Context Free Grammar

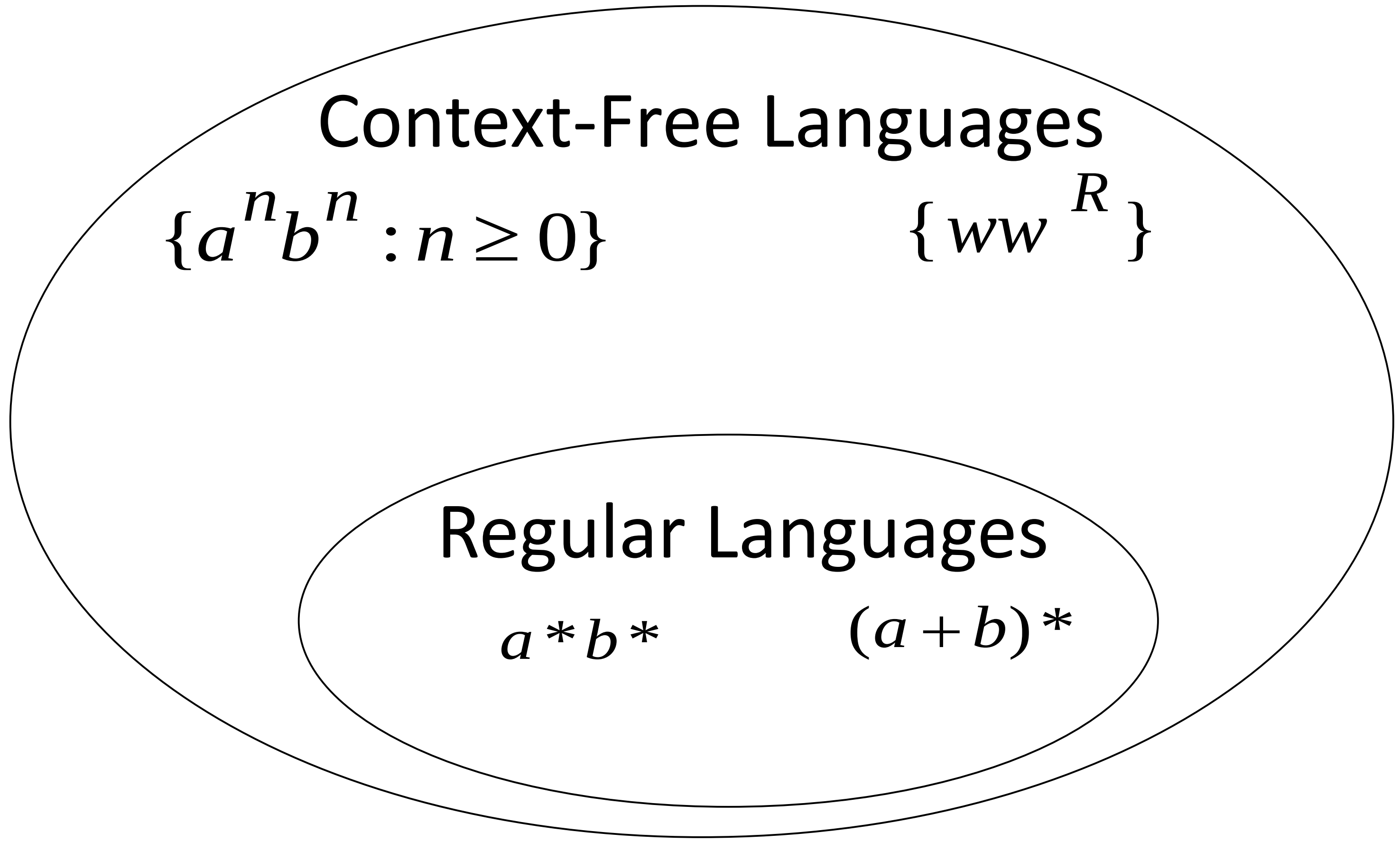
Dr. Mousumi Dutt

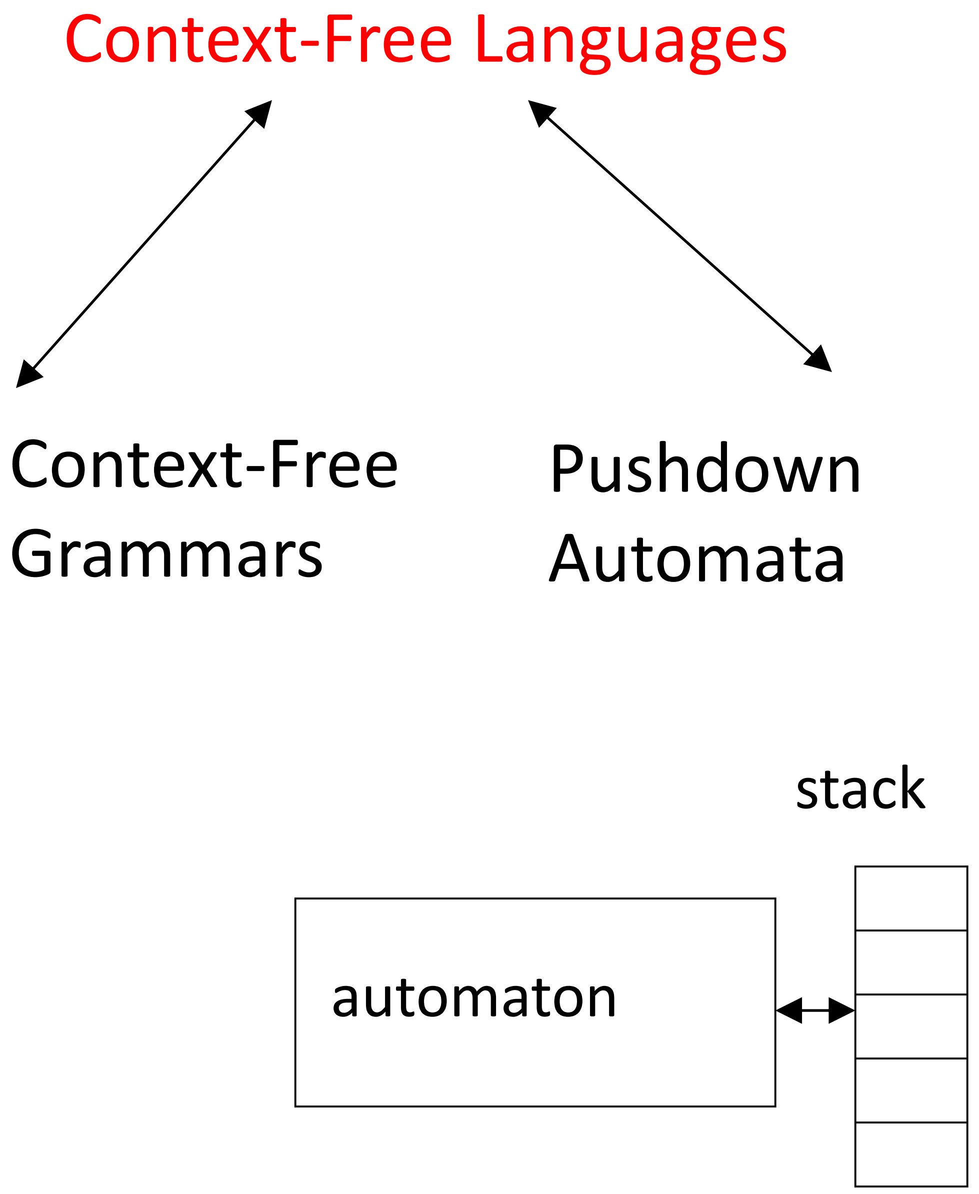
Module 3

Introduction: CFG

- The pumping lemma showed there are languages that are not regular
  - There are many classes “larger” than that of regular languages
  - One of these classes are called “Context Free” languages
- Described by Context-Free Grammars (CFG)
  - Why named context-free?
  - Property that we can substitute strings for variables regardless of context (implies context sensitive languages exist)
- CFG’s are useful in many applications
  - Describing syntax of programming languages
  - Parsing
  - Structure of documents, e.g.XML
- Analogy of the day:
  - DFA: Regular Expression as Pushdown Automata : CFG

Introduction: CFL





Introduction: CFL

- The class of context-free languages generalizes over the class of regular languages, i.e., every regular language is a context-free language.
- The reverse of this is not true, i.e., every context-free language is not necessarily regular. For example, as we will see $\{0^k 1^k \mid k \geq 0\}$ is context-free but not regular.
- Many issues and questions we asked for regular languages will be the same for context-free languages:

  Machine model – PDA (Push-Down Automata)

  Descriptor – CFG (Context-Free Grammar)

  Pumping lemma for context-free languages (and find CFL's limit)

  Closure of context-free languages with respect to various operations

  Algorithms and conditions for finiteness or emptiness
- Some analogies don't hold, e.g., non-determinism in a PDA makes a difference and, in particular, deterministic PDAs define a subset of the context-free languages.
- We will only talk on non-deterministic PDA here.

Introduction: CFL

Context-free languages allow us to describe nonregular languages like $\{ 0^n1^n \mid n \geq 0\}$

General idea: CFLs are languages that can be recognized by automata that have one single stack:

$\{ 0^n1^n \mid n \geq 0\}$ is a CFL

$\{ 0^n1^n0^n \mid n \geq 0\}$ is not a CFL

Which simple machine produces the nonregular language $\{ 0^n1^n \mid n \in \mathbb{N} \}$?
Start symbol S with rewrite rules:

1) $S \rightarrow 0S1$
2) $S \rightarrow$ "stop"

S *yields* 0^n1^n according to
$S \rightarrow 0S1 \rightarrow 00S11 \rightarrow \ldots \rightarrow 0^nS1^n \rightarrow 0^n1^n$

CFG Example

- Language of palindromes
  - We can easily show using the pumping lemma that the language L = { w | w = w^R } is not regular.
  - However, we can describe this language by the following context-free grammar over the alphabet {0,1}:

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

Inductive definition

More compactly: $P \rightarrow \varepsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1$

Grammars

- Grammars express languages

- Example: the English language grammar

$$\langle sentence \rangle \to \langle noun\_phrase \rangle \langle predicate \rangle$$

$$\langle noun\_phrase \rangle \to \langle article \rangle \langle noun \rangle$$

$$\langle predicate \rangle \to \langle verb \rangle$$

$$\langle article \rangle \rightarrow a$$

$$\langle article \rangle \rightarrow the$$

$$\langle noun \rangle \rightarrow cat$$

$$\langle noun \rangle \rightarrow dog$$

$$\langle verb \rangle \rightarrow runs$$

$$\langle verb \rangle \rightarrow sleeps$$

- Derivation of string “the dog walks”:

$$
\begin{aligned}
\langle sentence \rangle &\Rightarrow \langle noun\_phrase \rangle \langle predicate \rangle \\
&\Rightarrow \langle noun\_phrase \rangle \langle verb \rangle \\
&\Rightarrow \langle article \rangle \langle noun \rangle \langle verb \rangle \\
&\Rightarrow the \ \langle noun \rangle \langle verb \rangle \\
&\Rightarrow the \ \ dog \ \langle verb \rangle \\
&\Rightarrow the \ \ dog \ \ sleeps
\end{aligned}
$$

- Derivation of string “a cat runs”:

$$\begin{aligned}
\langle sentence \rangle &\Rightarrow \langle noun\_phrase \rangle \langle predicate \rangle \\
&\Rightarrow \langle noun\_phrase \rangle \langle verb \rangle \\
&\Rightarrow \langle article \rangle \langle noun \rangle \langle verb \rangle \\
&\Rightarrow a \ \langle noun \rangle \langle verb \rangle \\
&\Rightarrow a \ cat \ \langle verb \rangle \\
&\Rightarrow a \ cat \ runs
\end{aligned}$$

- Language of the grammar:

L = { “a cat runs”,
“a cat sleeps”,
“the cat runs”,
“the cat sleeps”,
“a dog runs”,
“a dog sleeps”,
“the dog runs”,
“the dog sleeps” }

Productions

$$\langle noun \rangle \rightarrow \overbrace{cat}^{\text{Sequence of Terminals (symbols)}}$$

$$\langle sentence \rangle \rightarrow \underbrace{\langle noun\_phrase \rangle \langle predicate \rangle}_{\text{Sequence of Variables}}$$

Variables

Another Example

Sequence of
terminals and variables

Grammar:

$$S \rightarrow \overbrace{aSb}$$

$$S \rightarrow \lambda$$

Variable

The right side
may be λ

- Grammar:

$$S \to aSb$$
$$S \to \lambda$$

- Derivation of string ab:

$$\begin{tikzcd}
S & \Rightarrow & aSb & \Rightarrow & ab \\
 & S \to aSb \arrow[u] & & S \to \lambda \arrow[u] &
\end{tikzcd}$$

- Grammar:

$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

- Derivation of string:

$$aabb$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

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

Grammar:

$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

Other derivations:

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

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

$$\Rightarrow aaaaSbbbb \Rightarrow aaaabbbb$$

Grammar:

$$S \to aSb$$

$$S \to \lambda$$

Language of the grammar:

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

A Convenient Notation

- We write:

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

for zero or more derivation steps

- Instead of:

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

In general we write: $w_1 \overset{*}{\Rightarrow} w_n$

If: $w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \cdots \Rightarrow w_n$

in zero or more derivation steps

Trivially: $w \overset{*}{\Rightarrow} w$

Example Grammar

$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

Possible Derivations

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

$$S \overset{*}{\Rightarrow} a\,b$$

$$S \overset{*}{\Rightarrow} a\,a\,a\,b\,b\,b$$

$$S \overset{*}{\Rightarrow} aaSbb \overset{*}{\Rightarrow} aaaaaSbbbbb \quad b$$

Another convenient notation:

$$\begin{array}{l} S \to aSb \\ S \to \lambda \end{array} \Longrightarrow S \to aSb \mid \lambda$$

$$\begin{array}{l} \langle article \rangle \to a \\ \langle article \rangle \to the \end{array} \Longrightarrow \langle article \rangle \to a \mid the$$

Formal Definition of Context Free Grammar

- There is a finite set of symbols that form the strings, i.e. there is a finite alphabet. The alphabet symbols are called **terminals** (think of a parse tree)
- There is a finite set of **variables**, sometimes called non-terminals or syntactic categories. Each variable represents a language (i.e. a set of strings).
  - In the palindrome example, the only variable is P.
- One of the variables is the **start symbol**. Other variables may exist to help define the language.
- There is a finite set of **productions** or production rules that represent the recursive definition of the language. Each production is defined:

1. Has a single variable that is being defined to the left of the production
2. Has the production symbol $\rightarrow$
3. Has a string of zero or more terminals or variables, called the body of the production. To form strings we can substitute each variable's production in for the body where it appears.

Formal Definition of Context Free Grammar

- A CFG G may then be represented by these four components, denoted G=(V,T,P,S)

V - A finite set of variables or *non-terminals*

T - A finite set of *terminals* (*V* and *T* do not intersect: *do not use same symbols*)

This is our ∑

P - A finite set of *productions*, each of the form A –> α, where *A* is in *V* and

α is in (V ∪ T)*

Note that *α* may be *ε*

S - A starting non-terminal (*S* is in *V*)

1. E→I // Expression is an identifier
2. E→E+E // Add two expressions
3. E→E*E // Multiply two expressions
4. E→(E) // Add parenthesis
5. I→ L // Identifier is a Letter
6. I→ ID // Identifier + Digit
7. I→ IL // Identifier + Letter
8. D → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 // Digits
9. L → a | b | c | … A | B | … Z // Letters

Note Identifiers are regular; could describe as (letter)(letter + digit)*

Example of Context Free Grammar

- **Example CFG for** $\{0^k1^k \mid k \geq 0\}$**:**

  G = ({S}, {0, 1}, P, S)   *// Remember: G = (V, T, P, S)*

  P:

  (1) S –> 0S1   or just simply S –> 0S1 | ε

  (2) S –> ε

- **Example Derivations:**

  S => 0S1   (1)   S => ε   (2)

  => 01   (2)

  S => 0S1   (1)   => 00S11   (1)

  => 000S111   (1)   => 000111   (2)

- Note that G "generates" the language $\{0^k1^k \mid k \geq 0\}$

Example of Context Free Grammar

- **Example CFG for ?:**

G = ({A, B, C, S}, {a, b, c}, P, S)

P:

(1)	S –> ABC	
(2)	A –> aA	A –> aA | ε
(3)	A –> ε	
(4)	B –> bB	B –> bB | ε
(5)	B –> ε	
(6)	C –> cC	C –> cC | ε
(7)	C –> ε	

Example Derivations:

S => ABC	(1)
=> BC	(3)
=> C	(5)
=> ε	(7)

S => ABC	(1)
=> aABC	(2)
=> aaABC	(2)
=> aaBC	(3)
=> aabBC	(4)
=> aabC	(5)
=> aabcC	(6)
=> aabc	(7)

Note that G generates the language a*b*c*

Another Example

Context-free grammar : G

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

Example derivations:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba$$

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow abaaba$$

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

Palindromes of even length

Another Example

Context-free grammar : G

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

Example derivations:

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow ab$$

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \Rightarrow abab$$

$$L(G) = \{w : n_a(w) = n_b(w),$$

$$\text{and } n_a(v) \geq n_b(v)$$

$$\text{in any prefix } v\}$$

Describes matched parentheses: () ((( ))) (( ))

$a = ($, $\quad b =)$

CFGs & CFLs

$$\{\mathbf{a}^m\, \mathbf{b}^n\, \mathbf{c}^{m+n} \mid m,n \geq 0\}$$

? ?

Rewrite as $\{\mathbf{a}^m\, \mathbf{b}^n\, \mathbf{c}^n\, \mathbf{c}^m \mid m,n \geq 0\}$:

$S \to S' \mid \mathbf{a}\, S\, \mathbf{c}$

$S' \to \varepsilon \mid \mathbf{b}\, S'\, \mathbf{c}$

CFGs & CFLs

$\{\mathbf{a}^n\ \mathbf{b}^n\ \mathbf{c}^n \mid n \geq 0\}$

Can’t be done; CFL pumping lemma later.

Intuition: Can count to n, then can count down from n, but forgetting n.

- I.e., a stack as a counter.
- Will see this when using a machine corresponding to CFGs.

Definitions and Observations

Let G = (V, T, P, S) be a CFG.

Observation: "–>" forms a relation on V and $(V \cup T)^*$

Definition: Let *A* be in *V*, and *B* be in $(V \cup T)^*$, A –> B be in *P*, and let *α* and *β* be in $(V \cup T)^*$. Then:

$$\alpha A\beta \Rightarrow \alpha B\beta$$

In words, αAβ *directly derives* αBβ, or in other words αBβ follows from αAβ by the application of exactly one production from *P*.

Observation: "=>" forms a relation on $(V \cup T)^*$ and $(V \cup T)^*$.

Definitions and Observations

- **Definition:** Suppose that $\alpha_1, \alpha_2, \ldots, \alpha_m$ are in $(V \cup T)^*$, $m \geq 1$, and

$$\begin{array}{l} \alpha_1 \Rightarrow \alpha_2 \\ \alpha_2 \Rightarrow \alpha_3 \\ \quad \vdots \\ \alpha_{m-1} \Rightarrow \alpha_m \end{array}$$

  Then $\alpha_1 \Rightarrow^* \alpha_m$

  In words, α_m follows from α_1 by the application of *zero or more* productions. Note that: $\alpha \Rightarrow^* \alpha$.
- **Observation: "=>*"** forms a relation on $(V \cup T)^*$ and $(V \cup T)^*$.
- **Definition:** Let α be in $(V \cup T)^*$. Then α is a *sentential form* if and only if $S \Rightarrow^* \alpha$.
- **Definition:** Let G = (V, T, P, S) be a context-free grammar. Then the *language generated* by G, denoted L(G), is the set:

$$\{w \mid w \text{ is in } T^* \text{ and } S \Rightarrow^* w\}$$

- **Definition:** Let *L* be a language. Then *L* is a *context-free language* if and only if there exists a context-free grammar *G* such that L = L(G).
- **Definition:** Let G_1 and G_2 be context-free grammars. Then *G1* and *G2* are *equivalent* if and only if $L(G_1) = L(G_2)$.

- **Theorem:** Let *L* be a regular language. Then *L* is a context-free language. (or, RL $\subseteq$CFL)
- **Proof:** (by induction)

We will prove that if *r* is a regular expression then there exists a CFG *G* such that L(r) = L(G). The proof will be by induction on the number of operators in *r*.

Basis: Op(r) = 0

Then *r* is either Ø, ε, or ***a***, for some symbol ***a*** in Σ.

For Ø:

Let G = ({S}, {}, P, S) where P = {}

For ε:

Let G = ({S}, {}, P, S) where P = {S –> ε}

For **a**:

Let G = ({S}, {a}, P, S) where P = {S –> **a**}

Inductive Hypothesis:

Suppose that for any regular expression r, where $0 \leq op(r) \leq k$, that there exists a CFG G such that L(r) = L(G), for some k>=0.

Inductive Step:

Let r be a regular expression with op(r)=k+1. Then $r = r_1 + r_2$, $r = r_1r_2$ or $r = r_1^*$.

Case 1) $r = r_1 + r_2$

Since r has k+1 operators, one of which is +, it follows that r_1 and r_2 have at most k operators. From the inductive hypothesis it follows that there exist CFGs $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_2, P_2, S_2)$ such that $L(r_1) = L(G_1)$ and $L(r_2) = L(G_2)$. Assume without loss of generality that V_1 and V_2 have no non-terminals in common, and construct a grammar G = (V, T, P, S) where:

$V = V_1 \cup V_2 \cup \{S\}$

$T = T_1 \cup T_2$

$P = P_1 \cup P_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}$

Clearly, L(r) = L(G).

Case 2) $r = r_1r_2$

Let $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_2, P_2, S_2)$ be as in Case 1, and construct a grammar $G = (V, T, P, S)$ where:

$V = V_1 \cup V_2 \cup \{S\}$

$T = T_1 \cup T_2$

$P = P_1 \cup P_2$ U $\{S \rightarrow S_1S_2\}$

Clearly, $L(r) = L(G)$.

Case 3) $r = (r_1)^*$

Let $G_1 = (V_1, T_1, P_1, S_1)$ be a CFG such that $L(r_1) = L(G_1)$ and construct a grammar $G = (V, T, P, S)$ where:

$V = V_1 \cup \{S\}$

$T = T_1$

$P = P_1 \cup \{S \rightarrow S_1S, S \rightarrow \varepsilon\}$

Clearly, $L(r) = L(G)$.

- The preceding theorem is constructive, in the sense that it shows how to construct a CFG from a given regular expression.
- **Example #1:**

  $r = a^*b^*$

  $r = r_1r_2$

  $r_1 = r_3^*$

  $r_3 = a$

  $r_2 = r_4^*$

  $r_4 = b$

- **Example #1:** a*b*

$r_4 = b$	$S_1 \rightarrow b$
$r_3 = a$	$S_2 \rightarrow a$
$r_2 = r_4^*$	$S_3 \rightarrow S_1S_3$
	$S_3 \rightarrow \varepsilon$
$r_1 = r_3^*$	$S_4 \rightarrow S_2S_4$
	$S_4 \rightarrow \varepsilon$
$r = r_1r_2$	$S_5 \rightarrow S_4S_3$

- **Example #2:**

$r = (0+1)^*01$

$r = r_1r_2$

$r_1 = r_3^*$

$r_3 = (r_4+r_5)$

$r_4 = 0$

$r_5 = 1$

$r_2 = r_6r_7$

$r_6 = 0$

$r_7 = 1$

- **Example #2:** (0+1)*01

$r_7 = 1$	$S_1 \rightarrow 1$
$r_6 = 0$	$S_2 \rightarrow 0$
$r_2 = r_6r_7$	$S_3 \rightarrow S_2S_1$
$r_5 = 1$	$S_4 \rightarrow 1$
$r_4 = 0$	$S_5 \rightarrow 0$
$r_3 = (r_4+r_5)$	$S_6 \rightarrow S_4,\ S_6 \rightarrow S_5$
$r_1 = r_3^*$	$S_7 \rightarrow S_6S_7$
	$S_7 \rightarrow \varepsilon$
$r = r_1r_2$	$S_8 \rightarrow S_7S_3$

- **Definition:** A CFG is a <u>regular grammar</u> if each rule is of the following form:
  - A –> a
  - A –> aB
  - A –> ε

  where *A* and *B* are in *V*, and *a* is in *T*
- **Theorem:** A language *L* is a regular language iff there exists a regular grammar *G* such that L = L(G).
- **Proof:** Exercise. •Develop translation fromRegular form -> DFA; and

  DFA -> regular grammar]
- **Observation:** The grammar S –> 0S1 | ε is not a regular grammar.
- **Observation:** A language may have several CFGs, some regular, some not (The fact that the preceding grammar is not regular does not in and of itself prove that 0^n1^n is not a regular language).

- **Definition:** Let G = (V, T, P, S) be a CFG. A tree is a derivation (or parse) tree if:

  - Every vertex has a label from $V \cup T \cup \{\varepsilon\}$
  - The label of the root is S
  - If a vertex with label A has children with labels $X_1, X_2, \ldots, X_n$, from left to right, then

    $$A \rightarrow X_1, X_2, \ldots, X_n$$

    must be a production in P
  - If a vertex has label ε, then that vertex is a leaf and the only child of its' parent

- More Generally, a derivation tree can be defined with any non-terminal as the root.

Sample CFG

1. E→I // Expression is an identifier
2. E→E+E // Add two expressions
3. E→E*E // Multiply two expressions
4. E→(E) // Add parenthesis
5. I→ L // Identifier is a Letter
6. I→ ID // Identifier + Digit
7. I→ IL // Identifier + Letter
8. D → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 // Digits
9. L → a | b | c | … A | B | … Z // Letters

Note Identifiers are regular; could describe as (letter)(letter + digit)*

Recursive Inference

- The process of coming up with strings that satisfy individual productions and then concatenating them together according to more general rules is called *recursive inference*.
- This is a bottom-up process
- For example, parsing the identifier "r5"
  - Rule 8 tells us that D → 5
  - Rule 9 tells us that L → r
  - Rule 5 tells us that I→L so I→r
  - Apply recursive inference using rule 6 for I→ID and get
    - I → rD.
    - Use D→5 to get I→r5.
  - Finally, we know from rule 1 that E→I, so r5 is also an expression.

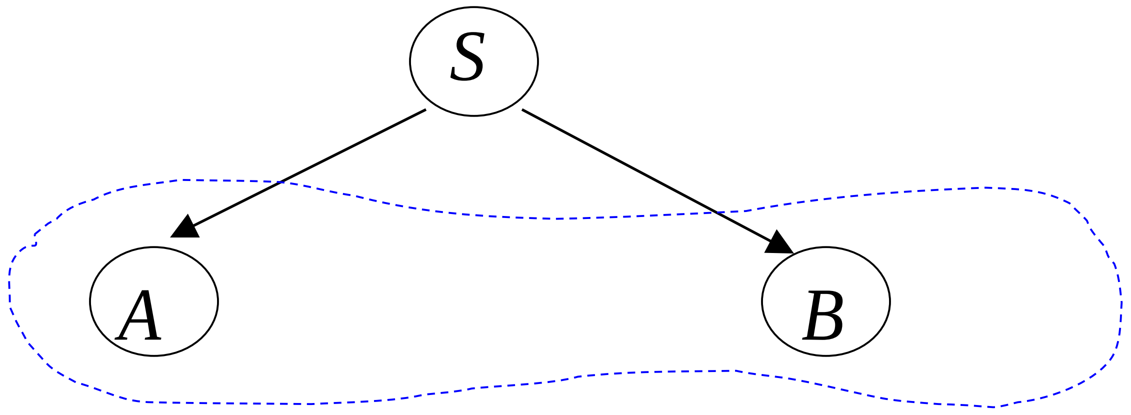
Recursive Inference Exercise

- Show the recursive inference for arriving at *(x+y1)*y* is an expression

1. E→I
2. E→E+E
3. E→E*E
4. E→(E)
5. I→ L
6. I→ ID
7. I→ IL
8. D → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
9. L → a | b | c | … A | B | … Z

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

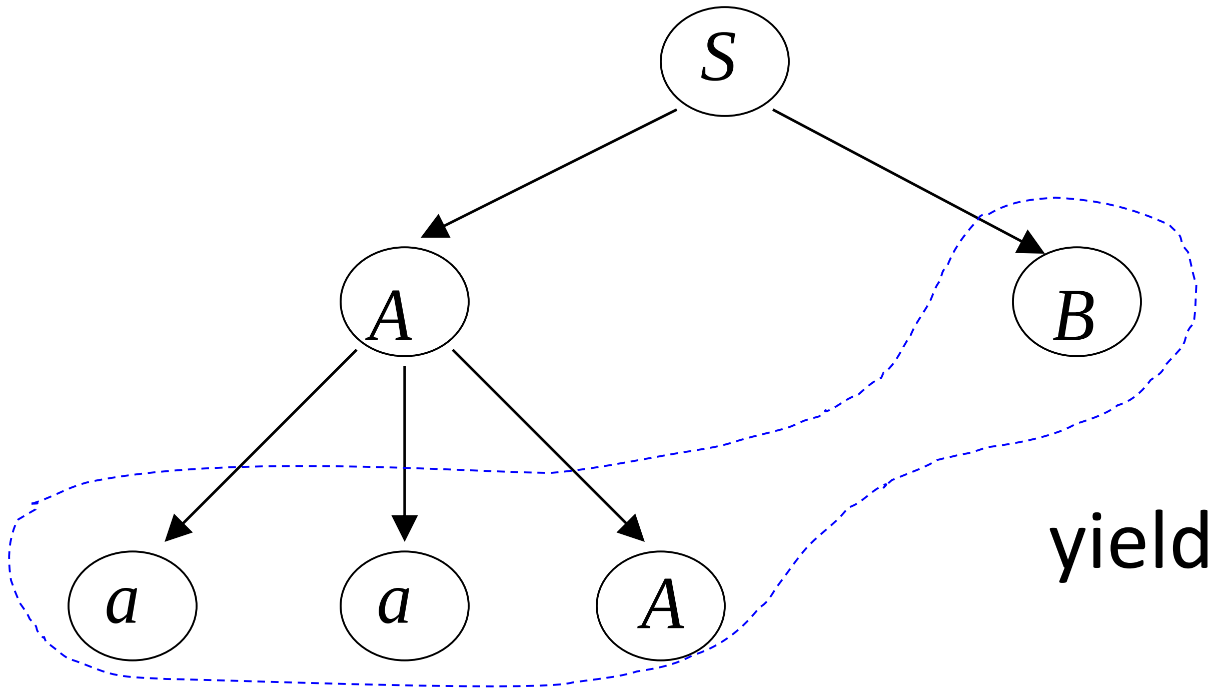
$$S \Rightarrow AB$$



yield AB

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

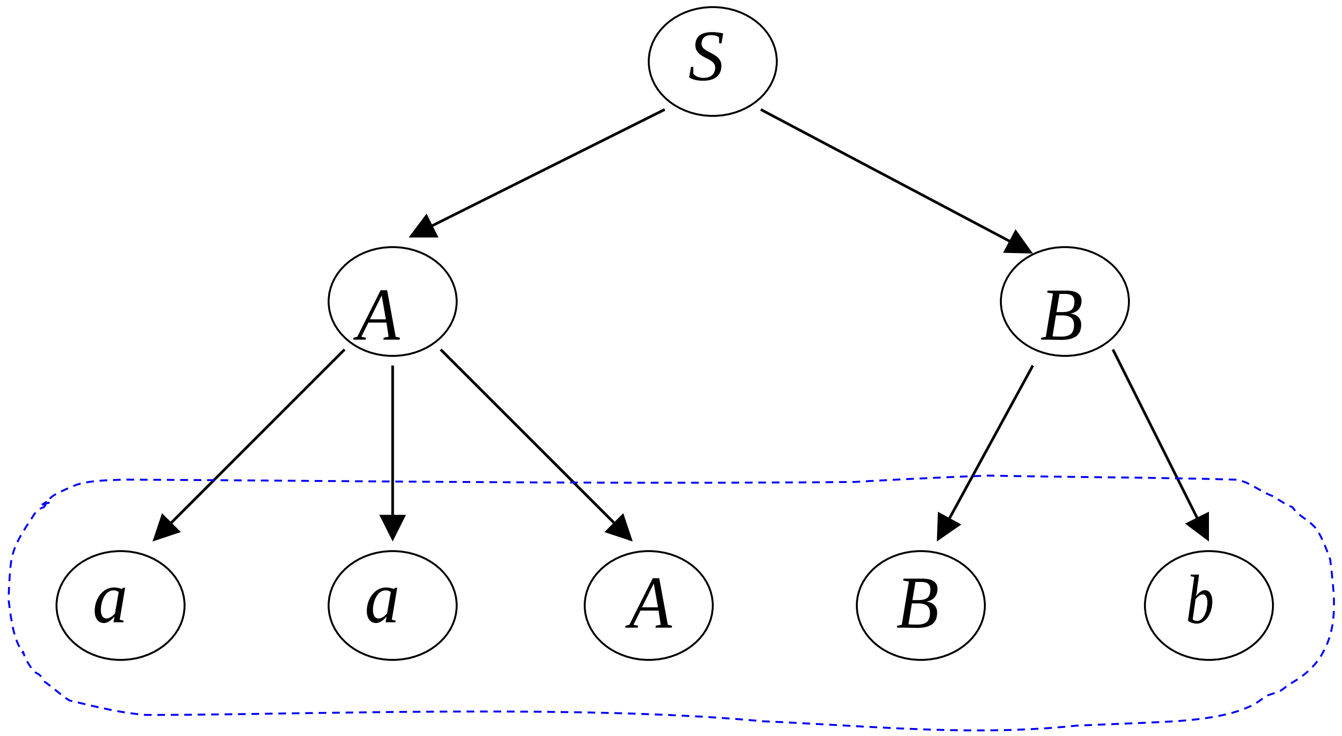
$$S \Rightarrow AB \Rightarrow aaAB$$



yield $aaAB$

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

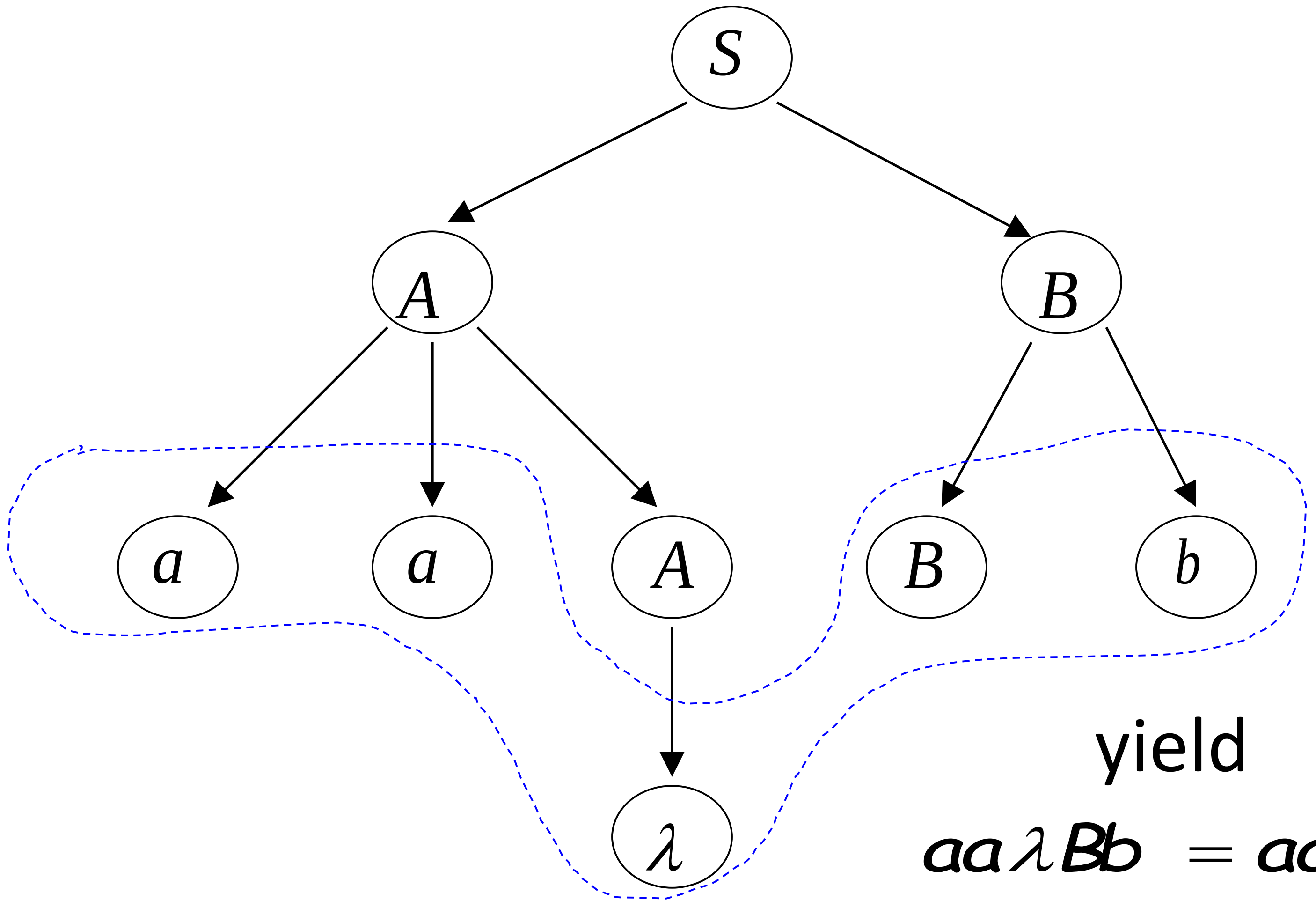
$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb$$



yield $aaABb$

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

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



yield

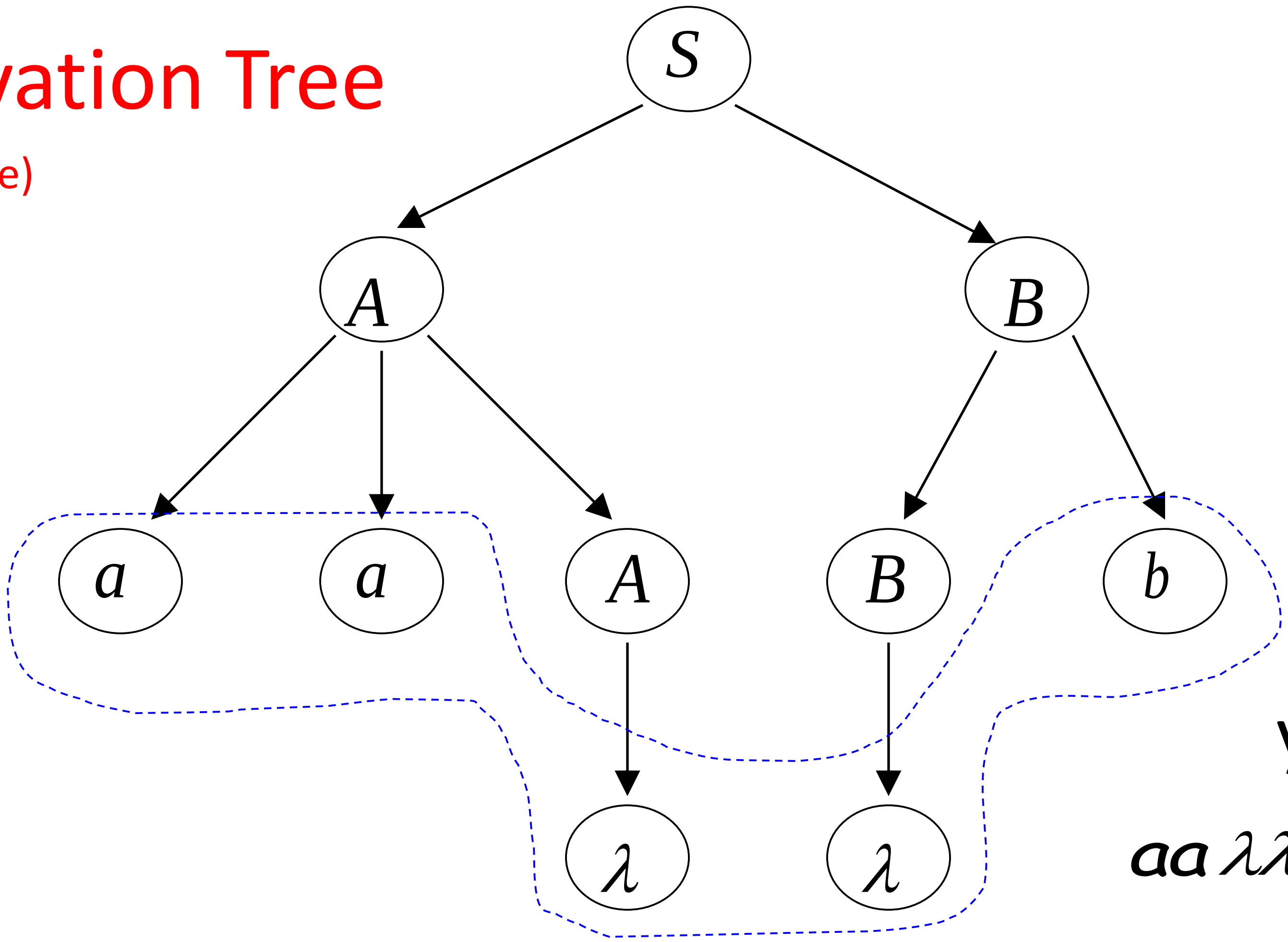
$$aa\lambda Bb = aaBb$$

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

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

Derivation Tree

(parse tree)



yield

$aa\lambda\lambda b = aab$

Sometimes, derivation order doesn't matter

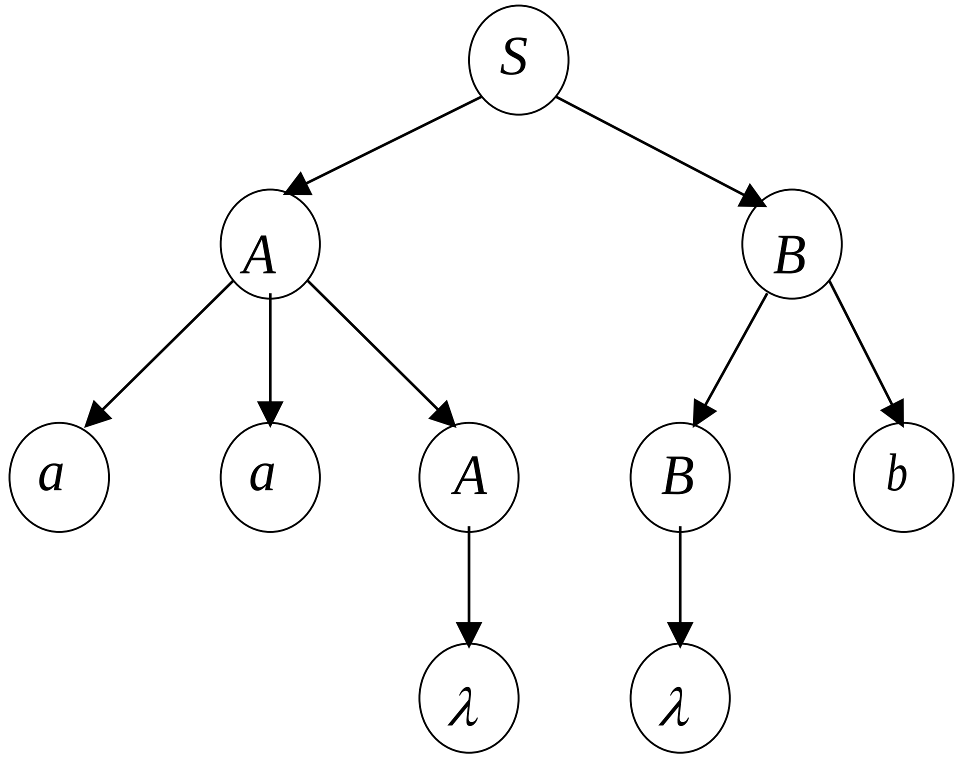
Leftmost derivation:

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

Rightmost derivation:

$$S \Rightarrow AB \Rightarrow ABb \Rightarrow Ab \Rightarrow aaAb \Rightarrow aab$$

Give same derivation tree



Example:

S –> AB

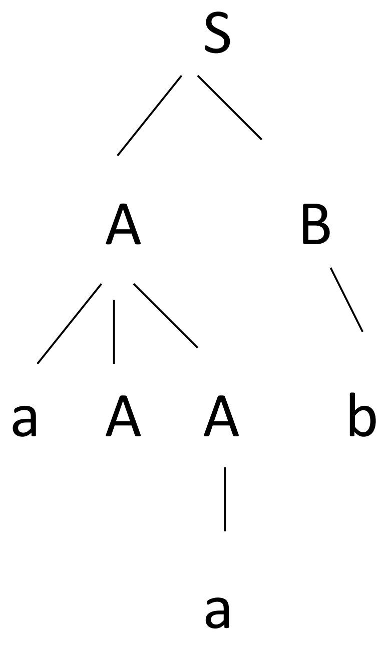
A –> aAA

A –> aA

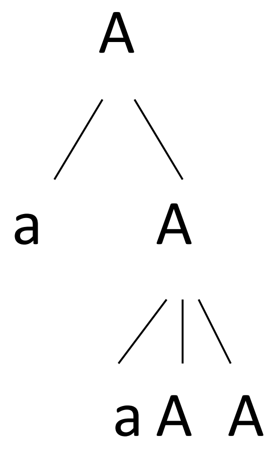
A –> a

B –> bB

B –> b



yield = aAab

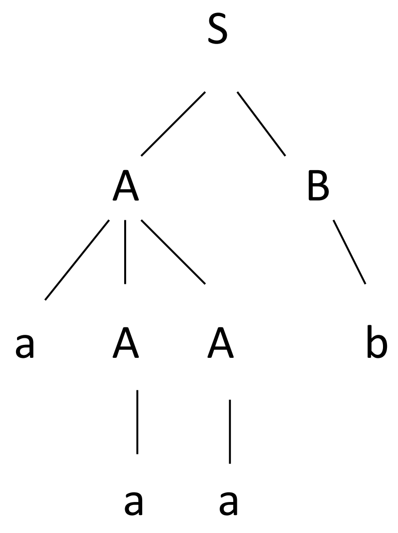


yield = aaAA

- **Notes:**
  - Root can be any non-terminal
  - Leaf nodes can be terminals or non-terminals
  - A derivation tree with root S shows the productions used to obtain a sentential form

- **Observation:** Every derivation corresponds to one derivation tree.

S => AB
=> aAAB
=> aaAB
=> aaaB
=> aaab



Rules:
S –> AB
A –> aAA
A –> aA
A –> a
B –> bB
B –> b

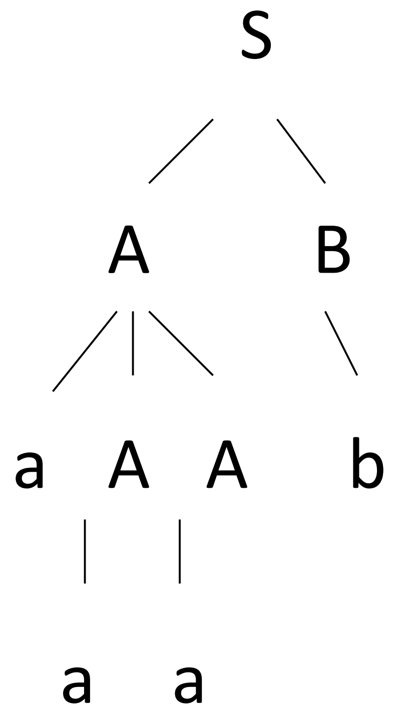
- **Observation:** Every derivation tree corresponds to one or more derivations.

leftmost:	*rightmost:*	*mixed:*
S => AB	S => AB	S => AB
=> aAAB	=> Ab	=> Ab
=> aaAB	=> aAAb	=> aAAb
=> aaaB	=>aAab	=> aaAb
=> aaab	=> aaab	=> aaab

- **Definition:** A derivation is *leftmost (rightmost)* if at each step in the derivation a production is applied to the leftmost (rightmost) non-terminal in the sentential form.
  - The first derivation above is leftmost, second is rightmost, the third is neither.

- **Observation:** Every derivation tree corresponds to exactly one leftmost (and rightmost) derivation.

```
S   => AB
    => aAAB
    => aaAB
    => aaaB
    => aaab
```



- **Observation:** Let G be a CFG. Then there may exist a string *x* in L(G) that has more than 1 leftmost (or rightmost) derivation. Such a string will also have more than 1 derivation tree.

- **Example:** Consider the string *aaab* and the preceding grammar.

S –> AB

A –> aAA

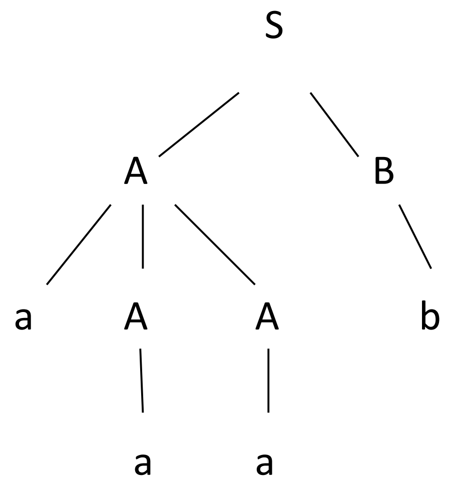
A –> aA

A –> a

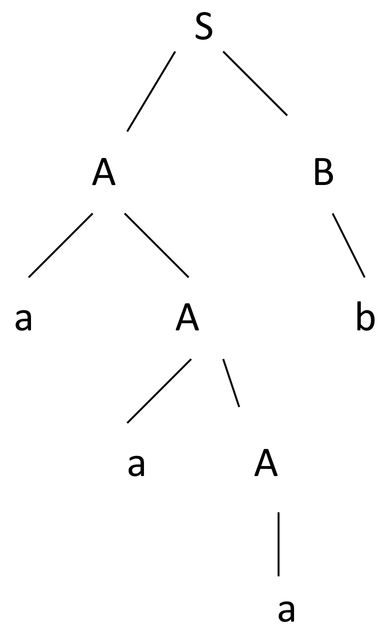
B –> bB

B –> b

S => AB
=> aAAB
=> aaAB
=> aaaB
=> aaab



S => AB
=> aAB
=> aaAB
=> aaaB
=> aaab



- The string has two left-most derivations, and therefore has two distinct parse trees.

- **Definition:** Let G be a CFG. Then G is said to be ambiguous if there exists an x in L(G) with >1 leftmost derivations. Equivalently, G is said to be ambiguous if there exists an x in L(G) with >1 parse trees, or >1 rightmost derivations.

- **Note:** Given a CFL L, there may be more than one CFG G with L = L(G). Some ambiguous and some not.

- **Definition:** Let L be a CFL. If every CFG G with L = L(G) is ambiguous, then L is inherently ambiguous.

- An ambiguous Grammar:

  E -> I   ∑ ={0,…,9, +, *, (,)}

  E -> E + E

  E -> E * E

  E -> (E)

  I -> ε | 0 | 1 | … | 9

A leftmost derivation
E=>E*E
=>I*E
=>3*E+E
=>3*I+E
=>3*2+E
=>3*2+I
=>3*2+5

- A string: 3*2+5
- Two parse trees:

  * on top, & + on top

  & two left-most derivation:

Another leftmost derivation
E=>E+E
=>E*E+E
=>I*E+E
=>3*E+E
=>3*I+E
=>3*2+I
=>3*2+5

E -> I **∑ ={0,…,9, +, *, (,)}**

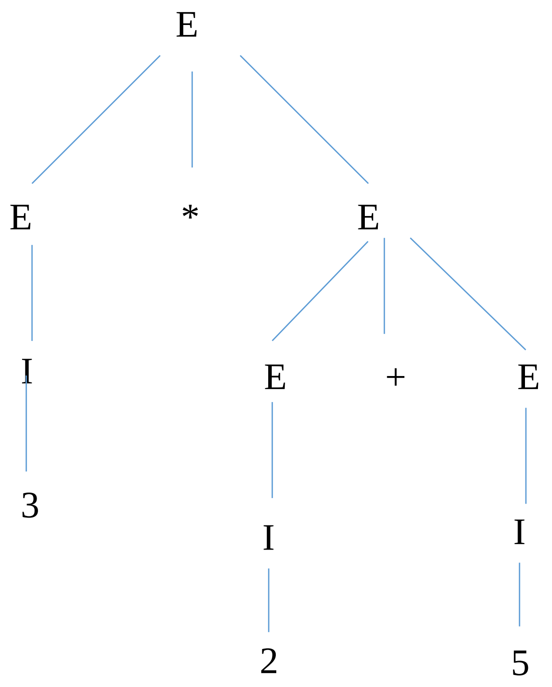
E -> E + E

E -> E * E

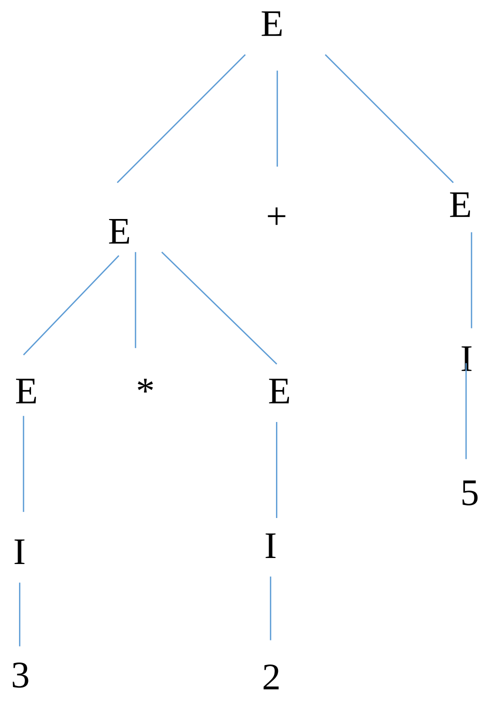
E -> (E)

I -> ε | 0 | 1 | … | 9

E=>E*E
=>I*E
=>3*E+E
=>3*I+E
=>3*2+E
=>3*2+I
=>3*2+5



Another leftmost derivation
E=>E+E
=>E*E+E
=>I*E+E
=>3*E+E
=>3*I+E
=>3*2+I
=>3*2+5



Removing Ambiguity

- No algorithm can tell us if an **arbitrary** CFG is ambiguous in the first place
    - Halting / Post Correspondence Problem
- Why care?
    - Ambiguity can be a problem in things like programming languages where we want agreement between the programmer and compiler over what happens
- Solutions
    - Apply precedence
    - e.g. Instead of: $E \rightarrow E+E \mid E*E$
    - Use: $E \rightarrow T \mid E + T$, $T \rightarrow F \mid T * F$
        - This rule says we apply + rule before the * rule (which means we multiply first before adding)

- Disambiguation of the Grammar:

∑ ={0,,,9, +, *, (,)}

E -> T | E + T   *// This T is a non-terminal, do not confuse with ∑*

T -> F | T * F

F -> I | (E)

I -> ε | 0 | 1 | … | 9

Ambiguous grammar:

E -> I

E -> E + E

E -> E * E

E -> (E)

I -> ε | 0 | 1 | … | 9

- A string: 3*2+5
- Only one parse tree & one left-most derivation now:

  + on top: *TRY PARSING THE EXPRESSION NOW*

Two different derivation trees
may cause problems in applications which
use the derivation trees:

- Evaluating expressions
- In general, in compilers for programming languages

- A language may be *Inherently ambiguous*:

L =$\{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$

- An ambiguous grammar:

S -> AB | C

A -> aAb | ab

B -> cBd | cd

C -> aCd | aDd

D -> bDc | bc

- Try the string: *aabbccdd*, two different derivation trees
- Grammar CANNOT be disambiguated for this (not showing the proof)

Rules:

S -> AB | C

A -> aAb | ab

B -> cBd | cd

C -> aCd | aDd

D -> bDc | bc

String *aabbccdd* belongs to two different parts of the language:

$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$

Derivation 1 of *aabbccdd*:

S => AB

=> aAbB

=> aabbB

=> aabb cBd

=> aabbccdd

Derivation 2 of *aabbccdd*:

S => C

=> aCd

=> aaDdd

=> aa bDc dd

=> aabbccdd

Another ambiguous grammar:

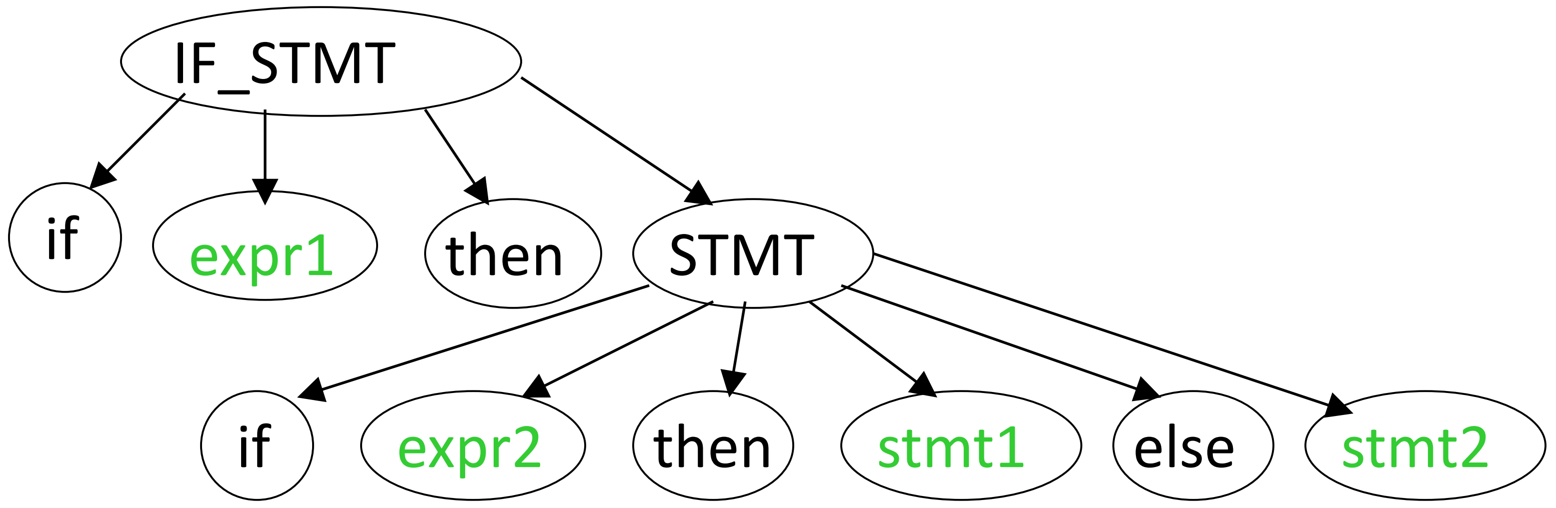
IF_STMT $\rightarrow$ if EXPR then STMT

$|$ if EXPR then STMT else STMT

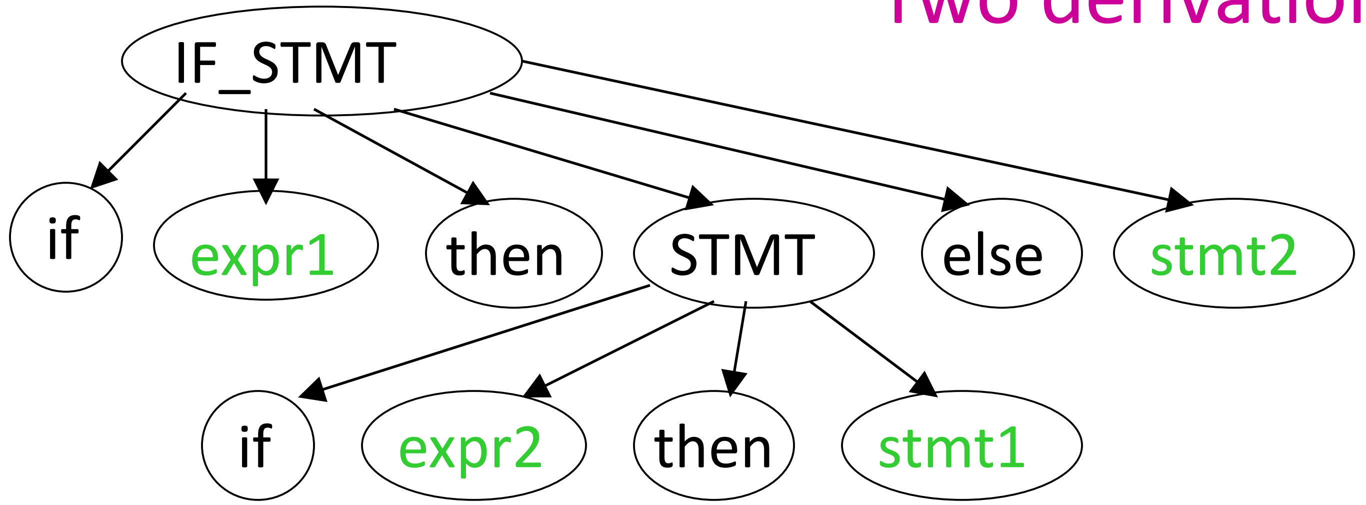
Variables Terminals

Very common piece of grammar
in programming languages

If expr1 then if expr2 then stmt1 else stmt2



Two derivation trees



In general, ambiguity is bad
and we want to remove it

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

But, in general we cannot do so

A successful example:

Ambiguous Grammar

$$\boxed{\begin{aligned} E &\to E + E \\ E &\to E * E \\ E &\to (E) \\ E &\to a \end{aligned}}$$

Equivalent

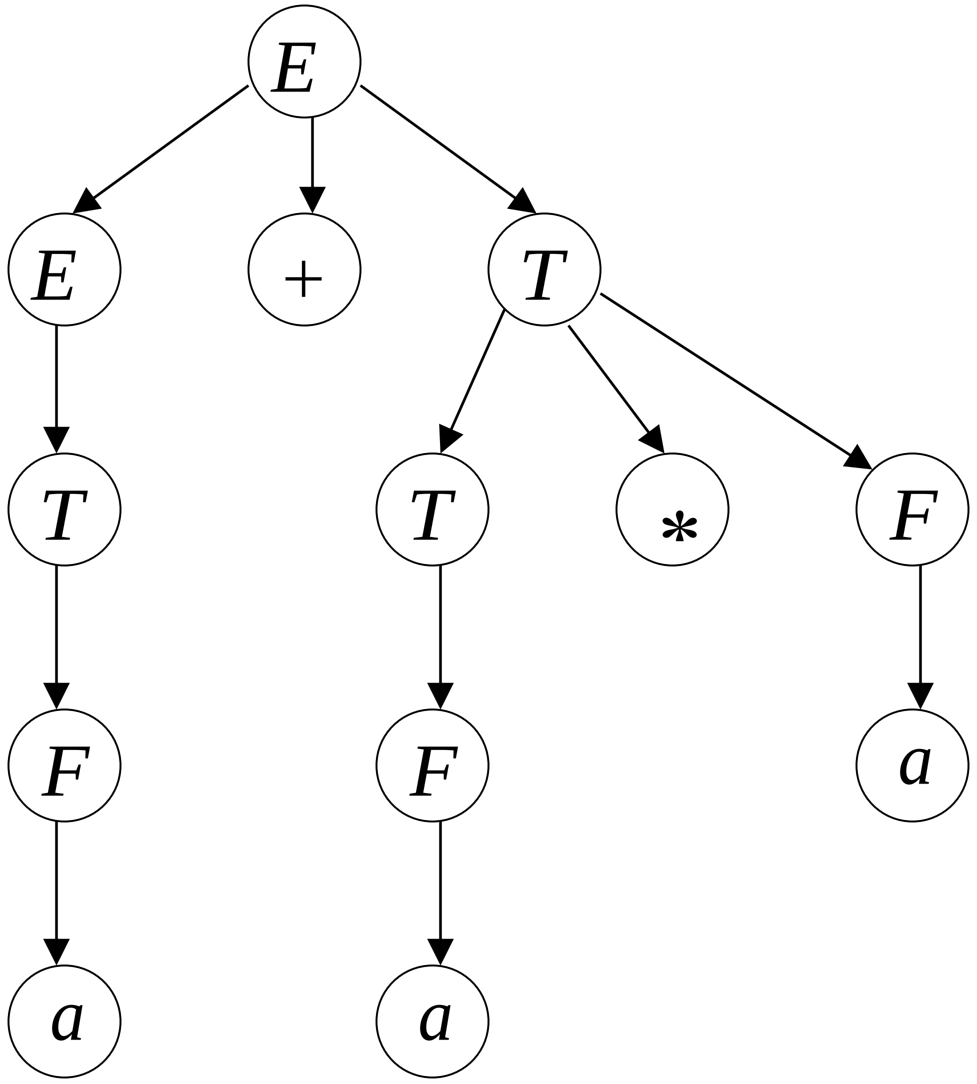
Non-Ambiguous Grammar

$$\boxed{\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid a \end{aligned}}$$

generates the same language

$$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T*F$$
$$\Rightarrow a+F*F \Rightarrow a+a*F \Rightarrow a+a*a$$

$$\boxed{\begin{aligned} E &\to E+T \mid T \\ T &\to T*F \mid F \\ F &\to (E) \mid a \end{aligned}}$$



Unique derivation tree for

$$a+a*a$$

An un-successful example:

$$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}$$

$$n, m \geq 0$$

L is inherently ambiguous:

every grammar that generates this language is ambiguous

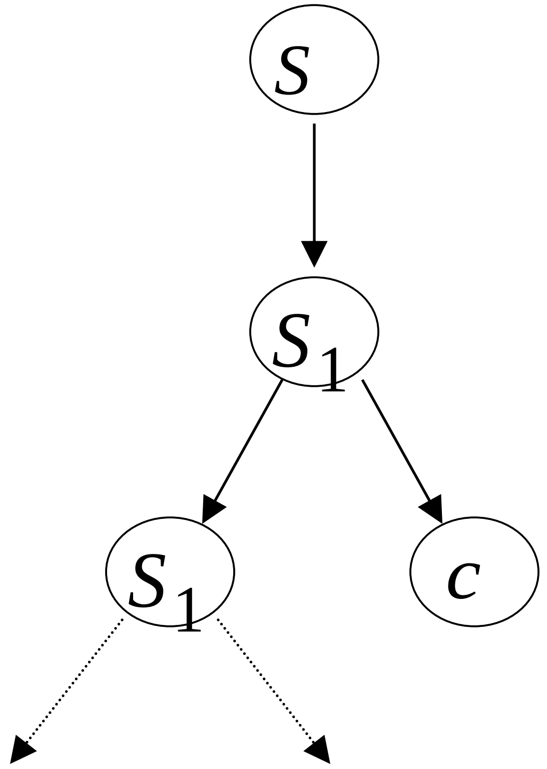
Example (ambiguous) grammar for : L

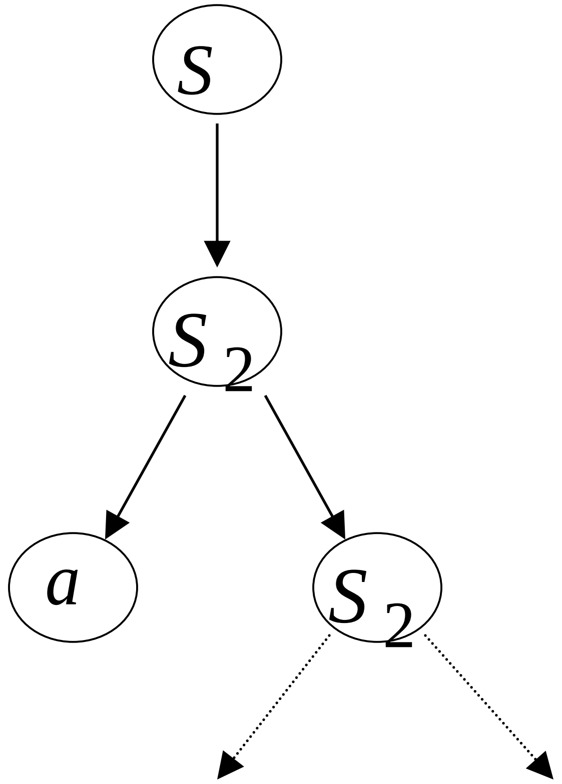
$$\begin{tikzcd}
L = \arrow[d] & \{a^n b^n c^m\} \cup \arrow[d] & \{a^n b^m c^m\} \arrow[d] \\
S \rightarrow S_1 \mid S_2 & S_1 \rightarrow S_1 c \mid A & S_2 \rightarrow a S_2 \mid B \\
 & A \rightarrow aAb \mid \lambda & B \rightarrow bBc \mid \lambda
\end{tikzcd}$$

The string $a^n b^n c^n \in L$

has always two different derivation trees (for any grammar)

For example





Potential Algorithmic Problems

- Potential algorithmic problems for context-free grammars:
  - Is L(G) empty?
  - Is L(G) finite?
  - Is L(G) infinite?
  - Is $L(G_1) = L(G_2)$?
  - Is G ambiguous?
  - Is L(G) inherently ambiguous?
  - Given ambiguous G, construct unambiguous G’ such that L(G) = L(G’)
  - Given G, is G “minimal?”

Disambiguation

What is a general algorithm?

? None exists! ?

There are CFLs that are *inherently ambiguous*

Every CFG for this language is ambiguous.

E.g., $\{\mathbf{a}^n\mathbf{b}^n\mathbf{c}^m\mathbf{d}^m \mid n\geq 1, m\geq 1\} \cup \{\mathbf{a}^n\mathbf{b}^m\mathbf{c}^m\mathbf{d}^n \mid n\geq 1, m\geq 1\}$.

So, can't necessarily eliminate ambiguity!

CFG Simplification

Can’t always eliminate ambiguity.

But, CFG simplification & restriction still useful theoretically & pragmatically.

- Simpler grammars are easier to understand.
- Simpler grammars can lead to faster parsing.
- Restricted forms useful for some parsing algorithms.
- Restricted forms can give you more knowledge about derivations.

CFG Simplification: Example

How can the following be simplified?

? ?

$S \rightarrow A\ B$

$S \rightarrow A\ C\ D$

$A \rightarrow A\ \mathbf{a}$

$A \rightarrow \mathbf{a}$

$A \rightarrow \mathbf{a}\ A$

$A \rightarrow \mathbf{a}$

$C \rightarrow \varepsilon$

$D \rightarrow \mathbf{d}\ D$

$D \rightarrow E$

$E \rightarrow \mathbf{e}\ A\ \mathbf{e}$

$F \rightarrow \mathbf{f}\ \mathbf{f}$

1) Delete: B useless because nothing derivable from B.

2) Delete either $A \rightarrow A\mathbf{a}$ or $A \rightarrow \mathbf{a}A$.

3) Delete one of the idential productions.

4) Delete & also replace $S \rightarrow ACD$ with $S \rightarrow AD$.

5) Replace with $D \rightarrow \mathbf{e}A\mathbf{e}$.

6) Delete: E useless after change #5.

7) Delete: F useless because not derivable from S.

CFG Simplification

Eliminate ambiguity.

Eliminate “useless” variables.

Eliminate ε-productions: $A \rightarrow \varepsilon$.

Eliminate unit productions: $A \rightarrow B$.

Eliminate redundant productions.

Trade left- & right-recursion.

Chomsky Normal Form

A context free grammar is said to be in **Chomsky Normal Form** if all productions are in the following form:

$$A \rightarrow BC$$
$$A \rightarrow \alpha$$

- A, B and C are non terminal symbols
- α is a terminal symbol

Preliminary Simplifications

There are three preliminary simplifications

1. Eliminate Useless Symbols
2. Eliminate ε productions
3. Eliminate unit productions

Elimination of useless symbols

- A variable is ***useful*** if it occurs in a derivation that begins with the start symbol *and* generates a terminal string.
    - Reachable from *S*
    - Derives terminal string

$$\boxed{S \Rightarrow_G^* uXv \quad \text{where } X \in V \quad u, v \in (V \cup \Sigma)^*}$$

$$\boxed{X \Rightarrow_G^* \omega \quad \text{where } \omega \in \Sigma^*}$$

- Construction of the set of variables that derive terminal string.
    - Bottom-up flow of information
        - Similar to the computation of nullable variables.
- Construction of the set of variables that are reachable
    - Top-down flow of information
        - Similar to the computation of chained variables.

Examples

$$\boxed{\begin{array}{l} S \to AB \mid \lambda \\ A \to aA \mid a \\ C \to cC \mid c \end{array}} \Rightarrow \boxed{\begin{array}{l} S \to \lambda \\ A \to aA \mid a \end{array}} \Rightarrow \boxed{S \to \lambda}$$

B does not derive terminal string;
C unreachable.

A unreachable.

$$\boxed{\begin{array}{l} S \to BD \\ B \to BD \mid B \\ D \to DB \mid D \end{array}} \Rightarrow \boxed{\text{Empty set of productions}}$$

"Non-termination"

Preliminary Simplifications

Eliminate ε Productions

- In a grammar ε productions are convenient but not essential
- If L has a CFG, then L – {ε} has a CFG

$$\boxed{A \overset{*}{\Longrightarrow} \varepsilon}$$

Nullable variable

Preliminary Simplifications

If A is a nullable variable

- Whenever A appears on the body of a production A might or might not derive ε

$S \rightarrow ASA \mid aB$
$A \rightarrow B \mid S$
$B \rightarrow b \mid \varepsilon$

Nullable: {A, B}

Preliminary Simplifications

Eliminate ε Productions

- Create two version of the production, one with the nullable variable and one without it
- Eliminate productions with ε bodies

$$\begin{array}{l} S \rightarrow ASA \mid aB \\ A \rightarrow B \mid S \\ B \rightarrow b \mid \varepsilon \end{array} \quad \Rightarrow \quad \begin{array}{l} S \rightarrow ASA \mid aB \mid AS \mid SA \mid S \mid a \\ A \rightarrow B \mid S \\ B \rightarrow b \end{array}$$

Algorithm Nullable Nonterminals

```
NULL := {A | A−>$\lambda$ $\varepsilon$  P};
repeat
        PREV := NULL;
        foreach A $\varepsilon$ V do
            if   there is an A-rule A−>w
                      and   w $\varepsilon$ PREV*
            then NULL := NULL $\cup$ {A}
until   NULL = PREV;
```

Proof of correctness

- Soundness
  - If A ε NULL(*final*) then A=>* λ.
    - Induction on the number of iterations of the loop.
- Completeness
  - If A=>* λ then A ε NULL(*final*).
    - Induction on the minimal derivation of the null string from a non-terminal.
- Termination
    - Bounded by the number of non-terminals.

Elimination of Chain rules

Removing renaming rules: redundant procedure calls.

$$A \rightarrow aA \mid a \mid B$$
$$B \rightarrow bB \mid b \mid C$$
$$C \rightarrow c$$

$$A \rightarrow aA \mid a \mid bB \mid b \mid c$$
$$B \rightarrow bB \mid b \mid c$$
$$C \rightarrow c$$

Top-down flow of information

Construction of Chain(A)

```
Chain(A) := {A};        PREV := $\phi$;
repeat
    NEW := Chain(A)  -  PREV;
    PREV := Chain(A);
    foreach B $\varepsilon$ NEW do
      if   there is a rule B->C
      then Chain(A) := Chain(A) $\cup$ {C}
until  Chain(A) = PREV;
```

Examples

$$\boxed{\begin{aligned} S &\rightarrow AB \mid A \mid B \\ A &\rightarrow aA \mid a \mid B \\ B &\rightarrow b \end{aligned}}$$

$$\boxed{\begin{aligned} S &\rightarrow AB \mid aA \mid a \mid b \\ A &\rightarrow aA \mid a \mid b \\ B &\rightarrow b \end{aligned}}$$

$$\boxed{\begin{aligned} S &\rightarrow aA \mid b \mid A \\ A &\rightarrow Sa \mid B \\ B &\rightarrow bB \mid S \end{aligned}}$$

$$\boxed{\begin{aligned} S &\rightarrow aA \mid b \mid Sa \mid bB \\ A &\rightarrow Sa \mid bB \mid aA \mid b \\ B &\rightarrow bB \mid aA \mid b \mid Sa \end{aligned}}$$

Preliminary Simplifications

Eliminate unit productions

A unit production is one of the form A → B where both A and B are variables

Identify **unit pairs**

$$\boxed{A \overset{*}{\Longrightarrow} B}$$

A → B, B → ω, then A → ω

Preliminary Simplifications

Example:

T = {*, +, (,), a, b, 0, 1}

$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
$F \rightarrow I \mid (E)$
$T \rightarrow F \mid T * F$
$E \rightarrow T \mid E + T$

Basis: (A, A) is a unit pair of any variable A, if $A \overset{*}{\Rightarrow} A$ by 0 steps.

Pairs	Productions
(E, E)	$E \rightarrow E + T$
(E, T)	$E \rightarrow T * F$
(E, F)	$E \rightarrow (E)$
(E, I)	$E \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(T, T)	$T \rightarrow T * F$
(T, F)	$T \rightarrow (E)$
(T, I)	$T \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(F, F)	$F \rightarrow (E)$
(F, I)	$F \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(I, I)	$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Preliminary Simplifications

Example:

Pairs	Productions
…	…
(T, T)	**T → T * F**
(T, F)	**T → (E)**
(T, I)	**T → a | b | Ia |Ib | I0 | I1**
…	…

I → a | b | Ia | Ib | I0 | I1

E → E + T | T * F | (E) | a | b | Ia | Ib | I0 | I1

T → T * F | (E) | a | b | Ia | Ib | I0 | I1

F → (E) | a | b | Ia | Ib | I0 | I1

CNF

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

Proof idea:

- Show that any CFG G can be converted into a CFG G' in Chomsky normal form
- Conversion procedure has several stages where the rules that violate Chomsky normal form conditions are replaced with equivalent rules that satisfy these conditions
- Order of transformations: (1) add a new start variable, (2) eliminate all ϵ-rules, (3) eliminate unit-rules, (4) convert other rules
- Check that the obtained CFG G' defi nes the same language

Final Simplification

Chomsky Normal Form (CNF)

Starting with a CFL grammar with the preliminary simplifications performed

1. Arrange that all bodies of length 2 or more to consists only of variables.
2. Break bodies of length 3 or more into a cascade of productions, each with a body consisting of two variables.

Final Simplification

Step 1: For every terminal α that appears in a body of length 2 or more create a new variable that has only one production.

E → E + T | T * F | (E) | a | b | Ia | Ib | I0 | I1
T → T * F | (E) | a | b | Ia | Ib | I0 | I1
F → (E) | a | b | Ia | Ib | I0 | I1
I → a | b | Ia | Ib | I0 | I1

E → EPT | TMF | LER | a | b | IA | IB | IZ | IO
T → TMF | LER | a | b | IA | IB | IZ | IO
F → LER | a | b | IA | IB | IZ | IO
I → a | b | IA | IB | IZ | IO
A → a   B → b   Z → 0   O → 1
P → +   M → *   L → (R →)

Final Simplification

Step 2: Break bodies of length 3 or more adding more variables

$E \rightarrow E\mathbf{PT} \mid T\mathbf{MF} \mid L\mathbf{ER} \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$

$T \rightarrow T\mathbf{MF} \mid L\mathbf{ER} \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$

$F \rightarrow L\mathbf{ER} \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$

$I \rightarrow a \mid b \mid IA \mid IB \mid IZ \mid IO$

$A \rightarrow a \quad B \rightarrow b \quad Z \rightarrow 0 \quad O \rightarrow 1$

$P \rightarrow + \quad M \rightarrow * \quad L \rightarrow (\quad R \rightarrow)$

$C_1 \rightarrow PT$

$C_2 \rightarrow MF$

$C_3 \rightarrow ER$

Theorem: If G is in CNF, $w \in L(G)$ and $|w| > 0$, then any derivation of w in G has length $2|w| - 1$

Proof (by induction on |w|):

Base Case: If $|w| = 1$, then any derivation of w must have length 1 $(S \rightarrow a)$

Inductive Step: Assume true for any string of length at most $k \geq 1$, and let $|w| = k+1$

Since $|w| > 1$, derivation starts with $S \rightarrow AB$

So $w = xy$ where $A \Rightarrow^* x$, $|x| > 0$ and $B \Rightarrow^* y$, $|y| > 0$

By the inductive hypothesis, the length of any derivation of w must be

$$1 + (2|x| - 1) + (2|y| - 1) = 2(|x| + |y|) - 1$$

Theorem: Any context-free language can be generated by a context-free grammar in Chomsky normal form

“Can transform any CFG into Chomsky normal form”

Theorem: Any context-free language can be generated by a context-free grammar in Chomsky normal form

Proof Idea:

1. Add a new start variable
2. Eliminate all $\mathbf{A \rightarrow \varepsilon}$ rules. Repair grammar
3. Eliminate all $\mathbf{A \rightarrow B}$ rules. Repair

4. Convert $\mathbf{A \rightarrow u_1 u_2 \ldots u_k}$ to $\mathbf{A \rightarrow u_1 A_1, A_1 \rightarrow u_2 A_2, \ldots}$
If $\mathbf{u_i}$ is a terminal, replace $\mathbf{u_i}$ with $\mathbf{U_i}$ and add $\mathbf{U_i \rightarrow u_i}$

1. Add a new start variable S_0 and add the rule $S_0 \to S$
2. Remove all $A \to \varepsilon$ rules (where A is not S_0)

   For each occurrence of A on right hand side of a rule, add a new rule with the occurrence deleted

   If we have the rule $B \to A$, add $B \to \varepsilon$, unless we have previously removed $B \to \varepsilon$

3. Remove unit rules $A \to B$

   Whenever $B \to w$ appears, add the rule $A \to w$ unless this was a unit rule previously removed

$S_0 \to S$

$S \to 0S1$

$S \to T\#T$

$S \to T$

$T \to \varepsilon$

$S \to T\#$

$S \to \#T$

$S \to \#$

$S \to \varepsilon$

$S \to 01$

$S_0 \to 0S1$

$S_0 \to \varepsilon$

4. Convert all remaining rules into the proper form:

$S_0 \to 0S1$

$S_0 \to A_1A_2$

$A_1 \to 0$

$A_2 \to SA_3$

$A_3 \to 1$

$S_0 \to 01$

$S_0 \to A_1A_3$

$S \to 01$

$S \to A_1A_3$

$S_0 \to \varepsilon$

$S_0 \to 0S1$

$S_0 \to T\#T$

$S_0 \to T\#$

$S_0 \to \#T$

$S_0 \to \#$

$S_0 \to 01$

$S \to 0S1$

$S \to T\#T$

$S \to T\#$

$S \to \#T$

$S \to \#$

$S \to 01$

Convert the following into Chomsky normal form:

$A \rightarrow BAB \mid B \mid \varepsilon$

$B \rightarrow 00 \mid \varepsilon$

$S_0 \rightarrow A$

$A \rightarrow BAB \mid B \mid \varepsilon$

$B \rightarrow 00 \mid \varepsilon$

$S_0 \rightarrow A \mid \varepsilon$

$A \rightarrow BAB \mid B \mid BB \mid AB \mid BA$

$B \rightarrow 00$

$S_0 \rightarrow BAB \mid 00 \mid BB \mid AB \mid BA \mid \varepsilon$

$A \rightarrow BAB \mid 00 \mid BB \mid AB \mid BA$

$B \rightarrow 00$

$S_0 \rightarrow BC \mid DD \mid BB \mid AB \mid BA \mid \varepsilon$, $C \rightarrow AB$,

$A \rightarrow BC \mid DD \mid BB \mid AB \mid BA$, $B \rightarrow DD$, $D \rightarrow 0$

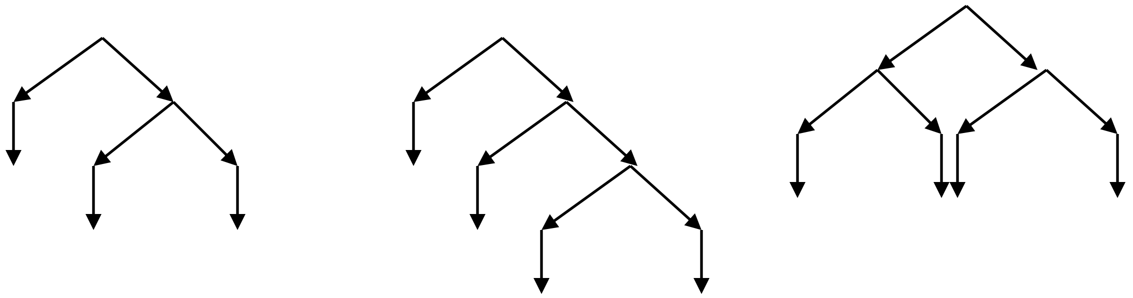
Significance of CNF

- Length of derivation of a string of length *n* in CNF = (2*n*-1)

  (*Cf*. Number of nodes of a strictly binary tree with *n*-leaves)

- Maximum depth of a parse tree = *n*
- Minimum depth of a parse tree =

$$\lceil \log_2 n \rceil + 1$$



Removal of direct left recursion

- Causes infinite loop in top-down (depth-first) parsers.

$$A \rightarrow Aa \mid b$$

$$\boxed{L(A) = ba*}$$

- *Approach*: Generate string from left to right.

$$A \rightarrow bZ \mid b$$
$$Z \rightarrow aZ \mid a$$

$$\boxed{\begin{array}{l} L(A) = ba* \\ L(Z) = a^{+} \end{array}}$$

Greibach Normal Form

A context free grammar is said to be in **Greibach Normal Form** if all productions are in the following form:

$$A \rightarrow \alpha X$$

- A is a non terminal symbols
- α is a terminal symbol
- X is a sequence of non terminal symbols.
  It may be empty.

Greibach Normal Form

Example:

CNF	New Labels	Updated CNF
$S \rightarrow XA \mid BB$	$S = A_1$	$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$B \rightarrow b \mid SB$	$X = A_2$	$A_4 \rightarrow b \mid A_1A_4$
$X \rightarrow b$	$A = A_3$	$A_2 \rightarrow b$
$A \rightarrow a$	$B = A_4$	$A_3 \rightarrow a$

Greibach Normal Form

Example:

$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$A_4 \rightarrow b \mid A_1A_4$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

First Step $\boxed{A_i \rightarrow A_jX_k \quad j > i}$

X_k is a string of zero or more variables

✗ $A_4 \rightarrow A_1A_4$

Greibach Normal Form

Example:

First Step $\boxed{A_i \rightarrow A_j X_k \quad j > i}$

$A_4 \rightarrow \underline{A_1}A_4$

$A_4 \rightarrow \underline{A_2}A_3A_4 \mid A_4A_4A_4 \mid b$

$A_4 \rightarrow bA_3A_4 \mid A_4A_4A_4 \mid b$

$A_1 \rightarrow A_2A_3 \mid A_4A_4$

$A_4 \rightarrow b \mid A_1A_4$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Greibach Normal Form

Example:

$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$A_4 \rightarrow bA_3A_4 \mid A_4A_4A_4 \mid b$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

Second Step

Eliminate Left Recursions

$A \rightarrow A\,\alpha \mid \beta$

Can be written as

$A \rightarrow \beta\, A'$
$A' \rightarrow \alpha\, A' \mid \varepsilon$

✗ $A_4 \rightarrow A_4A_4A_4$

Greibach Normal Form

Example:

Second Step

Eliminate Left Recursions

$A_4 \rightarrow bA_3A_4 \mid b \mid bA_3A_4Z \mid bZ$

$Z \rightarrow A_4A_4 \mid A_4A_4Z$

$A_1 \rightarrow A_2A_3 \mid A_4A_4$

$A_4 \rightarrow bA_3A_4 \mid \bigcirc\!\!\!\!\;A_4A_4A_4 \mid b$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Greibach Normal Form

Example:

$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$A_4 \rightarrow bA_3A_4 \mid b \mid bA_3A_4Z \mid bZ$
$Z \rightarrow A_4A_4 \mid A_4A_4Z$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

$\boxed{A \rightarrow \alpha X}$

GNF

Greibach Normal Form

Example:

$$A_1 \rightarrow \underline{A_2}A_3 \mid \underline{A_4}A_4$$

$$A_4 \rightarrow \boxed{bA_3A_4} \mid b \mid bA_3A_4Z \mid bZ$$

$$Z \rightarrow A_4A_4 \mid A_4A_4Z$$

$$A_2 \rightarrow b$$

$$A_3 \rightarrow a$$

$$A_1 \rightarrow bA_3 \mid bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4$$

$$Z \rightarrow bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4 \mid bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4$$

Greibach Normal Form

Example:

$A_1 \rightarrow bA_3 \mid bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4$
$A_4 \rightarrow bA_3A_4 \mid b \mid bA_3A_4Z \mid bZ$
$Z \rightarrow bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4 \mid bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

Grammar in Greibach Normal Form

Presentation Outline

Summary (Some properties)

- Every CFG that doesn’t generate the empty string can be simplified to the Chomsky Normal Form and Greibach Normal Form
- The derivation tree in a grammar in CNF is a binary tree
- In the GNF, a string of length n has a derivation of exactly n steps
- Grammars in normal form can facilitate proofs
- CNF is used as starting point in the algorithm CYK

- The size of the equivalent GNF can be large compared to the original grammar.
    - Example CFG has 5 rules, but the corresponding GNF has 24 rules!!

- Length of the derivation in GNF

  = Length of the string.

- GNF is useful in relating CFGs (“generators”) to pushdown automata (“recognizers”/”acceptors”).

- *Theorem:* There is an algorithm to construct a grammar G’ in GNF that is *equivalent* to a CFG G.

Trading Left- & Right-Recursion

Left recursion: $A \rightarrow A\ \alpha$

Right recursion: $A \rightarrow \alpha\ A$

Most algorithms have trouble with one,

In recursive descent, avoid left recursion.

Removing Left Recursion

For each rule which contains a left-recursive option,

$A \dashrightarrow A\alpha \mid \beta$

introduce a new nonterminal A' and rewrite the rule as

$A \dashrightarrow \beta A'$

$A' \dashrightarrow \varepsilon \mid \alpha A'$

Thus the production:

E --> E + T | T

E --> T E'

E' --> ε | + T E'

Of course, there may be more than one left-recursive part on the right-hand side. The general rule is to replace:

$A \dashrightarrow A\alpha_1 \mid \alpha_2 \mid \ldots\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$

by

$A \dashrightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$

$A' \dashrightarrow \varepsilon \mid \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A'$

Removing Indirect Left Recursion

A --> B **x y** | **x**
B --> C D
C --> A | **c**
D --> **d**

is indirectly recursive because
A ==> B **x y** ==> C D **x y** ==> A D **x y.**
That is, A ==> … ==> A   where   is D **x y.**

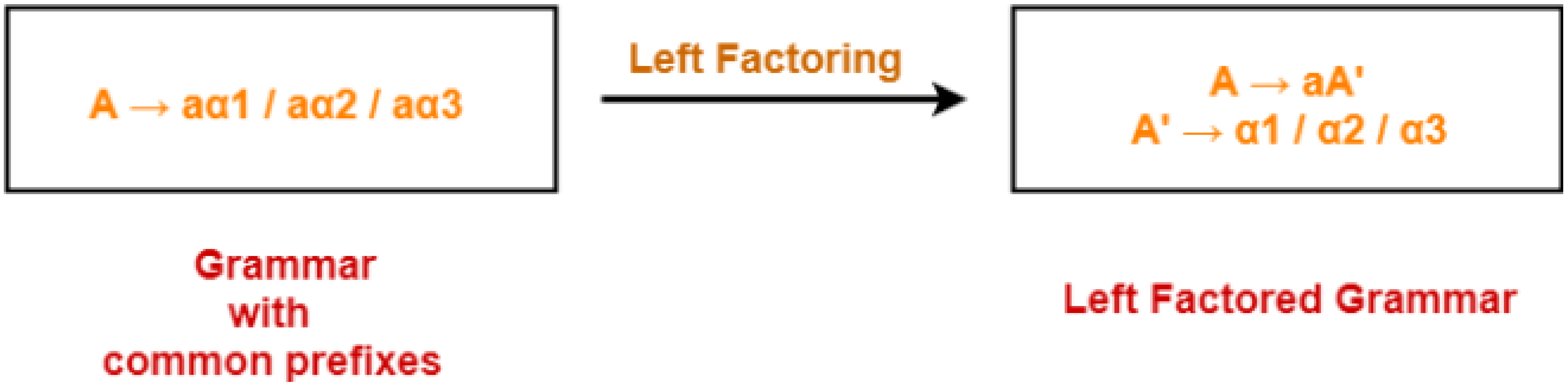
Removing Left Factoring of Grammar

Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers.

In left factoring,
•We make one production for each common prefixes.
•The common prefix may be a terminal or a non-terminal or a combination of both.
•Rest of the derivation is added by new productions.

The grammar obtained after the process of left factoring is called as **Left Factored Grammar**.

Removing Left Factoring of Grammar



Do left factoring in the following grammar-

S → iEtS / iEtSeS / a

E → b

The left factored grammar is-

S → iEtSS’ / a

S’ → eS / ∈

E → b

THANK YOU

More on next class…