 1S0$.

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{w \mid w \in \{0,1\}^*$ and of even length$\}$
- The grammar generates the basis strings of ϵ, 00, 01, 10, and 11.

- If w is a string in this grammar, $1w1$ will belong to this grammar.
- $1w1$ will be generated by using the rule $S \to 1S1$.

Designing Context-Free Grammars

http://www.eecs.yorku.ca/course_archive/2006-07/F/2001/handouts/lect11.pdf

- $L = \{w \mid w \in \{0,1\}^* \text{ and of even length}\}$
- $S \rightarrow \epsilon \mid 0S0 \mid 0S1 \mid 1S0 \mid 1S1$.

Designing Context-Free Grammars

http://www.cs.toronto.edu/~azadeh/page11/page12/material/hw5-sol.pdf

- $L = \{a^n b^m c^k \mid n, m, k \geq 0 \text{ and } n = m + k\}$

Designing Context-Free Grammars

http://www.cs.toronto.edu/~azadeh/page11/page12/material/hw5-sol.pdf

- $L = \{a^n b^m c^k \mid n, m, k \geq 0 \text{ and } n = m + k\}$
- Every b should match an a.
- Every c should match an a.

Designing Context-Free Grammars

http://www.cs.toronto.edu/~azadeh/page11/page12/material/hw5-sol.pdf

- $L = \{a^n b^m c^k \mid n, m, k \geq 0 \text{ and } n = m + k\}$
- Every b should match an a.
- Every c should match an a.
- Thinking recursively, we will want to build the $a^s \ldots c^s$ part first.
- And then, build the $a^r b^r$ *inside* the previously built $a^s \ldots c^s$, like, $a^s a^r b^r c^s$.

Designing Context-Free Grammars

http://www.cs.toronto.edu/~azadeh/page11/page12/material/hw5-sol.pdf

- $L = \left\{a^n b^m c^k \mid n, m, k \geq 0 \text{ and } n = m + k\right\}$
- Every b should match an a.
- Every c should match an a.
- Thinking recursively, we will want to build the $a^s \ldots c^s$ part first.
- And then, build the $a^r b^r$ *inside* the previously built $a^s \ldots c^s$, like, $a^s a^r b^r c^s$.
- Can we go in the other direction, like, build the $a^r b^r$ and then build the $a^s \ldots c^s$?

- $L = \{a^n b^m c^k \mid n, m, k \geq 0$ and $n = m + k\}$
-

$$S \rightarrow aSc \mid B$$
$$B \rightarrow aBb \mid \epsilon$$

Example

Adapted from `https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf`

- Let $\Sigma = \{\{, \}\}$ and let $L = \{w \in \Sigma^* \mid w \text{ is a string of balanced braces}\}$.
- Some sample strings in L:

{{{}}}

{{}}{}

{{}{}}{{}{}}

{{{{{}}}{{}}}}

ε

{}{}

Example — *continued*

Adapted from `https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf`

- $\Sigma = \{\{, \}\}$, $L = \{w \in \Sigma^* \mid w$ is a string of balanced braces$\}$.
- Let's think about this recursively.
- Base case: the empty string is a string of balanced braces.
- Recursive step: Look at the closing brace that matches the first open brace.

{{{}{{}}}{{}}}{{}}{{{}}}

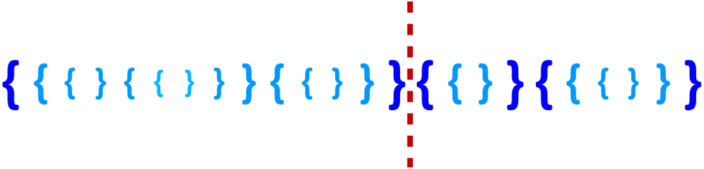
- $\Sigma = \{\{, \}\}$, $L = \{w \in \Sigma^* \mid w$ is a string of balanced braces$\}$.
- Let's think about this recursively.
- Base case: the empty string is a string of balanced braces.
- Recursive step: Look at the closing brace that matches the first open brace.

$$\{\{\{\}\{\{\}\}\}\{\{\}\}\}\{\{\}\}\{\{\{\}\}\}$$

Example — *continued*

Adapted from `https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf`

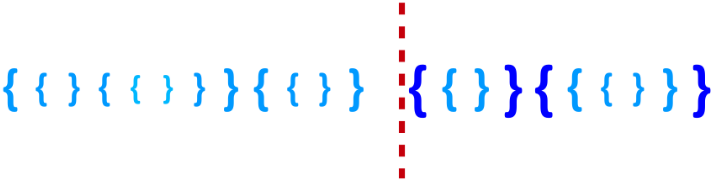
- $\Sigma = \{\{, \}\}$, $L = \{w \in \Sigma^* \mid w$ is a string of balanced braces$\}$.
- Let's think about this recursively.
- Base case: the empty string is a string of balanced braces.
- Recursive step: Look at the closing brace that matches the first open brace.



Example — *continued*

Adapted from `https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf`

- $\Sigma = \{\{, \}\}$, $L = \{w \in \Sigma^* \mid w$ is a string of balanced braces$\}$.
- Let's think about this recursively.
- Base case: the empty string is a string of balanced braces.
- Recursive step: Look at the closing brace that matches the first open brace.



Example — *continued*

Adapted from https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf

- $\Sigma = \{\{, \}\}$, $L = \{w \in \Sigma^* \mid w$ is a string of balanced braces$\}$.
- Let's think about this recursively.
- Base case: the empty string is a string of balanced braces.
- Recursive step: Look at the closing brace that matches the first open brace.

Example — *continued*

Adapted from https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf

- $\Sigma = \{\{,\}\}$, $L = \{w \in \Sigma^* \mid w$ is a string of balanced braces$\}$.
- Let's think about this recursively.
- Base case: the empty string is a string of balanced braces.
- Recursive step: Look at the closing brace that matches the first open brace.
-

$$S \rightarrow \{S\}\,S \mid \epsilon$$

Designing CFGs — Storing Information in Nonterminals

Adapted from `https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf`

- Different non-terminals should represent different states or different types of strings.
- For example, different phases of the build, or different possible structures for the string.
- Think like the same ideas from DFA/NFA design where states in your automata represent pieces of information.

Example

Adapted from `https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf`

- Let $\Sigma = \{a, b\}$ and let

  $L = \{w \in \Sigma^* \mid$ Length of w is a multiple of 3 and all the characters in the first third of w are the same.$\}$

- Examples:

$\epsilon \in L$	$a \notin L$
$a \mid bb \in L$	$b \notin L$
$b \mid ab \in L$	$ab \mid abab \notin L$
$aa \mid baba \in L$	$aab \mid aaaaaa \notin L$
$bb \mid bbbb \in L$	$bbbb \notin L$

Example — *continued*

Adapted from `https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf`

aaa	bab
abb	bbb
$aaabab$	$bbabbb$
$aababa$	$bbbaaaaaa$
$aaaaaaaaa$	$bbbbbabaa$

Observation 1: Strings in this language are either the first third is as or the first third is bs.

Example — *continued*

Adapted from `https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf`

$\boldsymbol{a}aa$	$\boldsymbol{b}ab$
$\boldsymbol{a}bb$	$\boldsymbol{b}bb$
$\boldsymbol{aa}abab$	$\boldsymbol{bb}abbb$
$\boldsymbol{aa}baba$	$\boldsymbol{bbb}aaaaaa$
$\boldsymbol{aaa}aaaaaa$	$\boldsymbol{bbb}bbabaa$

Observation 2: Amongst these strings, for every $\boldsymbol{a}$ we have in the first third, we need two other characters in the last two thirds.

- This pattern of "for every x we see here, we need a y somewhere else in the string" is very common in CFGs!

Example — *continued*

Adapted from `https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf`

$\mathbf{a}aa$	$\mathbf{b}ab$
$\mathbf{a}bb$	$\mathbf{b}bb$
$\mathbf{aa}abab$	$\mathbf{bb}abbb$
$\mathbf{aa}baba$	$\mathbf{bbb}aaaaaa$
$\mathbf{aaa}aaaaaa$	$\mathbf{bbb}bbabaa$

$$A \to \mathbf{a}AXX \mid \epsilon$$
$$X \to a \mid b$$

- Here the nonterminal A represents "a string where the first third is $\mathbf{a}$'s".
- The nonterminal X represents "any character".

Example — *continued*

Adapted from `https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf`

$\boldsymbol{a}aa$	$\boldsymbol{b}ab$
$\boldsymbol{a}bb$	$\boldsymbol{b}bb$
$\boldsymbol{aa}abab$	$\boldsymbol{bb}abbb$
$\boldsymbol{aa}baba$	$\boldsymbol{bbb}aaaaaa$
$\boldsymbol{aaa}aaaaaa$	$\boldsymbol{bbb}bbabaa$

$$B \to \boldsymbol{b}BXX \mid \epsilon$$
$$X \to a \mid b$$

- Here the nonterminal B represents "a string where the first third is $\boldsymbol{b}$'s".
- The nonterminal X represents "any character".

Example — *continued*

Adapted from `https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf`

$\mathbf{a}aa$	$\mathbf{b}ab$
$\mathbf{a}bb$	$\mathbf{b}bb$
$\mathbf{aa}abab$	$\mathbf{bb}abbb$
$\mathbf{aa}baba$	$\mathbf{bbb}aaaaaa$
$\mathbf{aaa}aaaaaa$	$\mathbf{bbb}bbabaa$

- Tying everything together:

$$
\begin{aligned}
S &\to A \mid B \\
A &\to \mathbf{a}AXX \mid \epsilon \\
B &\to \mathbf{b}BXX \mid \epsilon \\
X &\to a \mid b
\end{aligned}
$$

Example — *continued*

Adapted from `https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf`

$$S \rightarrow A \mid B \mid \epsilon$$

$$A \rightarrow \boldsymbol{a}AXX$$

$$B \rightarrow \boldsymbol{b}BXX$$

$$X \rightarrow a \mid b$$

- Overall strings in this language either follow the pattern of A or B.
- A represents “strings where the first third is $\boldsymbol{a}$’s”.
- B represents “strings where the first third is $\boldsymbol{b}$’s”.

Summary of CFG Design Tips

Adapted from `https://web.stanford.edu/class/archive/cs/cs103/cs103.1208/lectures/16-CFGs/CFGs.pdf`

- Look for recursive structures where they exist.
- They can help guide you toward a solution.
- Keep the build order in mind — often, you'll build two totally different parts of the string concurrently.
- Usually, those parts are built in opposite directions.
- One's built right-to-left, the other left-to-right.
- Use different nonterminals to represent different structures.

- Second, constructing a CFG for a language that happens to be regular is easy if you can first construct a DFA for that language.

Designing Context-Free Grammars — *continued*

Sipser, 2.1, p-107

- You can convert any DFA into an equivalent CFG as follows.
- Make a variable R_i for each state q_i of the DFA.
- Add the rule $R_i \rightarrow aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA.
- Add the rule $R_i \rightarrow \epsilon$ if q_i is an accept state of the DFA.
- Make R_0 the start variable of the grammar, where q_0 is the start state of the machine.

Designing Context-Free Grammars — *continued*

Sipser, Figure 1.22, p-44

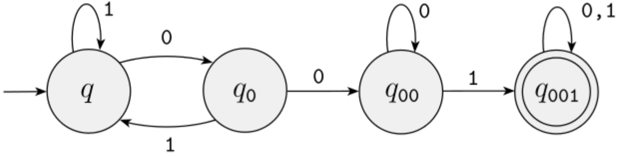
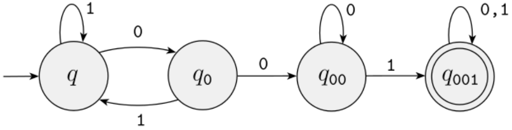


FIGURE 1.22
Accepts strings containing 001

Designing Context-Free Grammars — *continued*

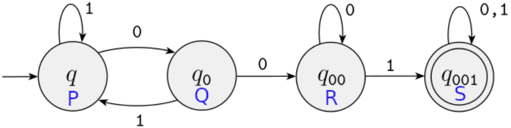
Sipser, 2.1, p-107



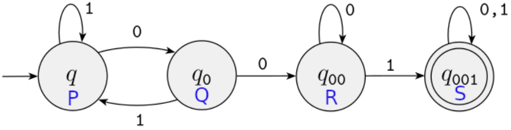
- Make a variable R_i for each state q_i of the DFA.
- Add the rule $R_i \rightarrow aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA.
- Add the rule $R_i \rightarrow \epsilon$ if q_i is an accept state of the DFA.
- Make R_0 the start variable of the grammar, where q_0 is the start state of the machine.

Designing Context-Free Grammars — *continued*

Sipser, 2.1, p-107



- Make a variable R_i for each state q_i of the DFA.
- Add the rule $R_i \rightarrow aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA.
- Add the rule $R_i \rightarrow \epsilon$ if q_i is an accept state of the DFA.
- Make R_0 the start variable of the grammar, where q_0 is the start state of the machine.



- Make a variable R_i for each state q_i of the DFA.
- Add the rule $R_i \to aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA.
- Add the rule $R_i \to \epsilon$ if q_i is an accept state of the DFA.
- Make R_0 the start variable of the grammar, where q_0 is the start state of the machine.

- So, the resulting grammar is,

$$P \to 0Q \qquad R \to 0R$$
$$P \to 1P \qquad R \to 1S \qquad S \to \epsilon$$
$$Q \to 0R \qquad S \to 0S$$
$$Q \to 1P \qquad S \to 1S$$

Designing Context-Free Grammars — *continued*

Sipser, 2.1, p-107

- Third, certain context-free languages contain strings with two substrings.
- These are “linked” in the sense that a machine for such a language would need to remember an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other substring.

- This situation occurs in the language $\{0^n 1^n \mid n \geq 0\}$ because a machine would need to remember the number of 0s in order to verify that it equals the number of 1s.
- You can construct a CFG to handle this situation by using a rule of the form $R \rightarrow uRv$.
- Which generates strings wherein the portion containing the u's corresponds to the portion containing the v's.

- Finally, in more complex languages, the strings may contain certain structures that appear recursively as part of other (or the same) structures.
- That situation occurs in the grammar that generates arithmetic expressions.

$$E \to E + T \mid T$$
$$T \to T \times F \mid F$$
$$F \to (E) \mid a$$

Designing Context-Free Grammars — *continued*

Sipser, 2.1, p-107

$$E \to E + T \mid T$$
$$T \to T \times F \mid F$$
$$F \to (E) \mid a$$

- Any time the symbol a appears, an entire parenthesized expression might appear recursively instead.
- To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure may recursively appear.

Leftmost and Rightmost Derivations

Hopcroft, Motwani, and Ullman, 5.1.4, p-175

- We want to restrict the number of choices we have in deriving a string.
- It is often useful to require that at each step we replace the leftmost variable by one of its production bodies.
- Such a derivation is called a *leftmost derivation*.
- We indicate that a derivation is leftmost by using the relations $\underset{lm}{\Longrightarrow}$ and $\overset{*}{\underset{lm}{\Longrightarrow}}$, for one or many steps, respectively.
- If the grammar G that is being used is not obvious, we can place the name G below the arrow in either of these symbols.

Leftmost and Rightmost Derivations — *continued*

Hopcroft, Motwani, and Ullman, 5.1.4, p-175

- Similarly, it is possible to require that at each step the rightmost variable is replaced by one of its bodies.
- If so, we call the derivation *rightmost* and use the symbols $\underset{rm}{\Longrightarrow}$ and $\overset{*}{\underset{rm}{\Longrightarrow}}$ to indicate one or many rightmost derivation steps, respectively.
- Again, the name of the grammar may appear below these symbols if it is not clear which grammar is being used.

Example

Hopcroft, Motwani, and Ullman, Example 5.6, p-176

- A leftmost derivation.

$$
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow I * E \\
&\Rightarrow a * E \\
&\Rightarrow a * (E) \\
&\Rightarrow a * (E + E) \\
&\Rightarrow a * (I + E) \\
&\Rightarrow a * (a + E) \\
&\Rightarrow a * (a + I) \\
&\Rightarrow a * (a + I0) \\
&\Rightarrow a * (a + I00) \\
&\Rightarrow a * (a + b00)
\end{aligned}
$$

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 5.6, p-176

- Thus, we can describe the same derivation by:

$$
\begin{aligned}
E \underset{lm}{\Longrightarrow}\ & E * E \underset{lm}{\Longrightarrow} I * E \underset{lm}{\Longrightarrow} a * E \underset{lm}{\Longrightarrow} \\
& a * (E) \underset{lm}{\Longrightarrow} a * (E + E) \underset{lm}{\Longrightarrow} a * (I + E) \underset{lm}{\Longrightarrow} \\
& a * (a + E) \underset{lm}{\Longrightarrow} a * (a + I) \underset{lm}{\Longrightarrow} a * (a + I0) \underset{lm}{\Longrightarrow} \\
& a * (a + I00) \underset{lm}{\Longrightarrow} a * (a + b00)
\end{aligned}
$$

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 5.6, p-176

- Thus, we can describe the same derivation by:

$$\begin{aligned} E \underset{lm}{\Longrightarrow} & \; E * E \underset{lm}{\Longrightarrow} I * E \underset{lm}{\Longrightarrow} a * E \underset{lm}{\Longrightarrow} \\ & a * (E) \underset{lm}{\Longrightarrow} a * (E + E) \underset{lm}{\Longrightarrow} a * (I + E) \underset{lm}{\Longrightarrow} \\ & a * (a + E) \underset{lm}{\Longrightarrow} a * (a + I) \underset{lm}{\Longrightarrow} a * (a + I0) \underset{lm}{\Longrightarrow} \\ & a * (a + I00) \underset{lm}{\Longrightarrow} a * (a + b00) \end{aligned}$$

- We can also summarize the leftmost derivation by saying $E \overset{*}{\underset{lm}{\Longrightarrow}} a * (a + b00)$.
- Or express several steps of the derivation by expressions such as $E * E \overset{*}{\underset{lm}{\Longrightarrow}} a * (E)$.

- There is a rightmost derivation that uses the same replacements for each variable.
- Although it makes the replacements in different order.

- This rightmost derivation is:

$$
\begin{aligned}
E & \underset{rm}{\Longrightarrow} E * E \\
& \underset{rm}{\Longrightarrow} E * (E) \\
& \underset{rm}{\Longrightarrow} E * (E + E) \\
& \underset{rm}{\Longrightarrow} E * (E + I) \\
& \underset{rm}{\Longrightarrow} E * (E + I0) \\
& \underset{rm}{\Longrightarrow} E * (E + I00) \\
& \underset{rm}{\Longrightarrow} E * (E + b00) \\
& \underset{rm}{\Longrightarrow} E * (I + b00) \\
& \underset{rm}{\Longrightarrow} E * (a + b00) \\
& \underset{rm}{\Longrightarrow} I * (a + b00) \\
& \underset{rm}{\Longrightarrow} a * (a + b00)
\end{aligned}
$$

$$
\begin{aligned}
E &\underset{rm}{\Longrightarrow} E * E \\
&\underset{rm}{\Longrightarrow} E * (E) \\
&\underset{rm}{\Longrightarrow} E * (E + E) \\
&\underset{rm}{\Longrightarrow} E * (E + I) \\
&\underset{rm}{\Longrightarrow} E * (E + I0) \\
&\underset{rm}{\Longrightarrow} E * (E + I00) \\
&\underset{rm}{\Longrightarrow} E * (E + b00) \\
&\underset{rm}{\Longrightarrow} E * (I + b00) \\
&\underset{rm}{\Longrightarrow} E * (a + b00) \\
&\underset{rm}{\Longrightarrow} I * (a + b00) \\
&\underset{rm}{\Longrightarrow} a * (a + b00)
\end{aligned}
$$

- This derivation allows us to conclude $E \overset{*}{\underset{rm}{\Longrightarrow}} a * (a + b00)$.

- Any derivation has an equivalent leftmost and an equivalent rightmost derivation.
- That is, if w is a terminal string, and A a variable, then
  - $A \overset{*}{\Rightarrow} w$ if and only if $A \underset{lm}{\overset{*}{\Longrightarrow}} w$, and
  - $A \overset{*}{\Rightarrow} w$ if and only if $A \underset{rm}{\overset{*}{\Longrightarrow}} w$.

The Language of a Grammar

Hopcroft, Motwani, and Ullman, 5.1.5, p-177

- If $G(V,T,P,S)$ is a CFG, the *language* of G, denoted $L(G)$, is the set of terminal strings that have derivations from the start symbol.

- If $G(V, T, P, S)$ is a CFG, the *language* of G, denoted $L(G)$, is the set of terminal strings that have derivations from the start symbol.
- That is,

$$L(G) = \left\{ w \text{ in } T^* \;\middle|\; S \overset{*}{\underset{G}{\Longrightarrow}} w \right\}.$$

- If a language L is the language of some context-free grammar, then L is said to be a *context-free language*, or CFL.

Sentential Forms

Hopcroft, Motwani, and Ullman, 5.1.6, p-178

- Derivations from the start symbol produce strings that have a special role.
- We call these "sentential forms."

Sentential Forms

Hopcroft, Motwani, and Ullman, 5.1.6, p-178

- Derivations from the start symbol produce strings that have a special role.
- We call these "sentential forms."
- That is, if $G(V, T, P, S)$ is a CFG, then any string α in $(V \cup T)^*$ such that $S \stackrel{*}{\Rightarrow} \alpha$ is a *sentential form*.

Sentential Forms — *continued*

Hopcroft, Motwani, and Ullman, 5.1.6, p-178

- If $S \underset{lm}{\overset{*}{\Longrightarrow}} \alpha$, then α is a left-sentential form.

Sentential Forms — *continued*

Hopcroft, Motwani, and Ullman, 5.1.6, p-178

- If $S \overset{*}{\underset{lm}{\Longrightarrow}} \alpha$, then α is a left-sentential form.
- And if $S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha$, then α is a right-sentential form.

- If $S \overset{*}{\underset{lm}{\Longrightarrow}} \alpha$, then α is a left-sentential form.
- And if $S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha$, then α is a right-sentential form.
- Note that the language $L(G)$ is those sentential forms that are in T^*; *i.e.*, they consist solely of terminals.

Example

Hopcroft, Motwani, and Ullman, 5.1.6, p-178

- Consider the grammar for expressions,

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

Example — *continued*

Hopcroft, Motwani, and Ullman, 5.1.6, p-178

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

- For example, $E * (I + E)$ is a sentential form, since there is a derivation,

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E).$$

Example — *continued*

Hopcroft, Motwani, and Ullman, 5.1.6, p-178

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

- For example, $E * (I + E)$ is a sentential form, since there is a derivation,

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E).$$

- However this derivation is neither leftmost nor rightmost, since at the last step, the middle E is replaced.

Example — *continued*

Hopcroft, Motwani, and Ullman, 5.1.6, p-178

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

- As an example of a left-sentential form, consider $\alpha * E$, with the leftmost derivation,

$$E \underset{lm}{\Longrightarrow} E * E \underset{lm}{\Longrightarrow} I * E \underset{lm}{\Longrightarrow} \alpha * E.$$

Example — *continued*

Hopcroft, Motwani, and Ullman, 5.1.6, p-178

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

- Additionally, the derivation,

$$E \underset{rm}{\Longrightarrow} E * E \underset{rm}{\Longrightarrow} E * (E) \underset{rm}{\Longrightarrow} E * (E + E)$$

shows that $E * (E + E)$ is a right-sentential form.

Parse Trees

Hopcroft, Motwani, and Ullman, 5.2, p-181

- There is a tree representation for derivations that has proved extremely useful.
- This tree shows us clearly how the symbols of a terminal string are grouped into substrings.
- Each of the terminals belongs to the language of one of the variables of the grammar.

Parse Trees — *continued*

Hopcroft, Motwani, and Ullman, 5.2, p-181

- The tree, known as a “parse tree” when used in a compiler, is the data structure of choice to represent the source program.
- In a compiler, the tree structure of the source program facilitates the translation of the source program into executable code by allowing natural, recursive functions to perform this translation process.

- Certain grammars allow a terminal string to have more than one parse tree.
- That situation makes the grammar unsuitable for a programming language.
- The compiler could not tell the structure of certain source programs.
- And therefore could not with certainty deduce what the proper executable code for the program was.

Constructing Parse Trees

Hopcroft, Motwani, and Ullman, 5.2.1, p-181

- Let us fix on a grammar $G(V, T, P, S)$.
- The parse trees for G are trees with the following conditions:

1. Each interior node is labeled by a variable in V.

Constructing Parse Trees

Hopcroft, Motwani, and Ullman, 5.2.1, p-181

- Let us fix on a grammar $G(V, T, P, S)$.
- The parse trees for G are trees with the following conditions:

2. Each leaf is labeled by either a variable, a terminal, or ϵ.
   - However, if the leaf is labeled ϵ, then it must be the only child of its parent.

Constructing Parse Trees

Hopcroft, Motwani, and Ullman, 5.2.1, p-181

- Let us fix on a grammar $G(V, T, P, S)$.
- The parse trees for G are trees with the following conditions:

3. If an interior node is labeled A, and its children are labeled

$$X_1, X_2, \ldots, X_k$$

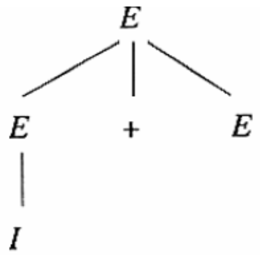
respectively, from the left, then $A \rightarrow X_1 X_2 \ldots X_k$ is a production in P.

  - Note that the only time one of the X's can be ϵ is if that is the label of the only child, and $A \rightarrow \epsilon$ is a production of G.

Example

Hopcroft, Motwani, and Ullman, Example 5.9, p-182

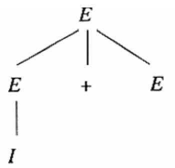
- Figure shows a parse tree that uses the expression grammar.



A parse tree showing the derivation of $I + E$ from E

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 5.9, p-182



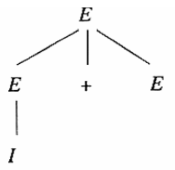
A parse tree showing the derivation of $I + E$ from E

1. $E \rightarrow I$
2. $E \rightarrow E+E$
3. $E \rightarrow E*E$
4. $E \rightarrow (E)$
5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

- The production used at the root is $E \rightarrow E + E$.

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 5.9, p-182



A parse tree showing the derivation of $I + E$ from E

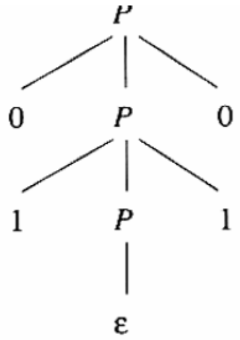
1. $E \rightarrow I$
2. $E \rightarrow E+E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

- The production used at the root is $E \rightarrow E + E$.
- At the leftmost child of the root, the production $E \rightarrow I$ is used.

Example

Hopcroft, Motwani, and Ullman, Example 5.10, p-182

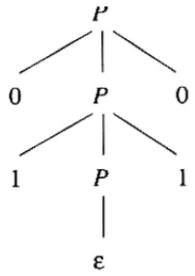
- Figure shows a parse tree for the palindromic grammar.



A parse tree showing the derivation $P \overset{*}{\Rightarrow} 0110$

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 5.10, p-182



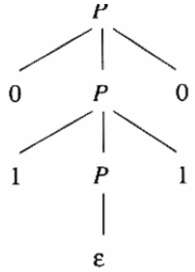
A parse tree showing the derivation $P \overset{*}{\Rightarrow} 0110$

$$
\begin{aligned}
P &\rightarrow \epsilon \\
P &\rightarrow 0 \\
P &\rightarrow 1 \\
\\
P &\rightarrow 0P0 \\
P &\rightarrow 1P1
\end{aligned}
$$

- The production used at the root is $P \rightarrow 0P0$.

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 5.10, p-182



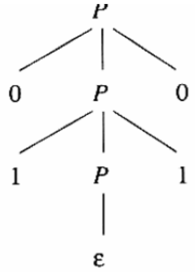
A parse tree showing the derivation $P \overset{*}{\Rightarrow} 0110$

$$
\begin{aligned}
P &\rightarrow \epsilon \\
P &\rightarrow 0 \\
P &\rightarrow 1 \\
\\
P &\rightarrow 0P0 \\
P &\rightarrow 1P1
\end{aligned}
$$

- At the middle child of the root it is $P \rightarrow 1P1$.

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 5.10, p-182



A parse tree showing the derivation $P \overset{*}{\Rightarrow} 0110$

$$\begin{aligned} P &\rightarrow \epsilon \\ P &\rightarrow 0 \\ P &\rightarrow 1 \\ \\ P &\rightarrow 0P0 \\ P &\rightarrow 1P1 \end{aligned}$$

- Note that at the bottom is a use of the production $P \rightarrow \epsilon$.
- That use, labeled ϵ, is the only time that a node labeled ϵ can appear in a parse tree.

The Yield of a Parse Tree

Hopcroft, Motwani, and Ullman, 5.2.2, p-183

- If we look at the leaves of any parse tree and concatenate them from the left, we get a string.
- This is called the yield of the tree.
- This is always a string that is derived from the root variable.

The Yield of a Parse Tree — *continued*

Hopcroft, Motwani, and Ullman, 5.2.2, p-183

- Of special importance are those parse trees such that:
  1. The yield is a terminal string.
     - That is, all leaves are labeled either with a terminal or with ϵ.
  2. The root is labeled by the start symbol.
- These are the parse trees whose yields are strings in the language of the underlying grammar.

The Yield of a Parse Tree — *continued*

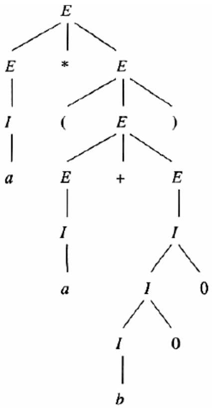
Hopcroft, Motwani, and Ullman, 5.2.2, p-183

- Of special importance are those parse trees such that:
  1. The yield is a terminal string.
     - That is, all leaves are labeled either with a terminal or with ϵ.
  2. The root is labeled by the start symbol.
- These are the parse trees whose yields are strings in the language of the underlying grammar.

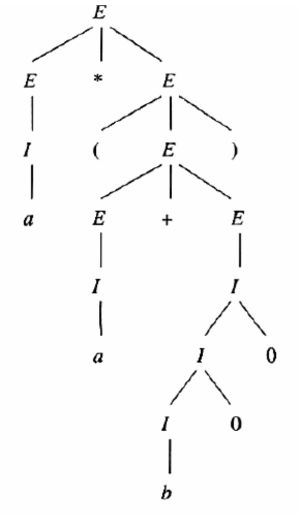
Example

Hopcroft, Motwani, and Ullman, Example 5.11, p-183

- The figure is an example of a tree with a terminal string as yield and the start symbol at the root.

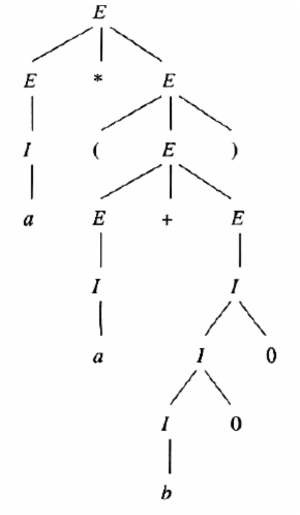


Parse tree showing $a * (a + b00)$ is in the language of our expression grammar



1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

- It is based on the grammar for expressions.
- This tree's yield is the string $a * (a + b00)$.



1. $E \to I$
2. $E \to E+E$
3. $E \to E*E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

- It is based on the grammar for expressions.
- This tree's yield is the string $a*(a+b00)$.
- This parse tree is a representation of derivation.

Ambiguous Grammars

Hopcroft, Motwani, and Ullman, 5.4.1, p-205

- Expression grammar of figure lets us generate expressions with any sequence of $*$ and $+$ operators.
- The productions $E \to E + E \mid E * E$ allow us to generate these expressions in any order we choose.

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

Example

Hopcroft, Motwani, and Ullman, Example 5.25, p-206

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

- Consider the sentential form $E + E * E$.
- It has two derivations from E:
  1. $E \Rightarrow E + E \Rightarrow E + E * E$
  2. $E \Rightarrow E * E \Rightarrow E + E * E$
- In derivation (1), the second E is replaced by $E * E$.
- While in derivation (2), the first E is replaced by $E + E$.

Example

Hopcroft, Motwani, and Ullman, Example 5.25, p-206

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

- Consider the sentential form $E + E * E$.
- It has two derivations from E:
  1. $E \Rightarrow E + E \Rightarrow E + E * E$
  2. $E \Rightarrow E * E \Rightarrow E + E * E$
- In derivation (1), the second E is replaced by $E * E$.
- While in derivation (2), the first E is replaced by $E + E$.

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 5.25, p-206

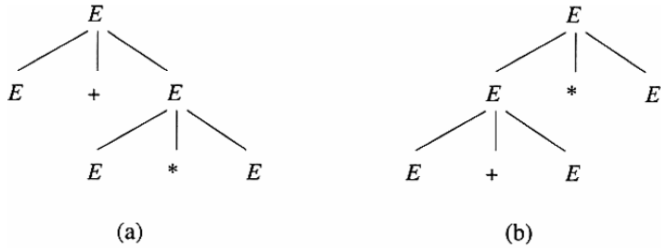
$$3 + 4 * 5 = 23?$$

$$3 + 4 * 5 = 35?$$

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 5.25, p-206

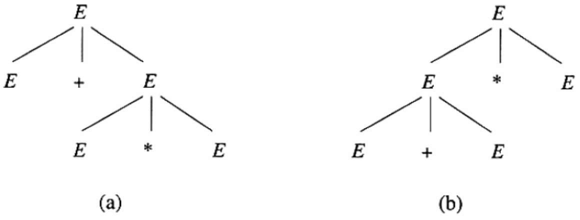
1. $E \Rightarrow E + E \Rightarrow E + E * E$
2. $E \Rightarrow E * E \Rightarrow E + E * E$



Two parse trees with the same yield

- Figure shows the two parse trees, which we should note are distinct trees.

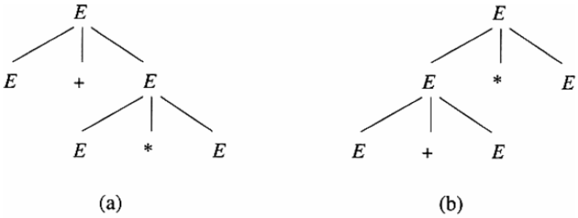
1. $E \Rightarrow E + E \Rightarrow E + E * E$
2. $E \Rightarrow E * E \Rightarrow E + E * E$



Two parse trees with the same yield

- The difference between these two derivations is significant.
- Derivation (1) says that the second and third expressions are multiplied, and the result is added to the first expression.
- Derivation (2) adds the first two expressions and multiplies the result by the third.

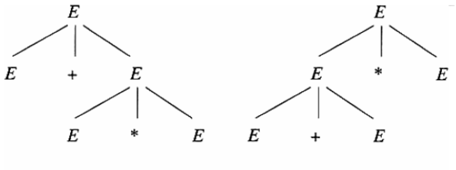
1. $E \Rightarrow E + E \Rightarrow E + E * E$
2. $E \Rightarrow E * E \Rightarrow E + E * E$



Two parse trees with the same yield

- In more concrete terms, the first derivation suggests that $1 + 2 * 3$ should be grouped $1 + (2 * 3) = 7$.
- The second derivation suggests the same expression should be grouped $(1 + 2) * 3 = 9$.
- Obviously, the first of these, and not the second, matches our notion of correct grouping of arithmetic expressions.

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

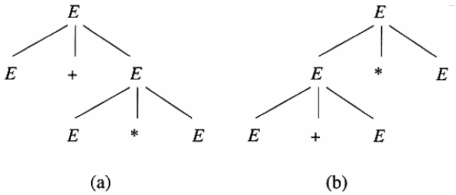


(a) (b)

- The grammar of figure gives two different structures to any string of terminals that is derived by replacing the three expressions in $E + E * E$ by identifiers.
- We see that this grammar is not a good one for providing unique structure.

Example — *continued*

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$



- In particular, while it can give strings the correct grouping as arithmetic expressions, it also gives them incorrect groupings.
- To use this expression grammar in a compiler, we would have to modify it to provide only the correct groupings.

- On the other hand, the mere existence of different derivations for a string (as opposed to different parse trees) does not imply a defect in the grammar.
- The following is an example.

Example

Hopcroft, Motwani, and Ullman, Example 5.26, p-206

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

- Using the same expression grammar, we find that the string $a + b$ has many different derivations.

Example

Hopcroft, Motwani, and Ullman, Example 5.26, p-206

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

- Two examples are:
  - $E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$
  - $E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 5.26, p-206

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

- $E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$
- $E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$

- However, there is no real difference between the structures provided by these derivations.

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 5.26, p-206

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

- $E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$
- $E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$

- They each say that a and b are identifiers, and that their values are to be added.

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 5.26, p-206

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

- $E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$
- $E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$

- The two examples above suggest that it is not a multiplicity of derivations that cause ambiguity, but rather the existence of two or more parse trees.

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 5.26, p-206

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

- $E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$
- $E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$

- Thus, we say a CFG $G(V, T, P, S)$ is *ambiguous* if there is at least one string w in T^* for which we can find two different parse trees, each with root labeled S and yield w.

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 5.26, p-206

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

- $E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$
- $E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$

- If each string has at most one parse tree in the grammar, then the grammar is *unambiguous*.

Ambiguity

Sipser, 2.1, p-107

- Grammar, G_5.

$$
\begin{aligned}
\texttt{<EXPR>} \rightarrow\ & \texttt{<EXPR>} + \texttt{<EXPR>} \\
\mid\ & \texttt{<EXPR>} \times \texttt{<EXPR>} \\
\mid\ & (\texttt{<EXPR>}) \mid a
\end{aligned}
$$

- This grammar doesn’t capture the usual precedence relations and so may group the $+$ before the $\times$ or vice versa.

Ambiguity — *continued*

Sipser, Figure 2.6, p-108

$$\begin{aligned} \texttt{<EXPR>} &\to \texttt{<EXPR>} + \texttt{<EXPR>} \\ &\mid \texttt{<EXPR>} \times \texttt{<EXPR>} \\ &\mid (\texttt{<EXPR>}) \mid a \end{aligned}$$

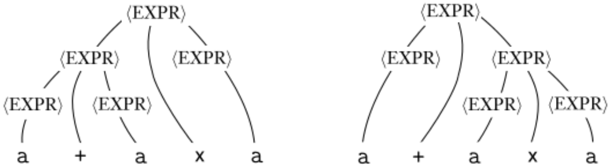


FIGURE 2.6
The two parse trees for the string a+a×a in grammar G_5

Ambiguity — *continued*

Sipser, 2.1, p-107

- In contrast, the following grammar generates exactly the same language, but every generated string has a unique parse tree.
- $$\begin{aligned} \texttt{<EXPR>} &\to \texttt{<EXPR>} + \texttt{<TERM>} \mid \texttt{<TERM>} \\ \texttt{<TERM>} &\to \texttt{<TERM>} \times \texttt{<FACTOR>} \mid \texttt{<FACTOR>} \\ \texttt{<FACTOR>} &\to (\texttt{<EXPR>}) \mid a \end{aligned}$$

Ambiguity — *continued*

Sipser, Figure 2.5, p-105

$$\texttt{<EXPR>} \rightarrow \texttt{<EXPR>} + \texttt{<TERM>} \mid \texttt{<TERM>}$$

$$\texttt{<TERM>} \rightarrow \texttt{<TERM>} \times \texttt{<FACTOR>} \mid \texttt{<FACTOR>}$$

$$\texttt{<FACTOR>} \rightarrow (\texttt{<EXPR>}) \mid a$$

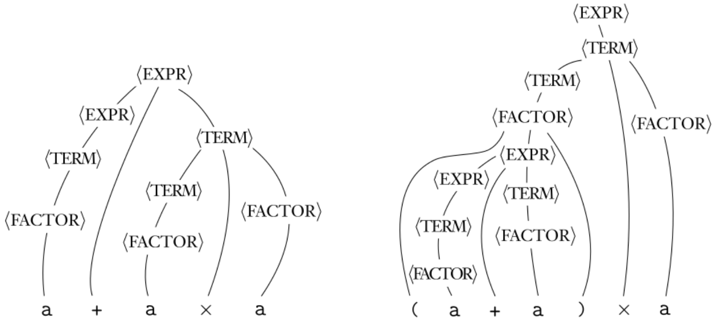


FIGURE 2.5
Parse trees for the strings a+a×a and (a+a)×a

Ambiguity — *continued*

Sipser, Definition 2.7, p-108

Definition 2.7

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

Ambiguity — *continued*

Sipser, Example 2.1, p-108

- Sometimes when we have an ambiguous grammar we can find an unambiguous grammar that generates the same language.
- Some context-free languages, however, can be generated only by ambiguous grammars.
- Such languages are called inherently ambiguous.
- The language $\{a^i b^j c^k \mid i = j, \text{ or}, j = k\}$ is inherently ambiguous.

End of Slides