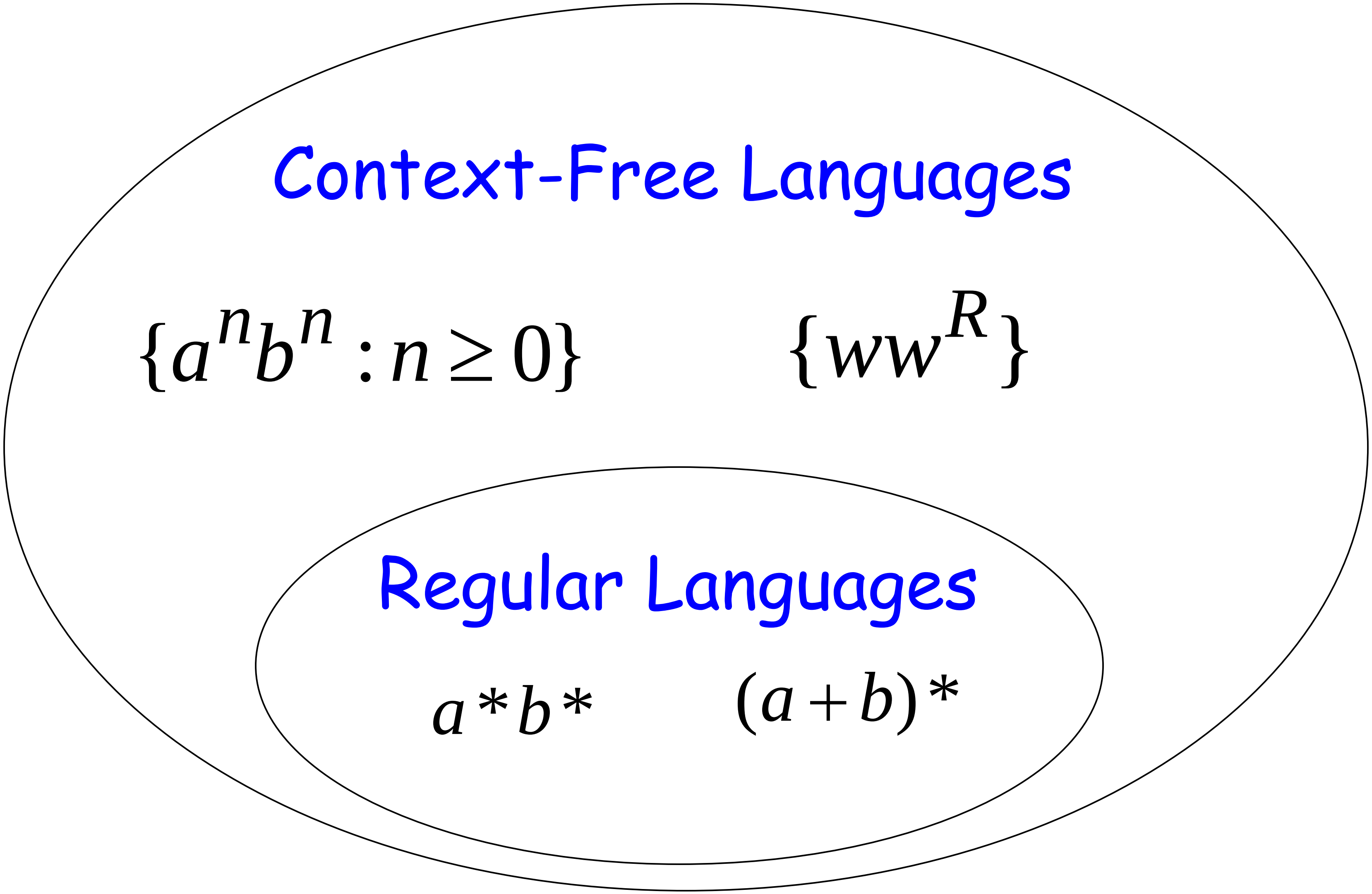
Lecture 11

Context-Free Languages

COT 4420

Theory of Computation

Chapter 5


Context-Free Languages
$\{a^n b^n : n \geq 0\}$
$\{ww^R\}$
Regular Languages
$a^* b^*$
$(a+b)^*$

Example 1

$G = (\{S\}, \{a, b\}, S, P)$

$S \rightarrow aSb$

$S \rightarrow \lambda$

Derivations:

S => aSb => aaSbb => aabb

S => aSb => aaSbb => aaaSbbb => aaaabbbb

Notation

We write: $S \overset{*}{\Rightarrow} aaabbb$

for zero or more derivation steps

Instead of:

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

Example

Grammar:

$S \rightarrow aSb$

$S \rightarrow \lambda$

Possible Derivations:

$S \overset{*}{\Rightarrow} \lambda$

$S \overset{*}{\Rightarrow} ab$

$S \overset{*}{\Rightarrow} aaaSbbb \overset{*}{\Rightarrow} aaaaabbbbb$

Language of a Grammar

- For a grammar G with start variable S

$$L(G) = \{ w : S \overset{*}{\Rightarrow} w, \; w \in T^* \}$$

Example

Grammar:

$$S \rightarrow aSb$$
$$S \rightarrow \lambda$$

Language of the grammar:

$$L = \{a^n b^n : n \geq 0\}$$

Context-Free Grammar

- A grammar G=(V, T, S, P) is context-free if all productions in P have the form:

Sequence of terminals and variables

$$A \rightarrow \overbrace{x} \quad \text{where} \quad A \in V \quad \text{and} \quad x \in (V \cup T)^*$$

- A language L is a context-free language iff there is a context-free grammar G such that L = L(G)

Context-Free Language

$L = \{a^n b^n : n \geq 0\}$ is a context-free language since context-free grammar:

$S \rightarrow aSb \mid \lambda$ generates $L(G) = L$

Another Example

Context-free grammar G:

$S \rightarrow aSa \mid bSb \mid \lambda$

A derivation: S => aSa => abSba => abba

$L(G) = \{ ww^R : w \in \{a,b\}^* \}$

Another Example

Context-free grammar G:

$$S \rightarrow (S) \mid SS \mid \lambda$$

A derivation:

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

L(G) : balanced parentheses

Example 2

$$L = \{ a^n b^m : n \neq m \}$$

$n > m$

aaaaaaaabbbbb

$S \to AS_1$

$S_1 \to aS_1b \mid \lambda$

$A \to aA \mid a$

$n < m$

aaaaabbbbbbbbb

$S \to S_1B$

$S_1 \to aS_1b \mid \lambda$

$B \to bB \mid b$

$S \to AS_1 \mid S_1B$

$S_1 \to aS_1b \mid \lambda$

$A \to aA \mid a$

$B \to bB \mid b$

Example 3

Suppose I have this grammar:

$S \rightarrow aB \mid bA$

$A \rightarrow aS \mid bAA \mid a$

$B \rightarrow bS \mid aBB \mid b$

Claim: L(G) is all words over {a, b} that have an equal number of a's and b's (excluding λ).

Proof: by induction on the size of $w \in L(G)$.

Proof by induction

Induction hypothesis:

1. S =>* w iff w has an equal number of a's and b's
2. A =>* w iff w has one more a's than b's
3. B =>* w iff w has one more b's than a's

Basis: true for $|w| = 1$ ✓

Inductive step: assume it is true for $|w| \leq k-1$

1. S =>* w iff w has an equal number of a's and b's

If S =>* w then w has an equal number of a's and b's

Suppose S=>* w, |w| = k. The first derivation must be $S \rightarrow aB$ or $S \rightarrow bA$. Suppose it is $S \rightarrow aB$. Then w = aw_1 where B =>* w_1. Since $|w_1|$ = k-1 by induction hypothesis (3) w_1 has one more b's than a's. Therefore w has equal number of a's and b's.

We can prove similarly if the first step is using the rule $S \rightarrow bA$.

1. S =>* w iff w has an equal number of a's and b's

If w has an equal number of a's and b's then S=>* w

Assume |w| = k and w has equal number of a's and b's. Suppose w = aw_1. So w_1 must have one more b's than a's. By induction hypothesis since $|w_1|$= k-1, B =>* w_1. Thus S => aB =>* aw_1 = w.

Therefore, S =>* w

Similarly if w = bw_1.

2. A =>* w iff w has one more a's than b's

If A=>* w then w has one more a's than b's.

Suppose A=>* w and |w| = k>1. Then the first derivation step must be $A \to aS$ or $A \to bAA$.

In the first case, S =>* w_1 with w_1 having equal a's and b's. In the second case first rhs A=>* w_1 and second rhs A=>* w_2, with w_1 and w_2 having one more a's than b's. Thus, A =>* bw_1w_2 has one more a's than b's overall.

2. A =>* w iff w has one more a's than b's

If w has one more a's than b's then A =>* w

Assume w has one more a's than b's and |w|=k.

Let w= aw_1. By induction S =>* w_1 therefore, A=> aS =>* aw_1=w.

Let w = bw_2. Now w_2 has two more a's than b's and can be written as $w_2 = w_3w_4$ with w_3 having one more a's than b's and w_4 having one more a's than b's (Why this is true?), by induction

A=>* w_3 and A =>* w_4 therefore:

A=> bAA =>* bw_3w_4 = w

Derivations

Derivations

- When a sentential form has a number of variables, we can replace any one of them at any step.

- As a result, we have many different derivations of the same string of terminals.

Derivations

Example: 1. $S \rightarrow aAS$ 2. $S \rightarrow a$

3. $A \rightarrow SbA$ 4. $A \rightarrow SS$ 5. $A \rightarrow ba$

$\underline{S} \overset{1}{\Rightarrow} aA\underline{S} \overset{2}{\Rightarrow} a\underline{A}a \overset{3}{\Rightarrow} aSb\underline{A}a \overset{4}{\Rightarrow} aSbS\underline{S}a \overset{2}{\Rightarrow} aSb\underline{S}aa$

$\overset{2}{\Rightarrow} a\underline{S}baaa \overset{2}{\Rightarrow} aabaaa$

$\underline{S} \overset{1}{\Rightarrow} a\underline{A}S \overset{3}{\Rightarrow} aSbA\underline{S} \overset{2}{\Rightarrow} aSb\underline{A}a \overset{4}{\Rightarrow} a\underline{S}bSSa \overset{2}{\Rightarrow}$

$aabS\underline{S}a \overset{2}{\Rightarrow} aab\underline{S}aa \overset{2}{\Rightarrow} aabaaa$

Leftmost Derivation

A derivation is said to be leftmost if in each step the leftmost variable in the sentential form is replaced.

Example:

$S \rightarrow aAS \mid a$

$A \rightarrow SbA \mid SS \mid ba$

S => aAS => aSbAS => aabAS => aabSSS => aabaSS => aabaaS => aabaaa

Leftmost

Rightmost Derivation

A derivation is said to be rightmost if in each step the rightmost variable is replaced.

Example: 1. $S \rightarrow aAS$ 2. $S \rightarrow a$

3. $A \rightarrow SbA$ 4. $A \rightarrow SS$ 5. $A \rightarrow ba$

$$S \overset{1}{\Rightarrow} aAS \overset{2}{\Rightarrow} aAa \overset{3}{\Rightarrow} aSbAa \overset{4}{\Rightarrow} aSbSSa \overset{2}{\Rightarrow} aSbSaa$$

$$\overset{2}{\Rightarrow} aSbaaa \overset{2}{\Rightarrow} aabaaa$$

Rightmost

Leftmost and Rightmost Derivation

Example: 1. $S \rightarrow aAS$ 2. $S \rightarrow a$

3. $A \rightarrow SbA$ 4. $A \rightarrow SS$ 5. $A \rightarrow ba$

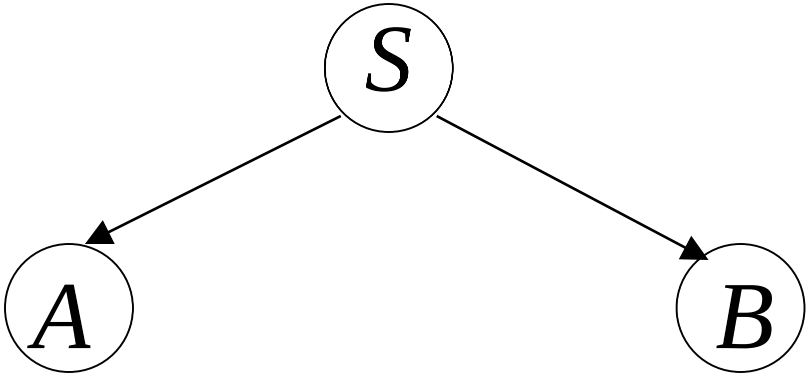
$\underline{S}$ => a$\underline{A}$S => aSbA$\underline{S}$ => aSb$\underline{A}$a => a$\underline{S}$bSSa =>
aabS$\underline{S}$a => aab$\underline{S}$aa => aabaaa

Neither

Derivation Trees

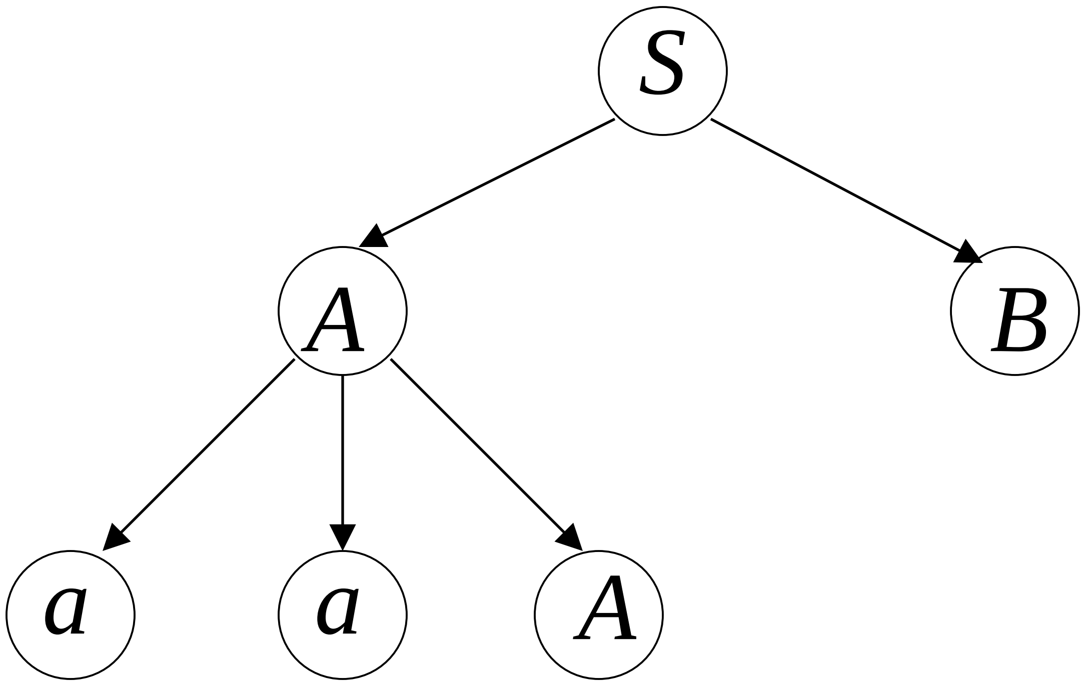
$$S \to AB \qquad A \to aaA \mid \lambda \qquad B \to Bb \mid \lambda$$

$$S \Rightarrow AB$$



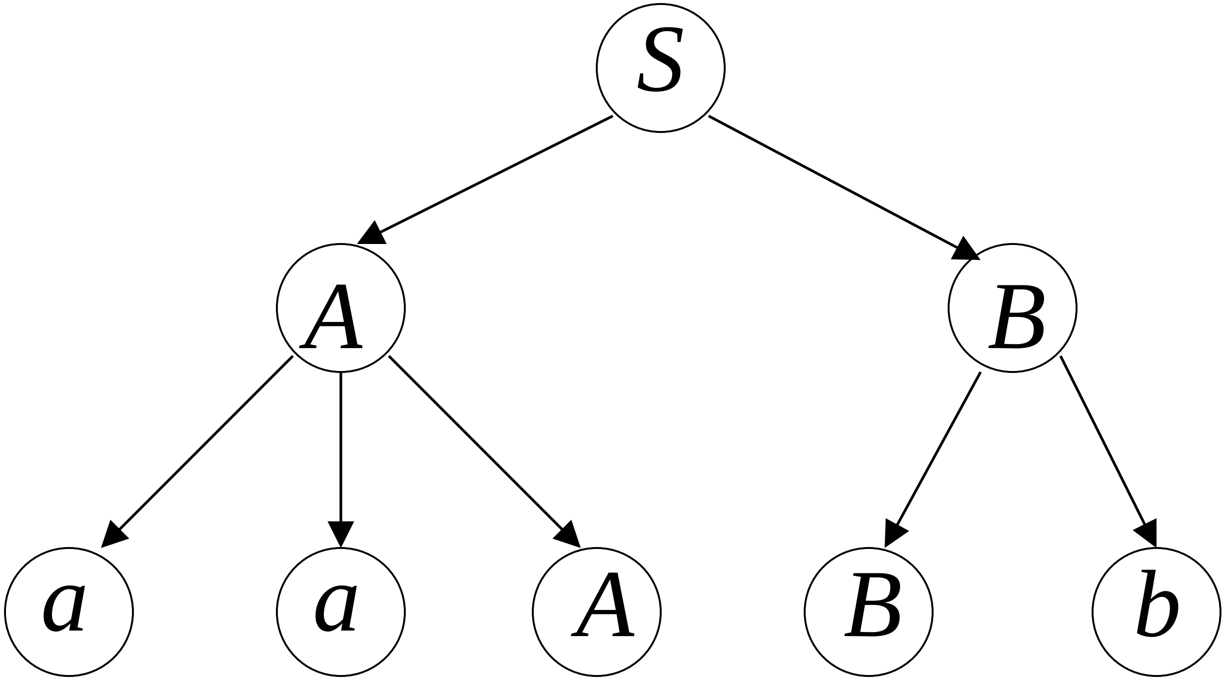
$$S \to AB \qquad A \to aaA \mid \lambda \qquad B \to Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB$$



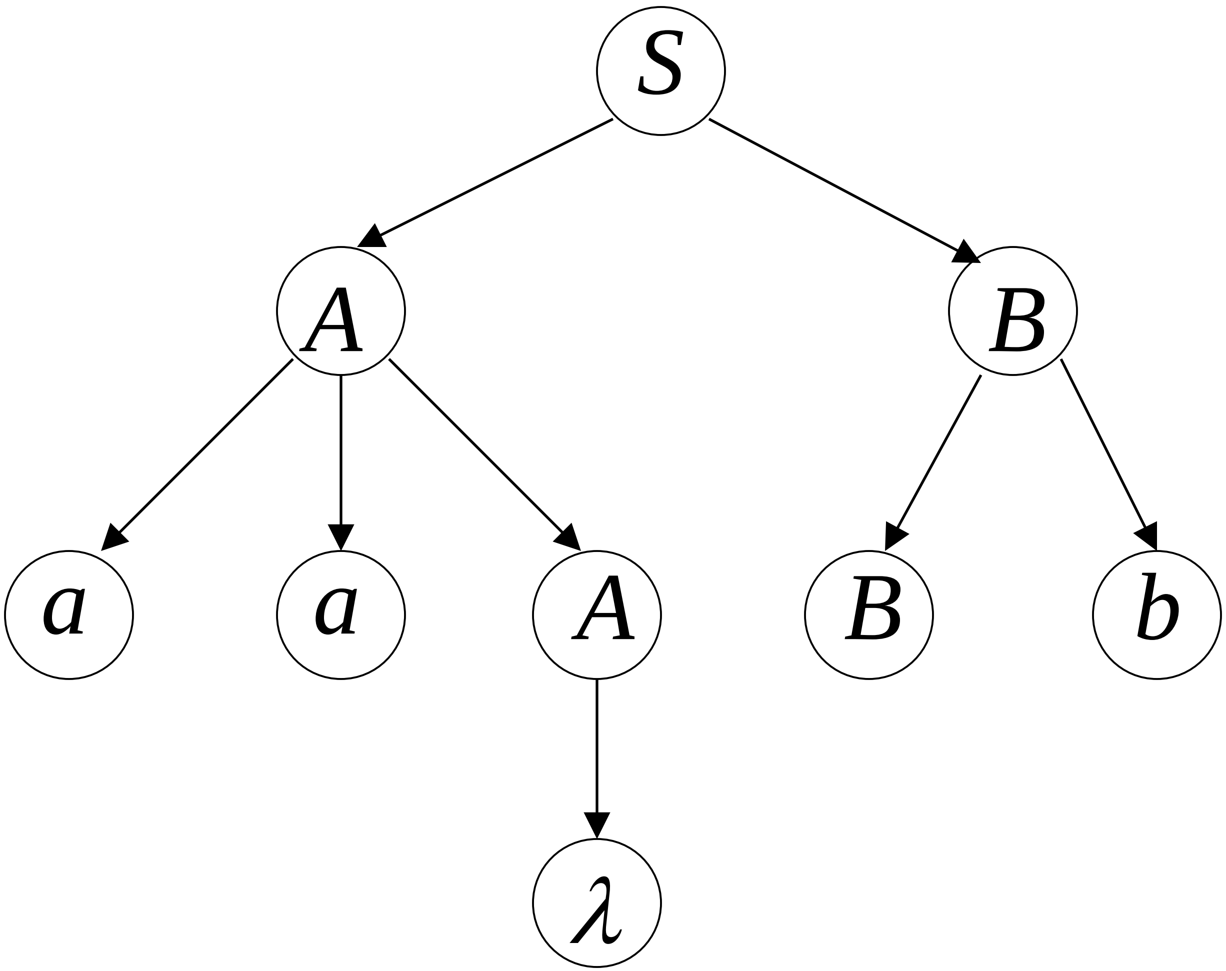
$$S \to AB \qquad A \to aaA \mid \lambda \qquad B \to Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb$$



$$S \to AB \qquad A \to aaA \mid \lambda \qquad B \to Bb \mid \lambda$$

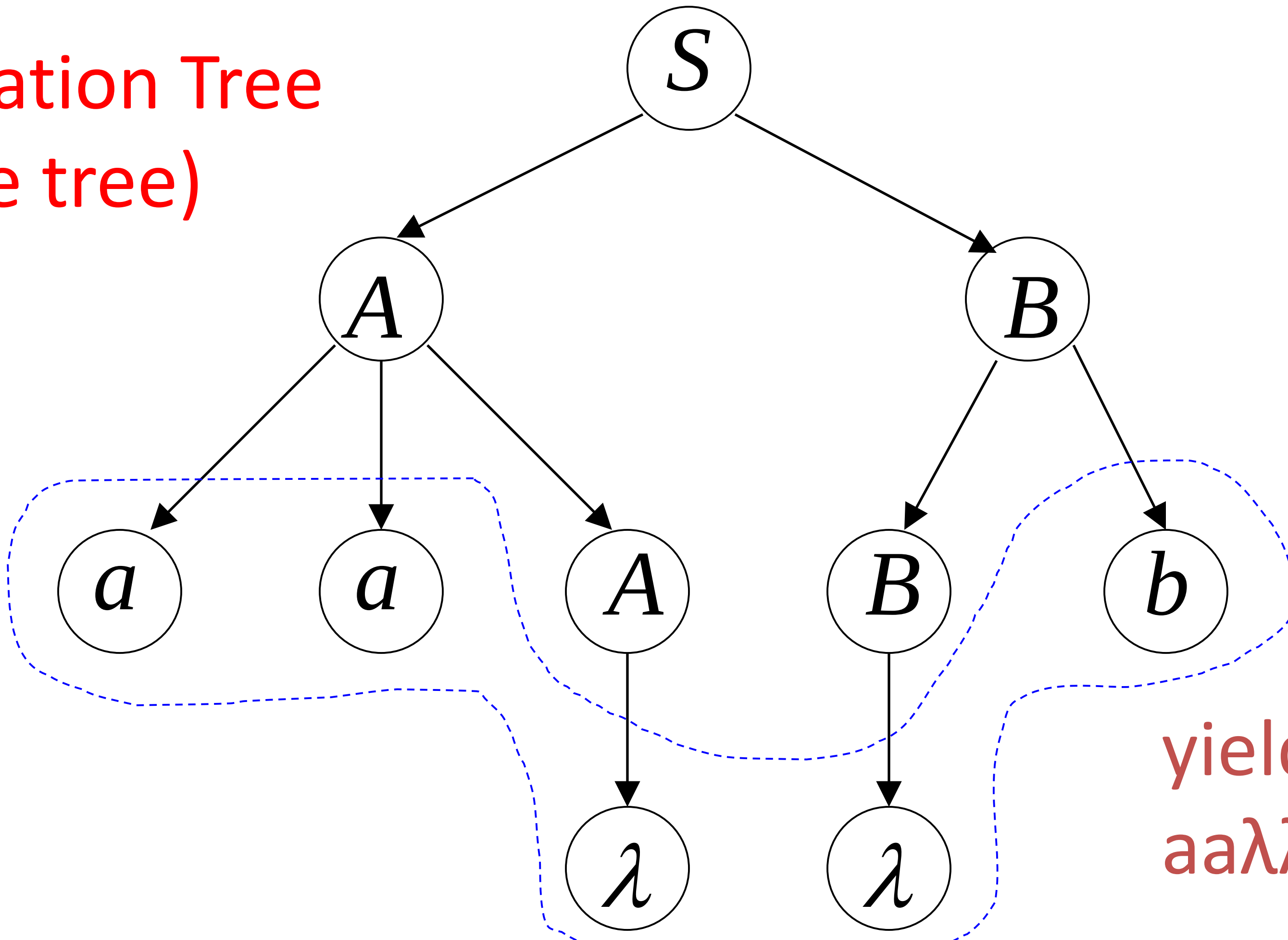
$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb$$



$$S \to AB \qquad A \to aaA \mid \lambda \qquad B \to Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$$

Derivation Tree
(parse tree)



yield
aaλλb = aab

Derivation Trees

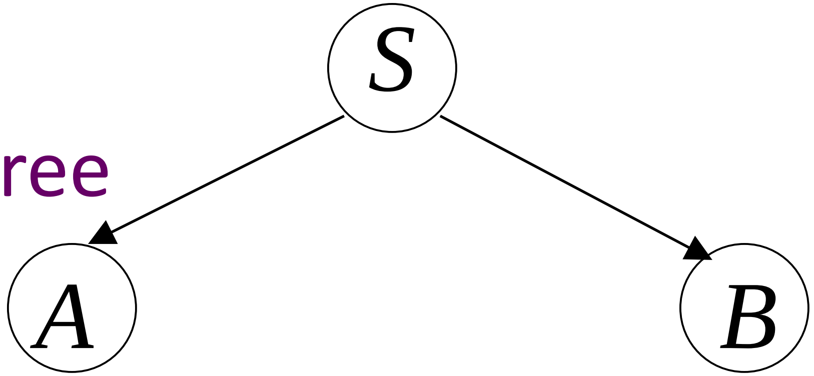
- Derivation trees are trees whose nodes are labeled by symbols of a CFG.
- Root is labeled by S (start symbol).
- Leaves are labeled by terminals $T \cup \{\lambda\}$
- Interior nodes are labeled by non-terminals V.
- If a node has label $A \in V$, and there is a production rule $A \rightarrow \alpha_1\alpha_2 \ldots \alpha_n$ then its children are labeled from left to right $\alpha_1, \alpha_2, \ldots, \alpha_n$.
- The string of symbols obtained by reading the leaves from left to right is said to be the yield.

Partial Derivation Tree

A partial derivation tree is a subset of the derivation tree (the leaves can be non-terminals or terminals.

$$S \to AB \qquad A \to aaA \mid \lambda \qquad B \to Bb \mid \lambda$$

Partial derivation tree

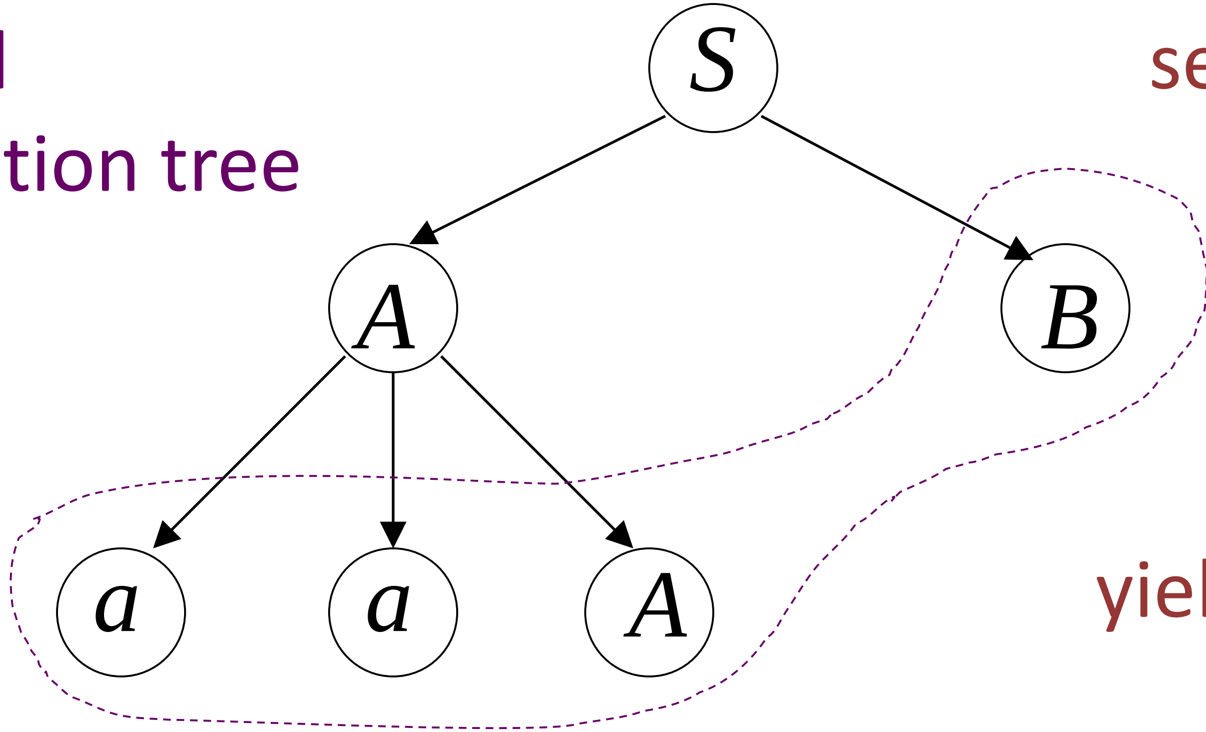


Partial Derivation Tree

$$S \Rightarrow AB \Rightarrow aaAB$$

sentential form

Partial derivation tree



yield aaAB

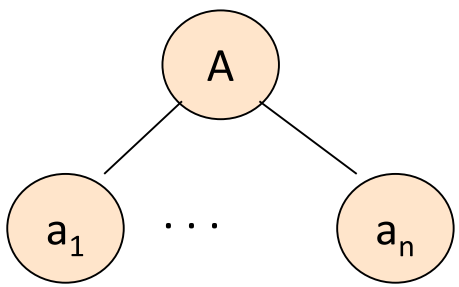
- Theorem:

1) If there is a derivation tree with root labeled A that yields w, then $A \Rightarrow^*_{lm} w$.

2) If $A \Rightarrow^*_{lm} w$, then there is a derivation tree with root A that yields w.

Proof - part 1

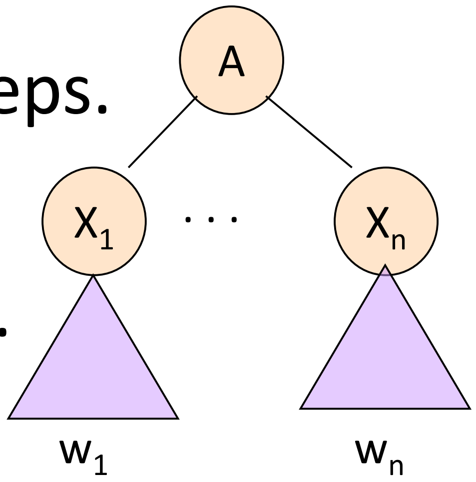
- Proof: by induction on the height of the tree
- Basis: if height is 1, $A \rightarrow a_1a_2 \ldots a_n$ must be a production rule. Therefore, $A \Rightarrow^*_{lm} a_1a_2 \ldots a_n$



- Inductive step: Assume it is true for trees of height < h, and you want to prove for height h.

Since h > 1, the production used at root has at least one variable on its right side.

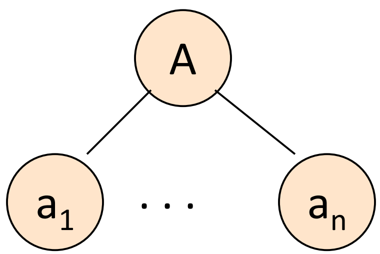
Proof - part 1



- Assume node A has children $X_1, X_2, \ldots, X_n$. Each of these X_i yield w_i in at most h-1 steps.
- Note that X_i might be a terminal, in that case $X_i = w_i$ and nothing needs to be done.
- If X_i is a non-terminal, because of the induction hypothesis we know that there is a leftmost derivation $X_i \Rightarrow^*_{lm} w_i$
- Thus, $A \Rightarrow_{lm} X_1 \ldots X_n \Rightarrow^*_{lm} w_1 X_2 \ldots X_n \Rightarrow^*_{lm} w_1 w_2 X_3 \ldots X_n \Rightarrow^*_{lm} \ldots \Rightarrow^*_{lm} w_1 \ldots w_n = w$.

Proof – part 2

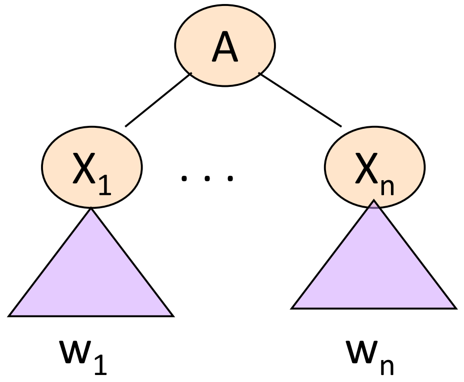
- Proof: by induction on the length of the derivation
- Basis: if $A \Rightarrow_{lm} a_1 a_2 \ldots a_n$ by a one step derivation then there must be a derivation tree



- Inductive step: Assume it is true for derivations of < k steps, and let $A \Rightarrow^*_{lm} w$ be a derivation of k steps. Since k>1, the first step must be $A \Rightarrow_{lm} X_1 X_2 \ldots X_n$

Proof – part 2

- If X_i is a terminal, in that case $X_i = w_i$ and nothing needs to be done.
- If X_i is a nonterminal $X_i =>^*_{lm} w_i$ in at most k-1 steps. By the induction hypothesis there is a derivation tree with root X_i and yield w_i.
- So we create the derivation tree as follows:



Ambiguity

Ambiguous grammars
Example

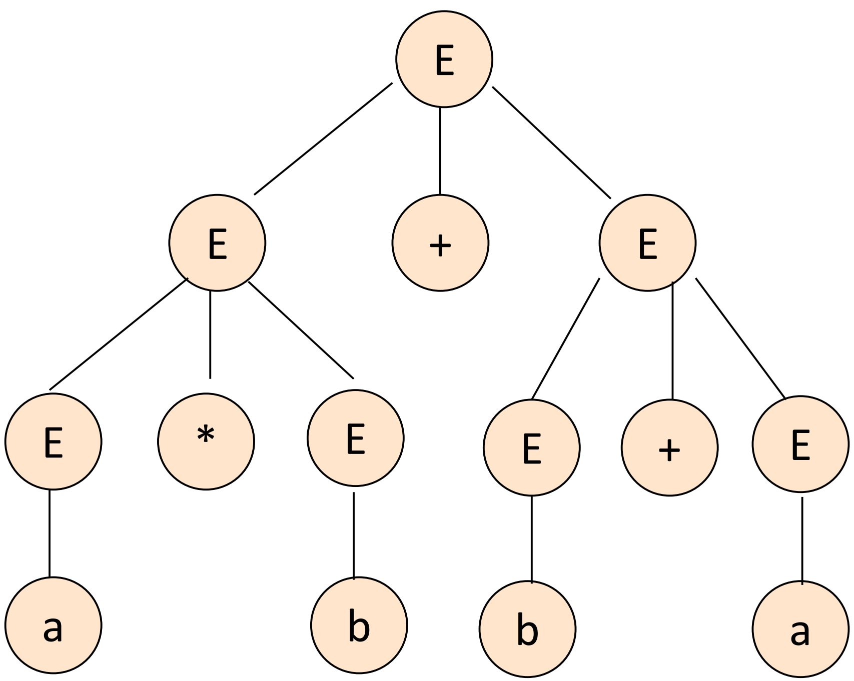
$E \rightarrow E + E$

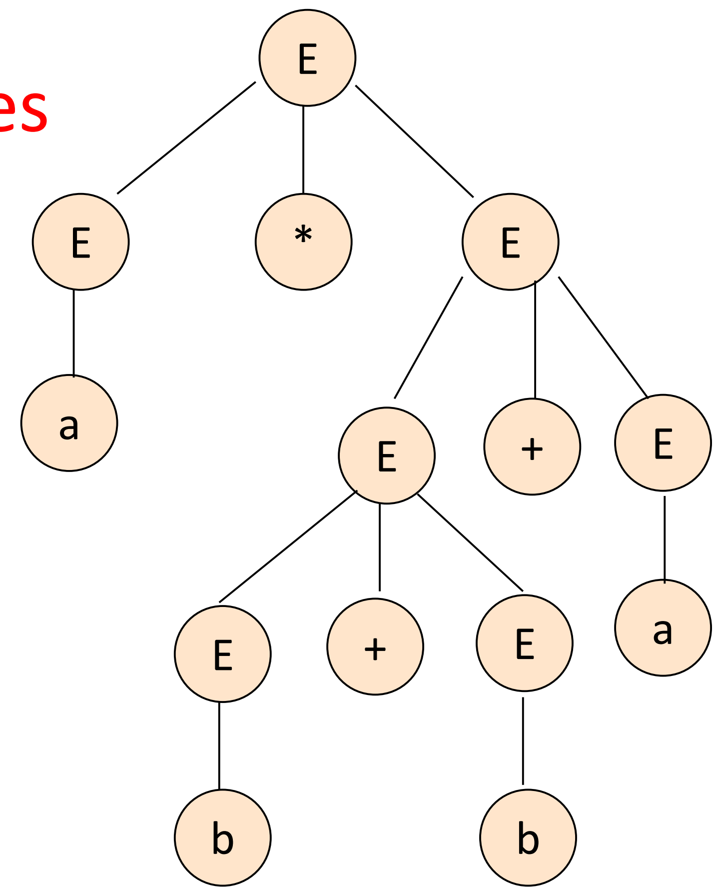
$E \rightarrow E * E$

$E \rightarrow a \mid b$

a * b + b + a

Two derivation trees





Example

E → E + E   E → E * E   E → a | b

E => E + E => E * E + E => a * E + E => a * b + E =>

a * b + E + E => a * b + b + E => a*

Leftmost derivation

E => E * E => a * E => a * E + E => a * E + E + E

=> a * b + E + E => a * b + b + E => a

Leftmost derivation

Ambiguous grammars

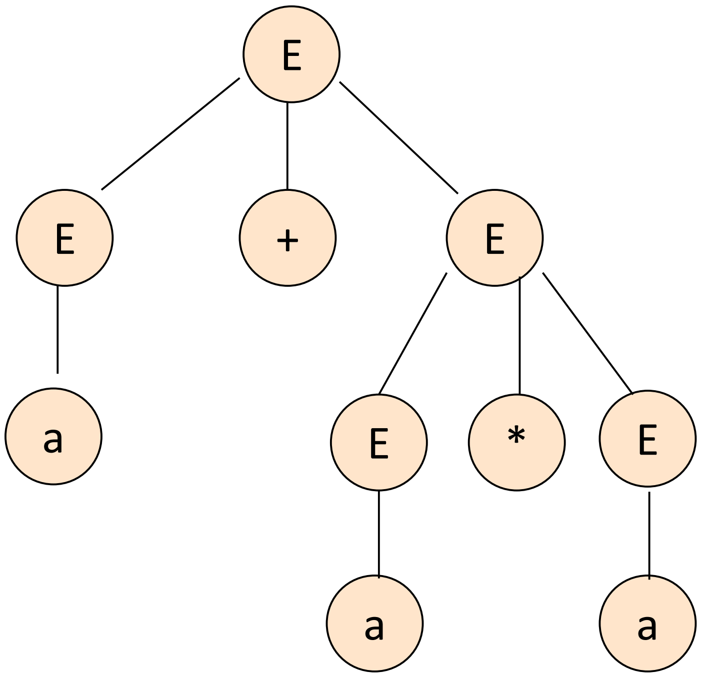
- A context-free grammar G is ambiguous if there exist some w $\in$ L(G) that has at least two distinct derivation trees.

- Or if there exists two or more leftmost derivations (or rightmost).

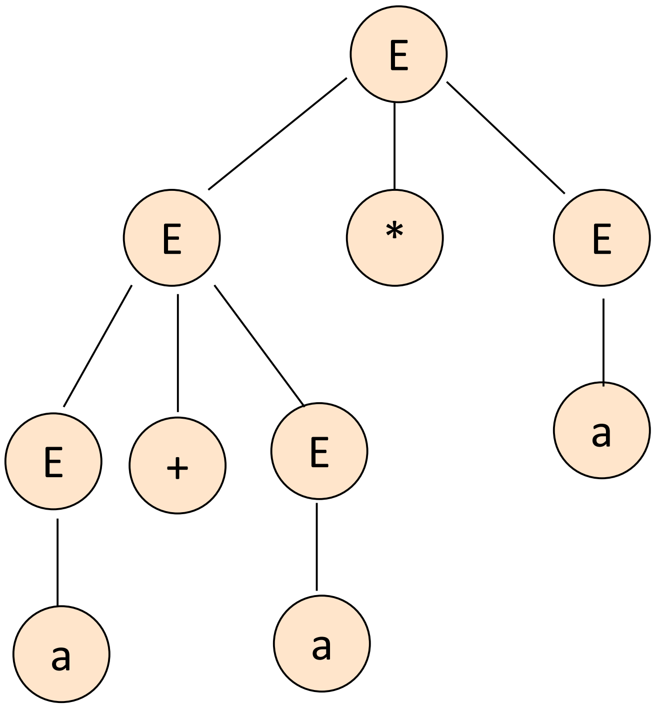
Why do we care about ambiguity?

Grammar for mathematical expressions:

$E \rightarrow E + E$ $E \rightarrow E * E$ $E \rightarrow a$

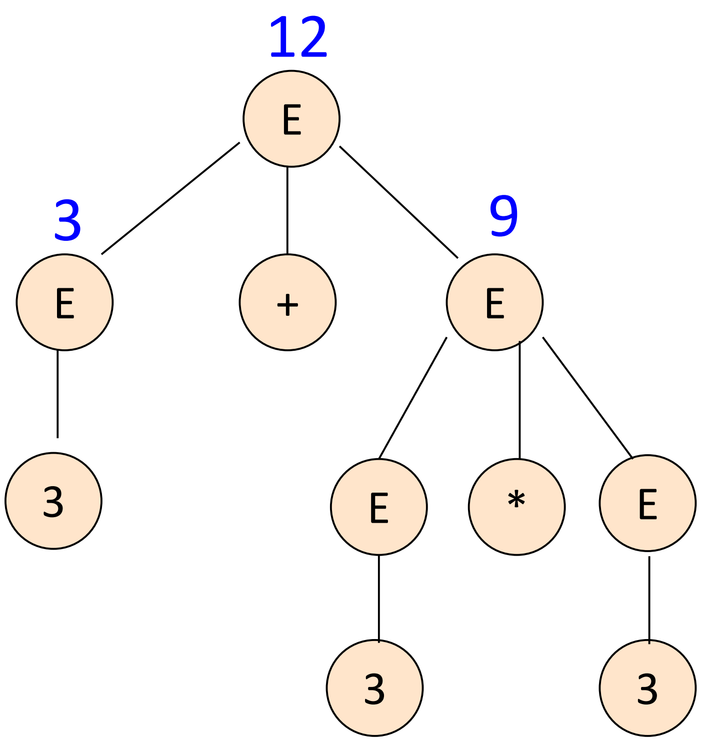


a + a * a

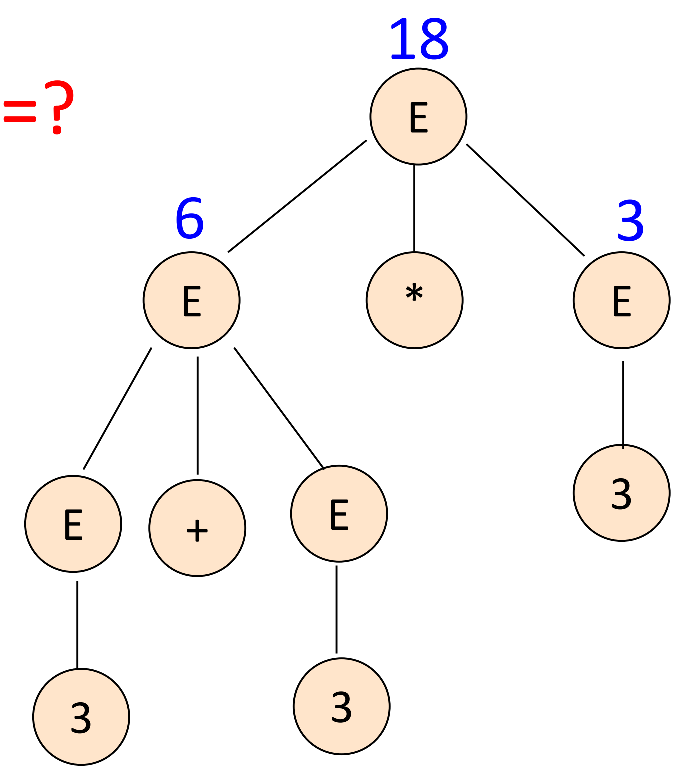


Why do we care about ambiguity?

Compute expressions result using the tree

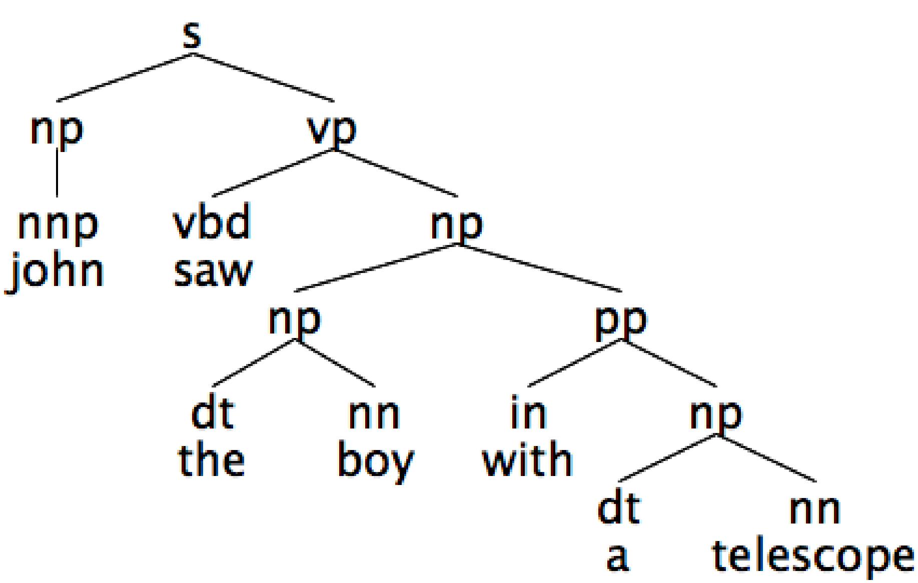


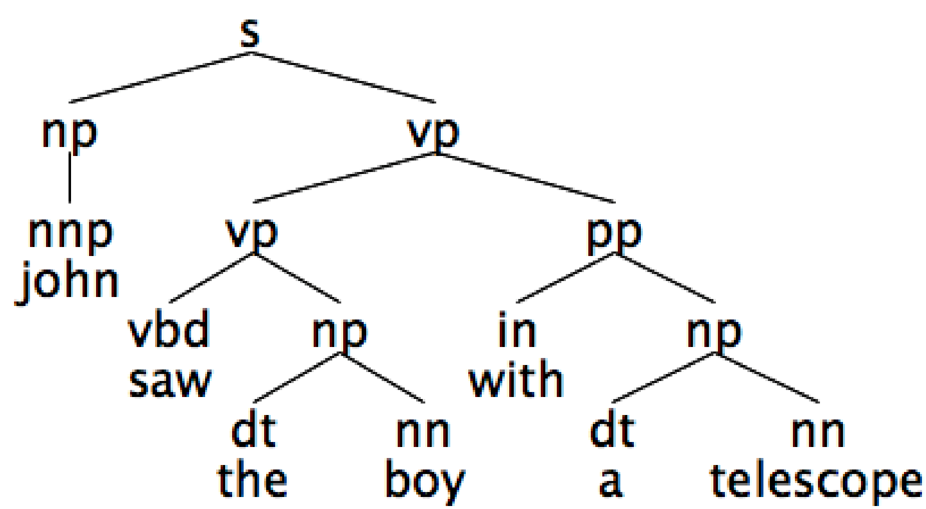
3 + 3 * 3 =?



Why do we care about ambiguity?

John saw the boy with a telescope.





Ambiguity

- In general, ambiguity is bad for programming languages and we want to remove it

- Sometimes it is possible to find a non-ambiguous grammar for a language

- But in general it is difficult to achieve this

Non-ambiguous Grammar Example

- Can we rewrite the previous grammar so that it is not ambiguous anymore?

Equivalent non-ambiguous grammar:

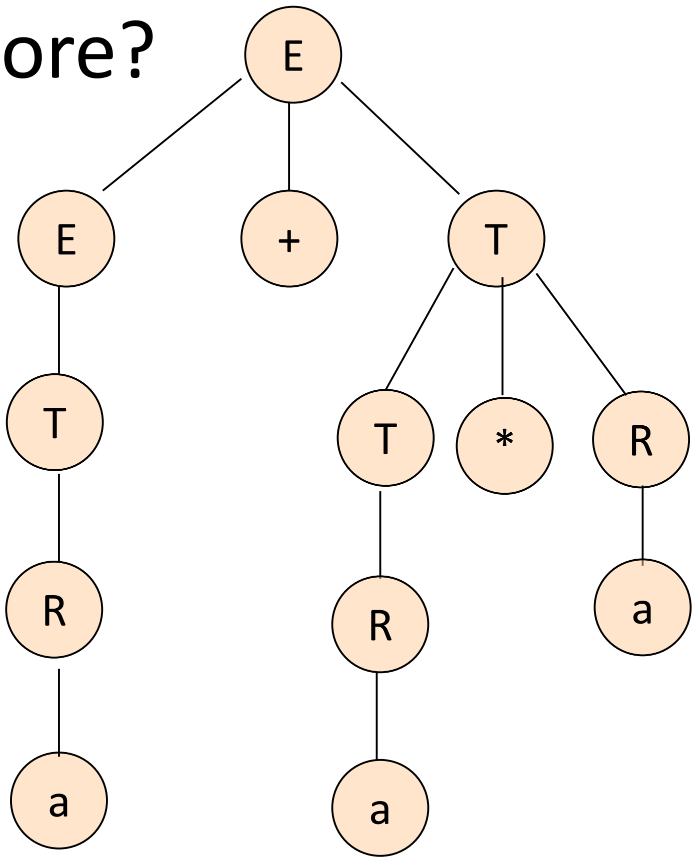
(Generates the same language)

$E \rightarrow E + T \mid T$

$T \rightarrow T * R \mid R$

$R \rightarrow a$

❖ Every string w in L(G) has a unique derivation tree



Unique derivation tree for a + a * a

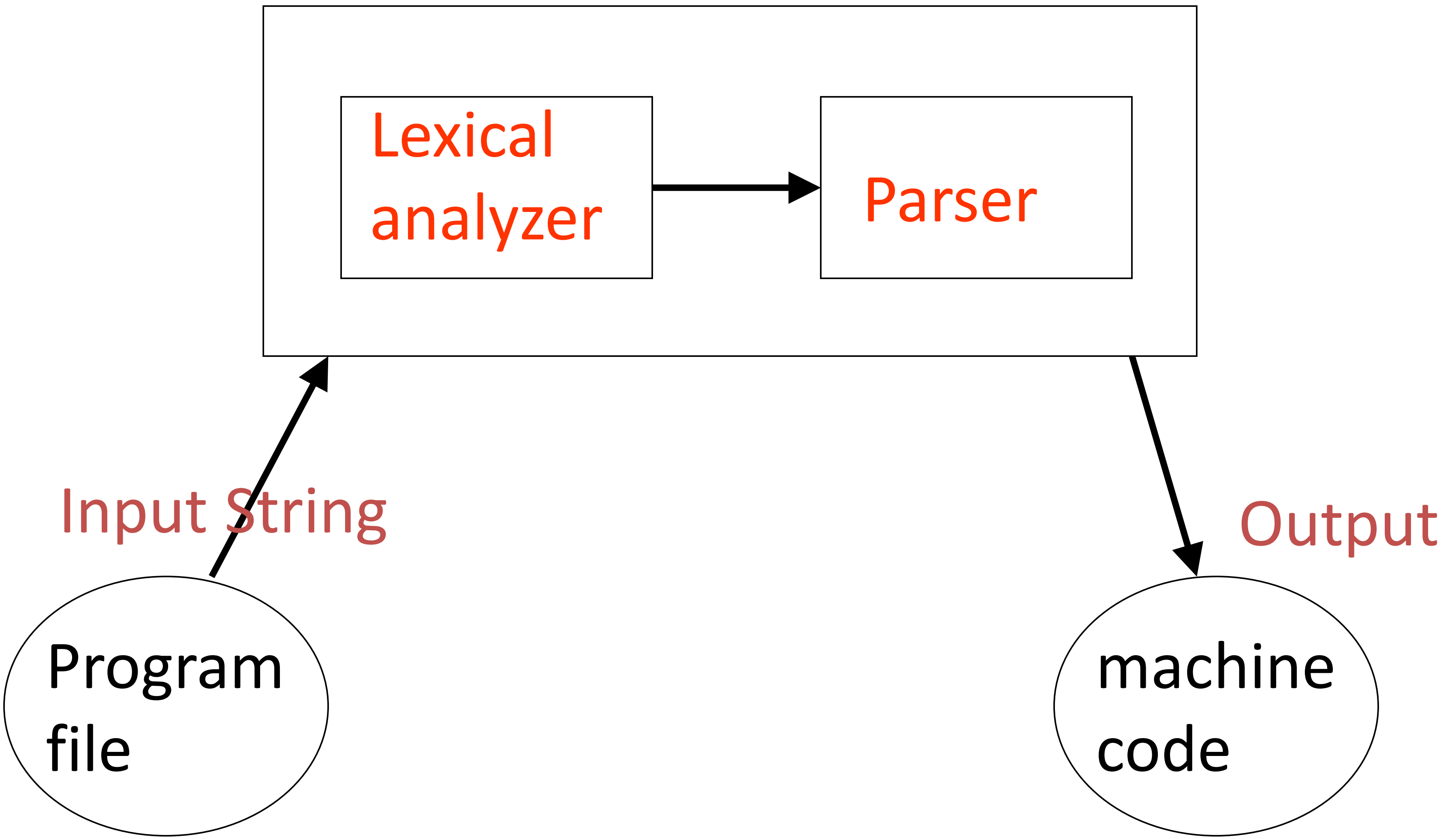
Ambiguous Grammars

- 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.

- In general it is very difficult to show whether or not a language is inherently ambiguous.

Parsing

Compiler



Lexical analyzer
Parser
Input String
Output
Program file
machine code

Lexical Analyzer

- Recognizes the lexemes of the input program file:

  Keywords (if, then, else, while,…),
  Integers,
  Identifiers (variables), etc
  Removes white space and comments

Lexical Analyzer

- Examples:

  letter $\rightarrow$ A | B | … | Z | a | b | … | z

  digit $\rightarrow$ 0 | 1 | … | 9

- digit: [0-9]

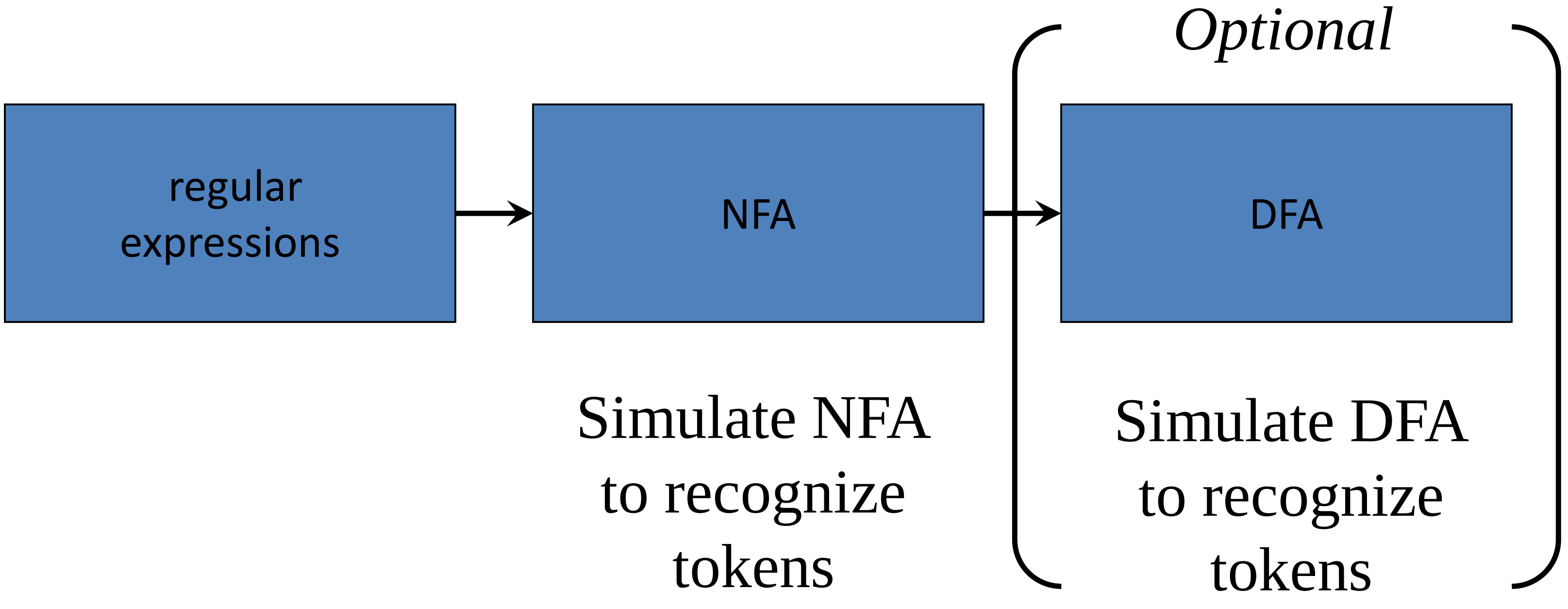
  letter: [a-zA-Z]

  num: digit$^+$ (. digit$^+$)? (E (+ | -)? digit$^+$)?

  identifier: letter (letter | digit)*

Design of a Lexical Analyzer Generator

- Translate regular expressions to NFA
- Translate NFA to an efficient DFA



Parser

- Parsing = *process of determining if a string of tokens can be generated by a grammar*
- Knows the grammar of the programming language to be compiled

- Constructs derivation (and derivation tree) for input program file (input string)

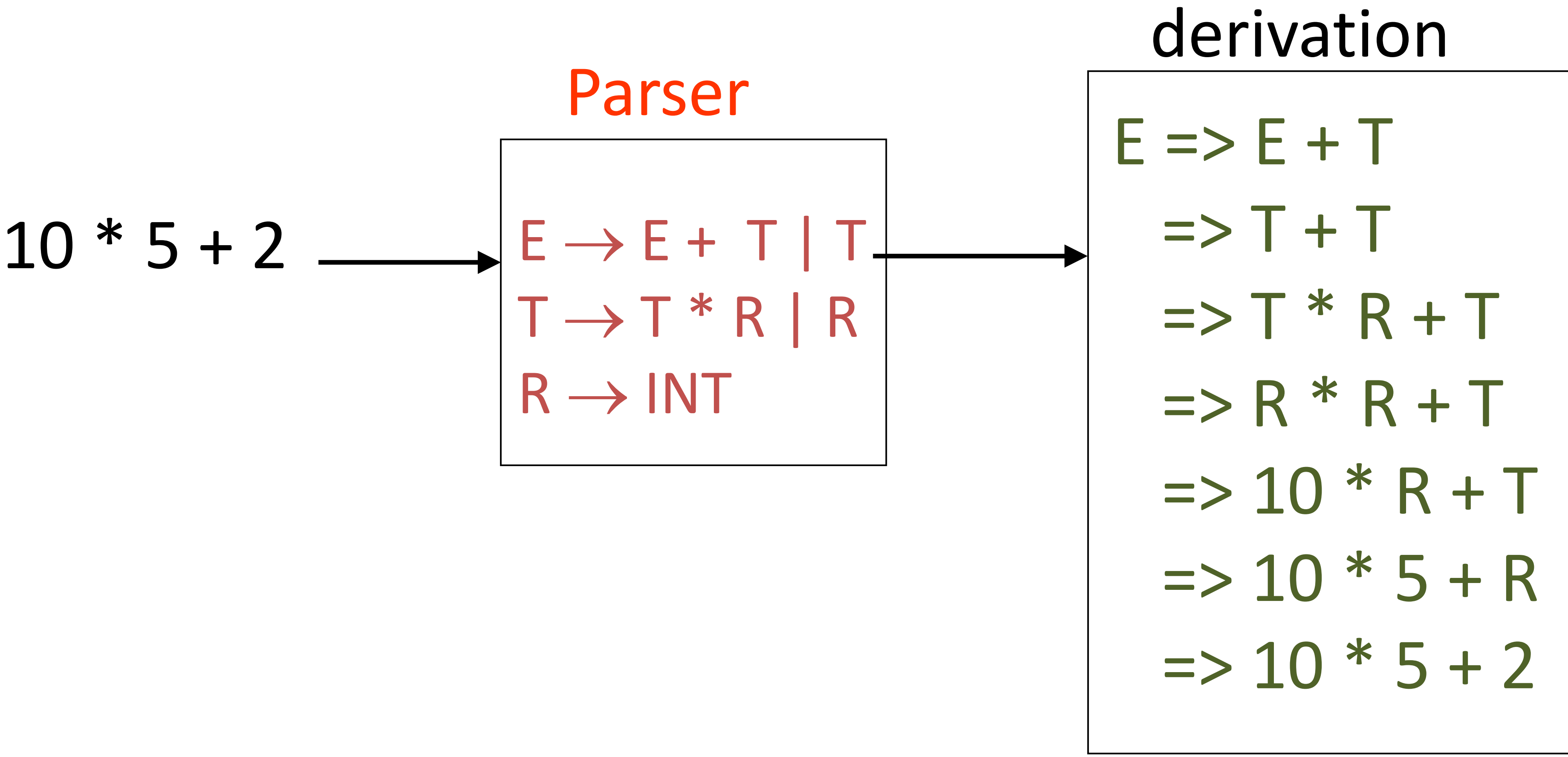
- Converts derivation to machine code

Example Parser

$$
\begin{aligned}
stmt &\rightarrow \textbf{id} \ \textbf{:=} \ expr \\
&\mid \textbf{if} \ expr \ \textbf{then} \ stmt \\
&\mid \textbf{if} \ expr \ \textbf{then} \ stmt \ \textbf{else} \ stmt \\
&\mid \textbf{while} \ expr \ \textbf{do} \ stmt \\
&\mid \textbf{begin} \ opt\_stmts \ \textbf{end} \\
opt\_stmts &\rightarrow stmt \ \textbf{;} \ opt\_stmts \\
&\mid \varepsilon
\end{aligned}
$$

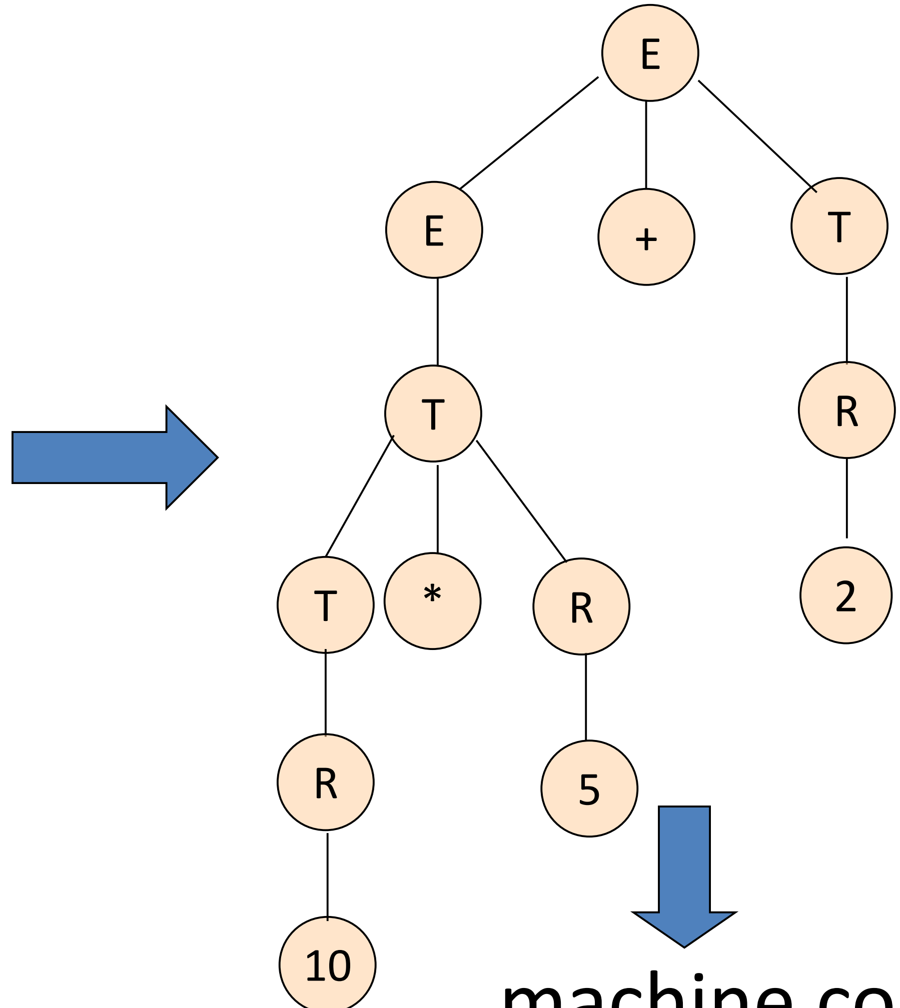
Parser

- Finds the derivation of a particular input



derivation

E => E + T
=> T + T
=> T * R + T
=> R * R + T
=> 10 * R + T
=> 10 * 5 + R
=> 10 * 5 + 2



machine code

mult a, 2, 5
add b, 10, a

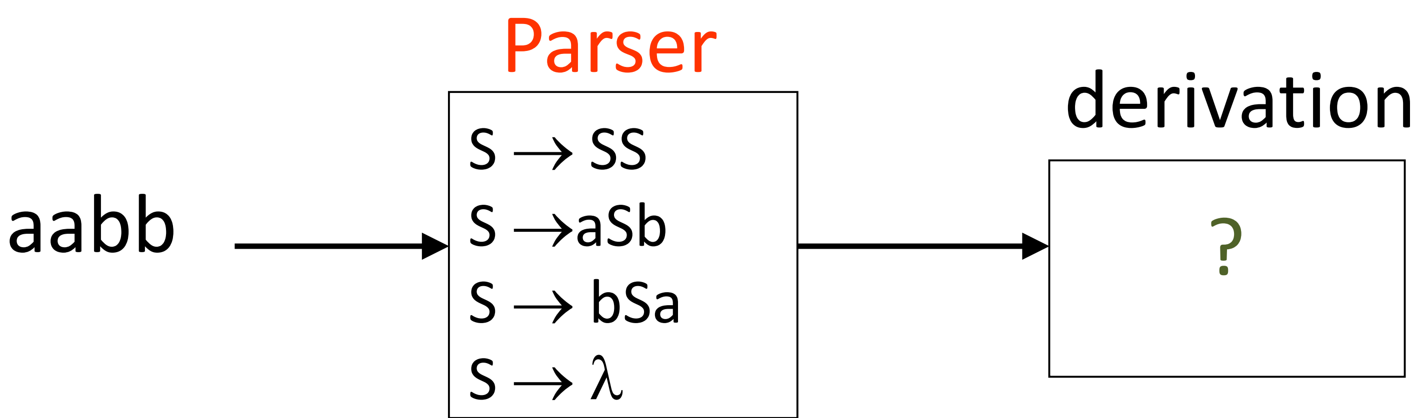
Derivation trees are used to build machine code

Parsing

- Parsing of a string $w \in L(G)$ is to find a sequence of productions by which w is derived or to determine that $w \notin L(G)$.

Example: Find derivation of string *aabb*

$S \rightarrow SS \mid aSb \mid bSa \mid \lambda$



Exhaustive search / Brute force

- $S \rightarrow SS \mid aSb \mid bSa \mid \lambda$ w = aabb

First derivation:

S => SS ✓

S => aSb ✓

S => bSa ✗

S => λ ✗

Cannot possibly produce aabb

All possible derivations of length 1

Exhaustive search / Brute force

- $S \rightarrow SS \mid aSb \mid bSa \mid \lambda$ w = aabb

First derivation:

S => SS ✓

S => aSb ✓

S => bSa ✗

S => λ ✗

Second derivation:

S => SS => SSS ✓

S => SS => aSbS ✓

S => SS => bSaS ✗

S => SS => S ✓

S => aSb => aSSb ✓

S => aSb => aaSbb ✓

S => aSb => abSab ✗

S => aSb => ab ✗

Exhaustive search / Brute force

- $S \rightarrow SS \mid aSb \mid bSa \mid \lambda$

w = aabb

Second derivation:

S => SS => SSS ✓

S => SS => aSbS ✓

S => SS => bSaS X

S => SS => S ✓

S => aSb => aSSb ✓

S => aSb => aaSbb ✓

S => aSb => abSab X

S => aSb => ab X

Third derivation:

Explore all possible derivations

A possible derivation found:

S => aSb = aaSbb => aabb

Exhaustive search / Brute force

- This approach is called exhaustive search parsing or brute force parsing which is a form of top-down parsing.

- Can we use this approach as an algorithm for determining whether or not $w \in L(G)$?

- Theorem: Suppose a CFG has no rules of the form $A \to \lambda$ and $A \to B$. Then the exhaustive search parsing method can be made into an algorithm to parse $w \in \Sigma^*$.

- Proof: In each derivation step, either the length of the sentential form or the number of terminals increases. Therefore, the maximum length of a derivation is $2|w|$. If w is parsed by then, you have the parse. If not, $w \notin L(G)$.

Parsing algorithm

- The exhaustive search algorithm is not very efficient since it may grow exponentially with the length of the string.

❖ For general context-free grammars there exists a parsing algorithm that parses a string w in time $|w|^3$

Faster Parsers

- There exists faster parsing algorithms for specialized grammars.

A context-free grammar is said to be a simple grammar (s-grammar) if all its productions are of the form:

$$A \rightarrow ax, \qquad A \in V, \; a \in T, x \in V^*$$

And any pair (A, a) occurs at most once.

Faster Parsers

S-grammar Example: $S \rightarrow aS \mid bSS \mid c$

- Looking at exhaustive search for this grammar, at each step there is only one choice to follow.

  w = abcc

  S => aS => abSS => abcS => abcc

- Total steps for parsing string w: |w|