CS375:
Logic and Theory of Computing

Fuhua (Frank) Cheng

Department of Computer Science

University of Kentucky

Table of Contents:

- **Week 1: Preliminaries (set algebra, relations, functions) (read Chapters 1-4)**
- **Weeks 3-6: Regular Languages, Finite Automata (Chapter 11)**
- **Weeks 7-9: Context-Free Languages, Pushdown Automata (Chapters 12)**
- **Weeks 10-12: Turing Machines (Chapter 13)**

Table of Contents (conti):

- **Weeks 13-14: Propositional Logic (Chapter 6), Predicate Logic (Chapter 7), Computational Logic (Chapter 9), Algebraic Structures (Chapter 10)**

7. Context-Free Languages & Pushdown Automat ee Parsing

A black box assembly line

Cut open the belly

Goal:

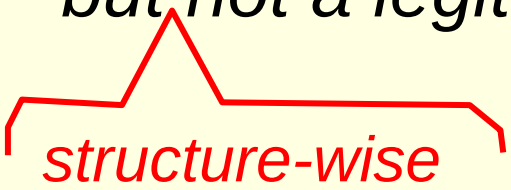
- Declarative rmalisms like CFGs, FSAs d ne the *legal string f a language*--but only tell y ‘this is a legal string the language X’
- *Parsing alg ithms* specify how to recog nize the strings of a anguage and how to do syntactic analysis of a string

7. Context-Free Languages & Pushdown Automata - Context-Free Parsing

For instance,

"I like like spaghetti" could be considered a legal string

but not a legitimate sentence



Important Info:

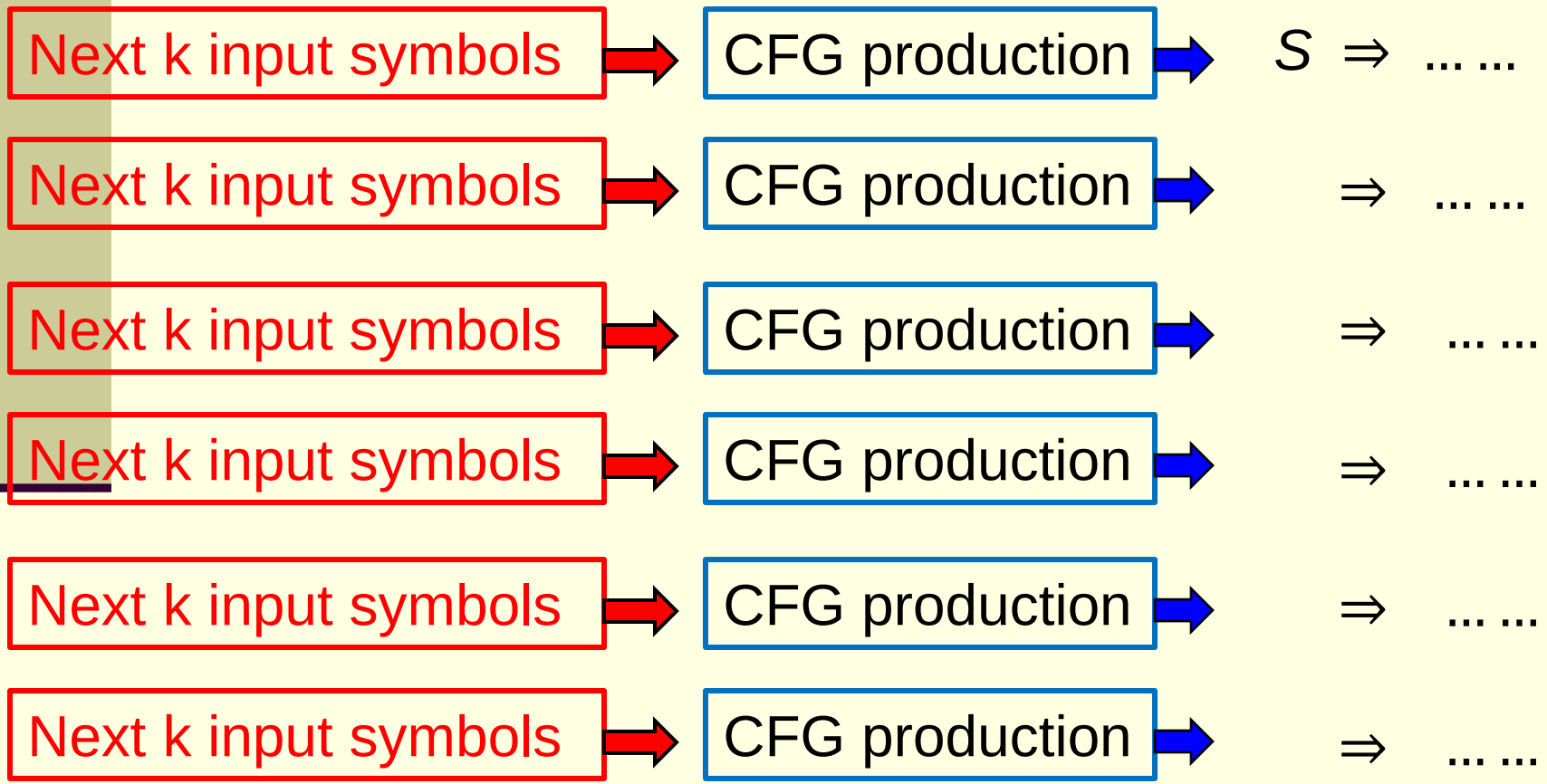
- Most programming languages have LL(1) grammars.
- LL(1) grammars are never ambiguous.
- LL(1) grammars are never left-recursive.

LL(k) Grammar:

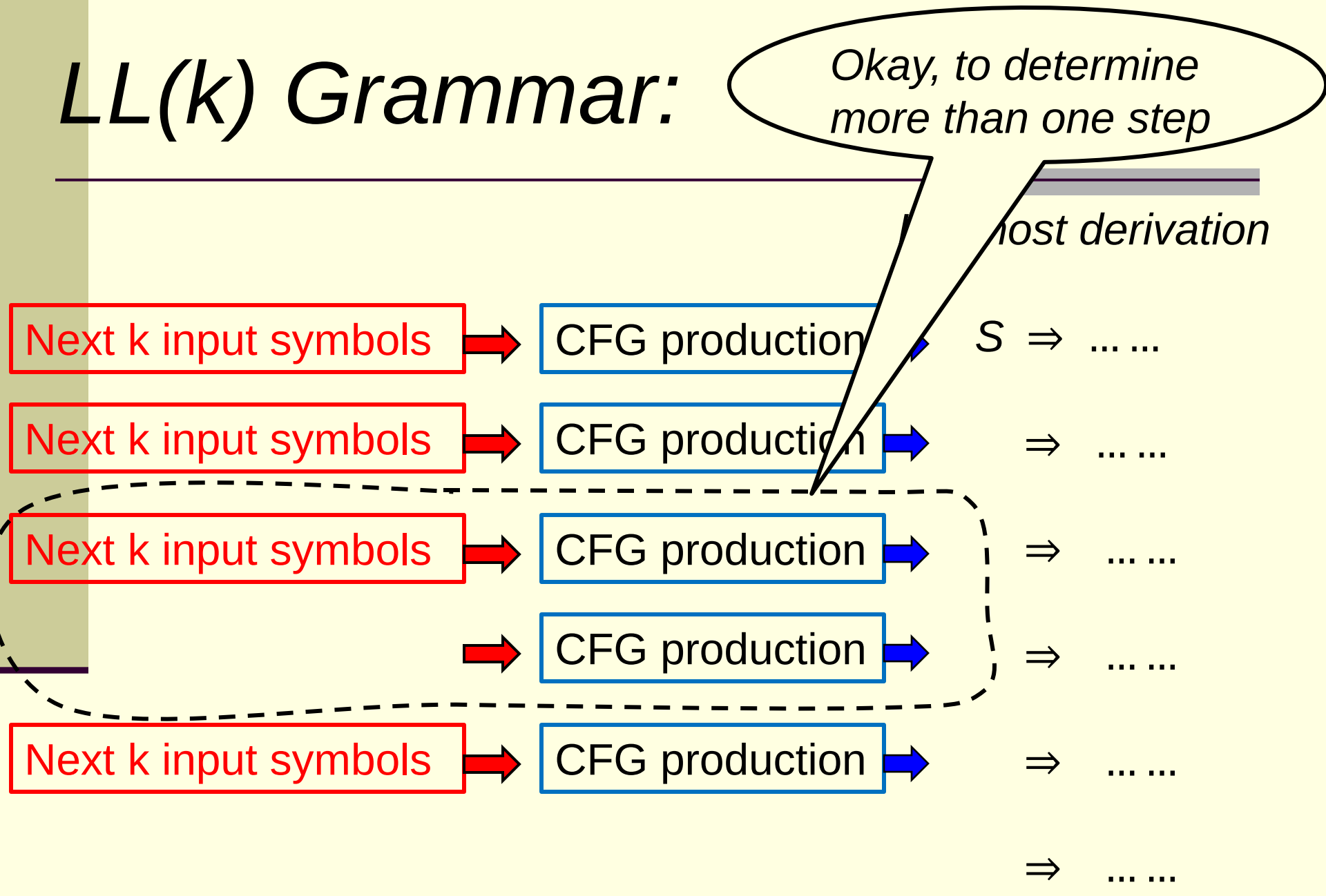
- A CFG is called an *LL(k) grammar* if a **parser** can be constructed to scan an input string from left to right and build a leftmost derivation by examining next *k* input symbols to determine the **unique** production for each derivation step.
- If a language has an LL(*k*) grammar, it is called an *LL(k) language*.

LL(k) Grammar:

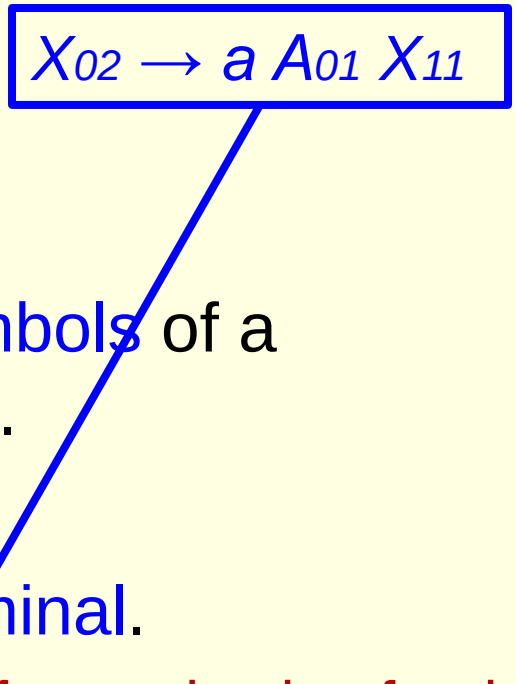
Leftmost derivation



LL(k) Grammar:


Okay, to determine more than one step
lost derivation
Next k input symbols
CFG production
Next k input symbols
CFG production
Next k input symbols
CFG production
CFG production
Next k input symbols
CFG production
S ⇒ ... ...
⇒ ... ...
⇒ ... ...
⇒ ... ...
⇒ ... ...
⇒ ... ...

First, a few things about parse tree, parser and parsing:

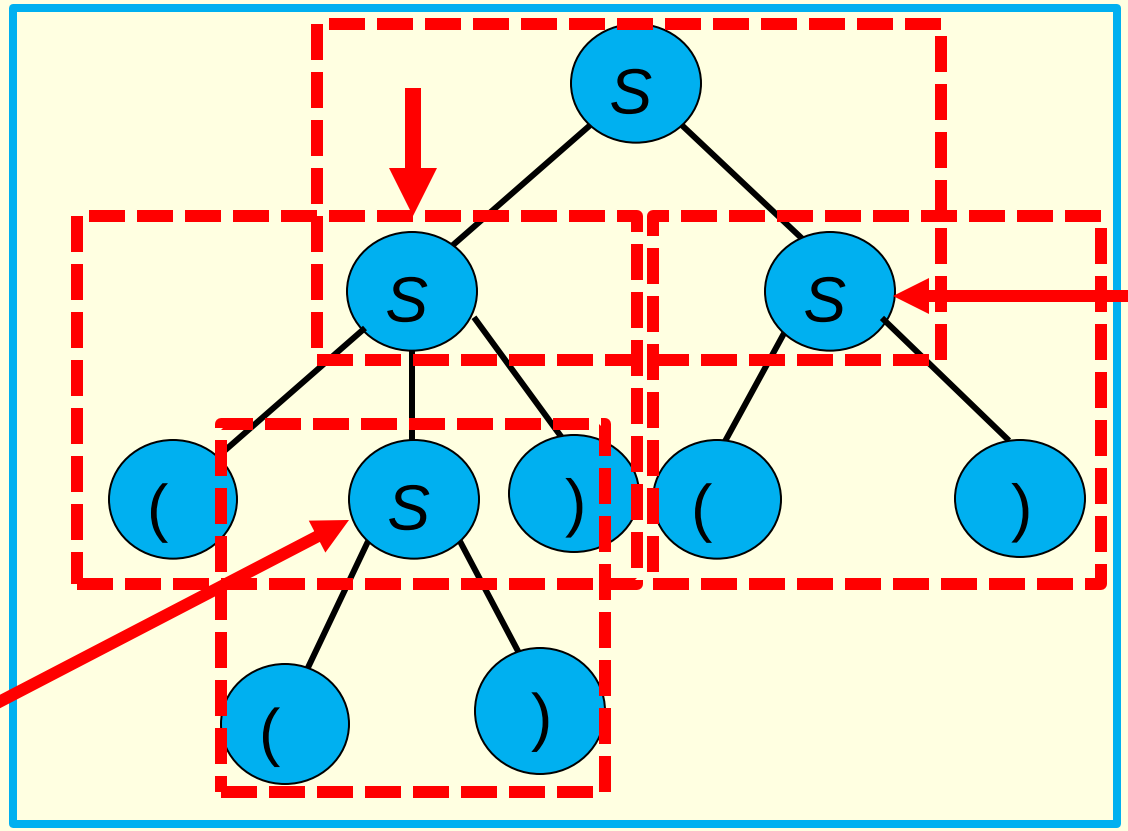


Parse Trees

- *Parse trees* are trees labeled by symbols of a particular CFG (in a particular order).
- *Leaves*: labeled by a terminal or Λ.
- *Interior nodes*: labeled by a non-terminal.
  - Children are labeled by the right side of a production for the parent.
- *Root*: must be labeled by the start symbol

Example: **Parse Tree**

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



$S \rightarrow SS$

$S \rightarrow (S)$

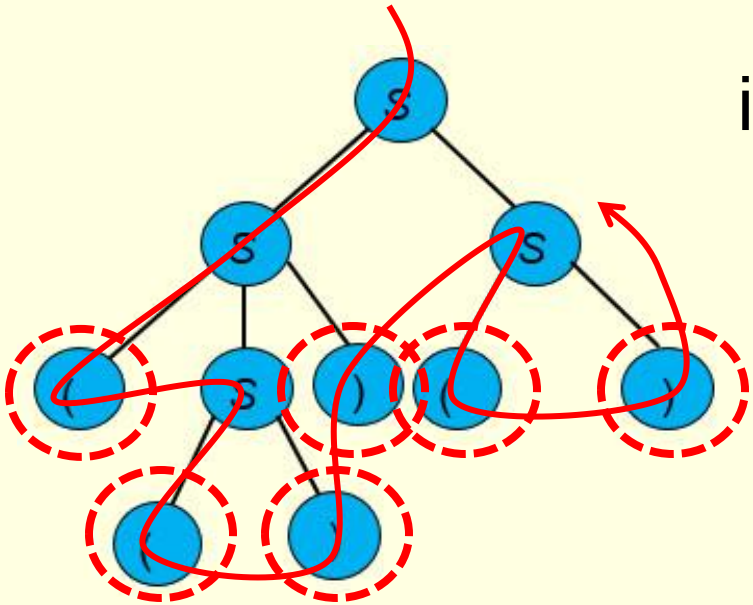
$S \rightarrow ( )$

$S \rightarrow ( )$

Yield of a Parse Tree

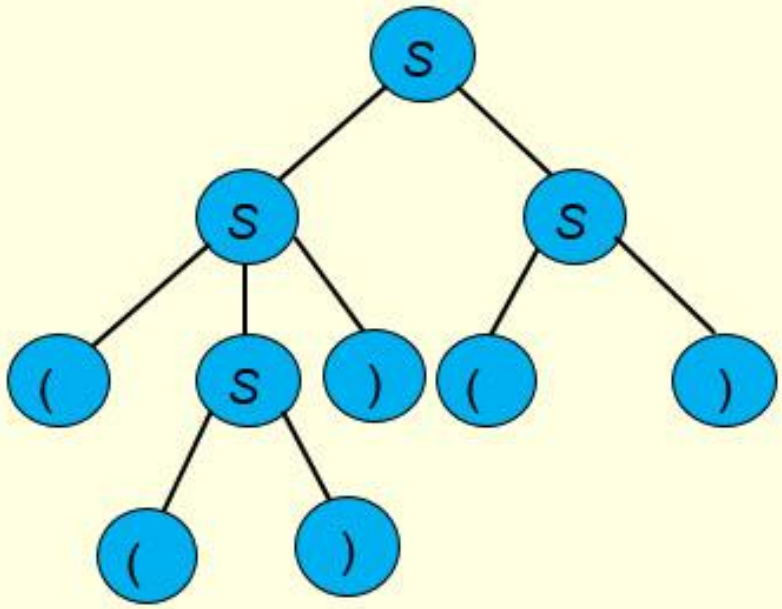
- The concatenation of the labels of the leaves in left-to-right order.
  - That is, in the order of a preorder traversal.
- is called the ***yield*** of the parse tree.

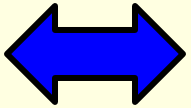
Root-L-R

- Example: yield of  is (( )) ( )

Parse Trees, Left- and Rightmost Derivations

- For every parse tree, there is a unique leftmost, and a unique rightmost derivation.





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

Example. Consider the language

$$\{\ a^n b\ \mid n \in \mathbf{N}\}.$$

(1) It has an LL(1) grammar

$$S \rightarrow aS \mid b$$

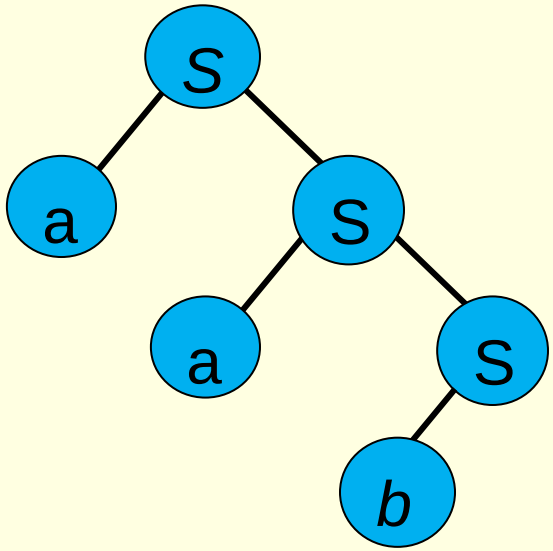
A parser can examine one input letter to decide whether to use $S \rightarrow aS$ or $S \rightarrow b$ for the next derivation step.

Consider: $a\ a\ b$

$S \Rightarrow aS$

$\Rightarrow a\ a\ S$

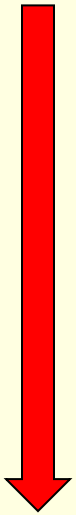
$\Rightarrow a\ a\ b$



Derivation:

Given a grammar: **G:** $S \rightarrow SS \mid (S) \mid ()$

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

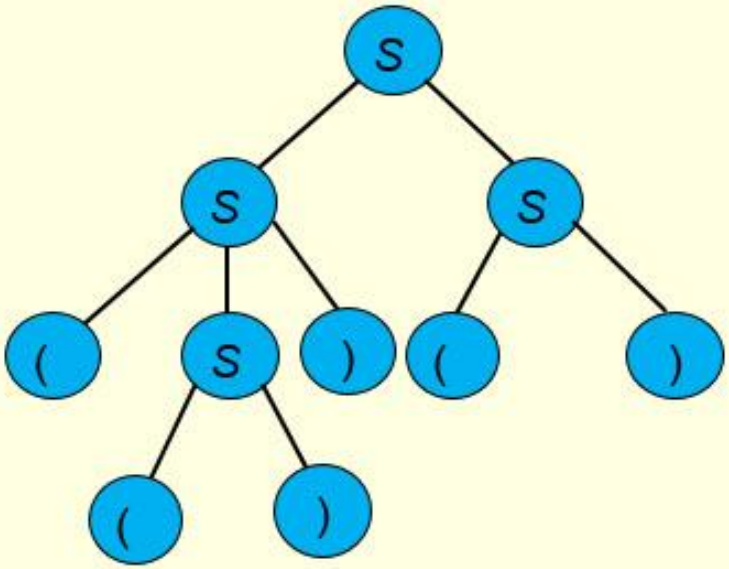


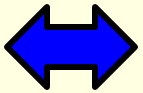
(leftmost) derivation

Conclusion: $(())()$ is a legal string in **L(G)**

Parsing:

Given a parse tree



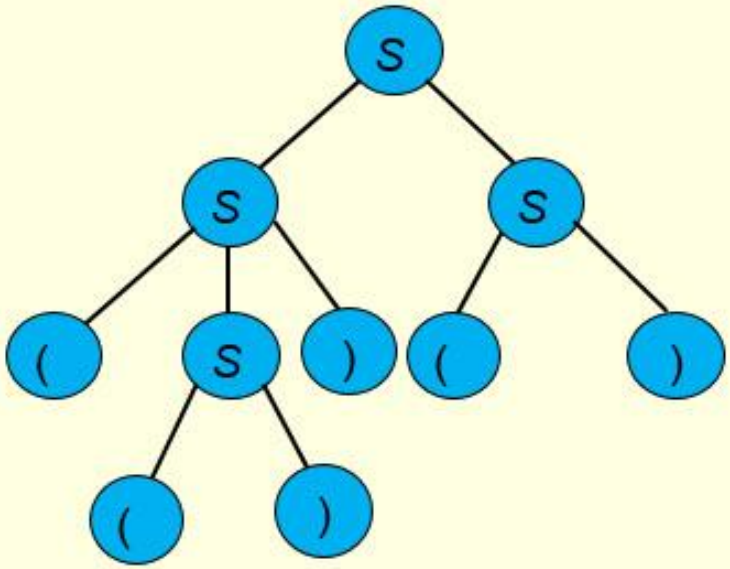


S
⇒ SS
⇒ (S)S
⇒ (())S
⇒ (())()

Conclusion: (())() is a legal string in L(G) because it has a valid structure

Parsing:

Given a parse tree



(productions)

S → SS

S → (S)

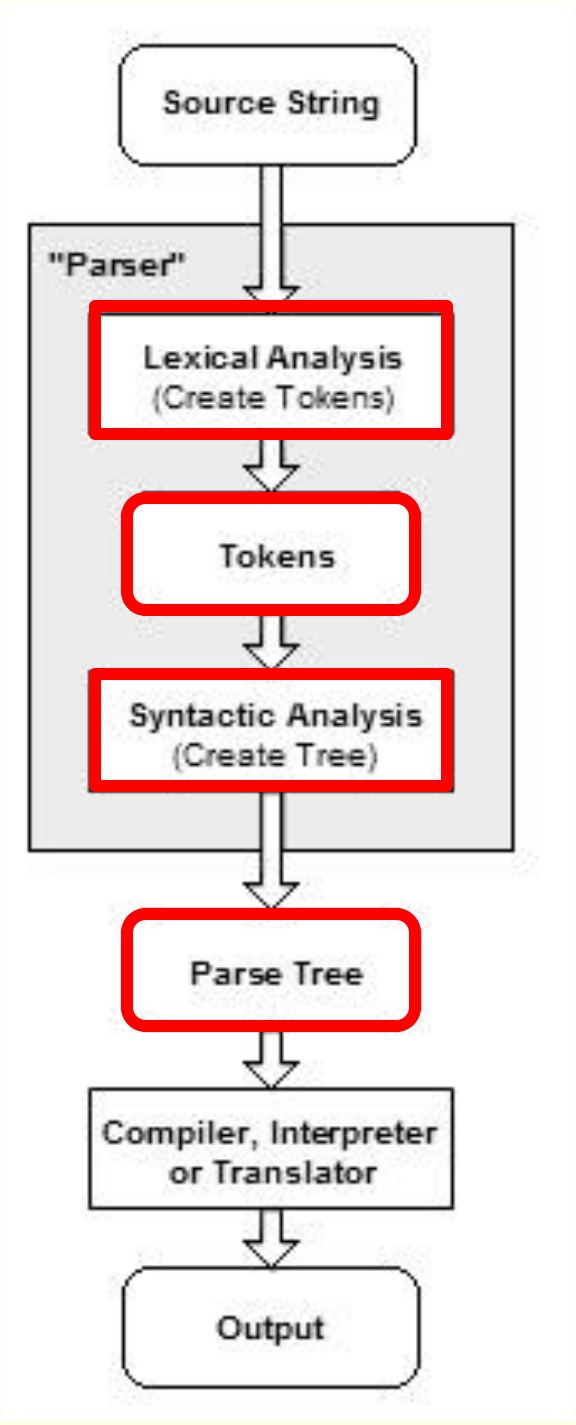
S → ( )

We can even tell if a grammar is an LL(1), LL(2), … grammar

Parsing:

Actually parsing contains several steps.

But for now, a conceptual understanding is enough



Continue on LL(k) grammar …

- A CFG is called an *LL(k) grammar* if a parser can be constructed to scan an input string from left to right and build a leftmost derivation by examining next *k* input symbols to determine the unique production for each derivation step.
- If a language has an LL(*k*) grammar, it is called an *LL(k) language*.

Example. Consider the language

$$\{\ a^n b\ \mid n \in \mathbf{N}\}.$$

(2) It has an LL(2) grammar

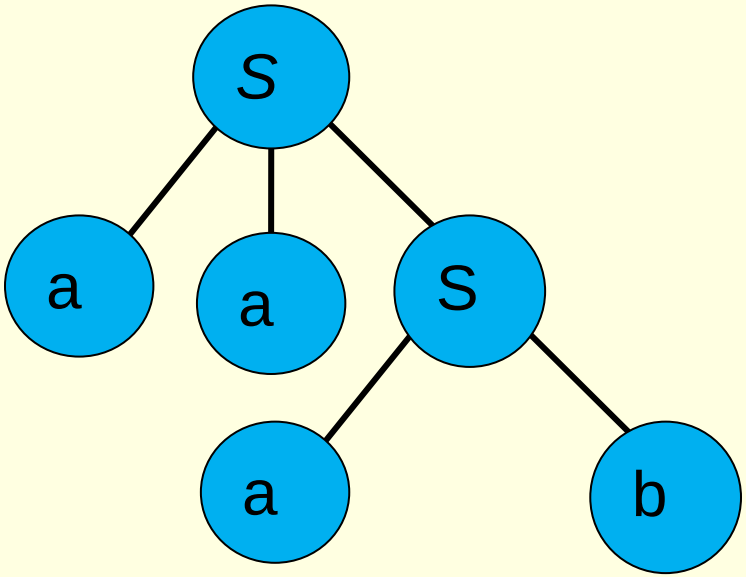
$$S \rightarrow aaS \mid ab \mid b$$

A parser can examine two input letters to decide whether to use $S \rightarrow aaS$, $S \rightarrow ab$ or $S \rightarrow b$ for the next derivation step.

Consider: | *a a* | *a b* | |

$$S \Rightarrow \boxed{a}\boxed{a}S$$

$$\Rightarrow a\ a\ \boxed{a}\boxed{b}$$



Example. Consider the language

$$\{\ a^n b\ \mid n \in \mathbf{N}\}.$$

(2) It has an LL(2) grammar

$$S \rightarrow aaS \mid ab \mid b$$

A parser can examine two input letters to decide whether to use $S \rightarrow aaS$, $S \rightarrow ab$ or $S \rightarrow b$ for the next derivation step.

Question 1: *Is this grammar LL(1)?*

No. *Consider* aab

Can not determine $S \rightarrow aaS$ *or* $S \rightarrow ab$ *to use*

Example. Consider the language

$$\{\ a^n b\ \mid n \in \mathbf{N}\}.$$

(2) It has an LL(2) grammar

$$S \rightarrow aaS \mid ab \mid b$$

A parser can examine two input letters to decide whether to use $S \rightarrow aaS$, $S \rightarrow ab$ or $S \rightarrow b$ for the next derivation step.

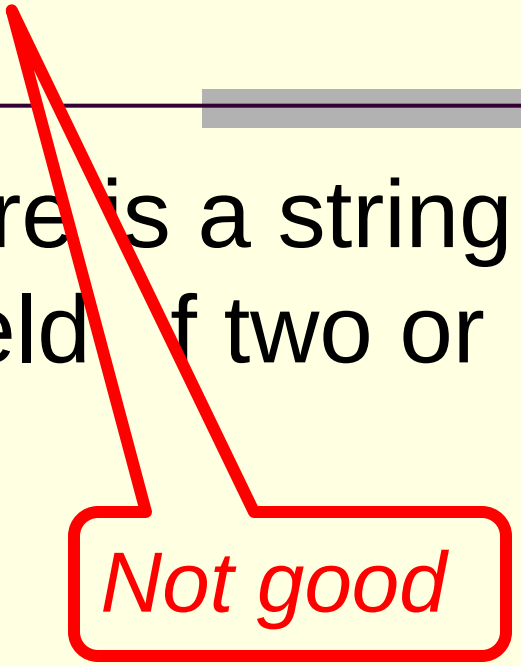
Question 2: *can you find an LL(3) grammar that is not LL(2)?*

Answer: $S \rightarrow aaaS \mid aab \mid ab \mid b$

and consider the strings aaaaab, aaaab, aaab

Ambiguous Grammars

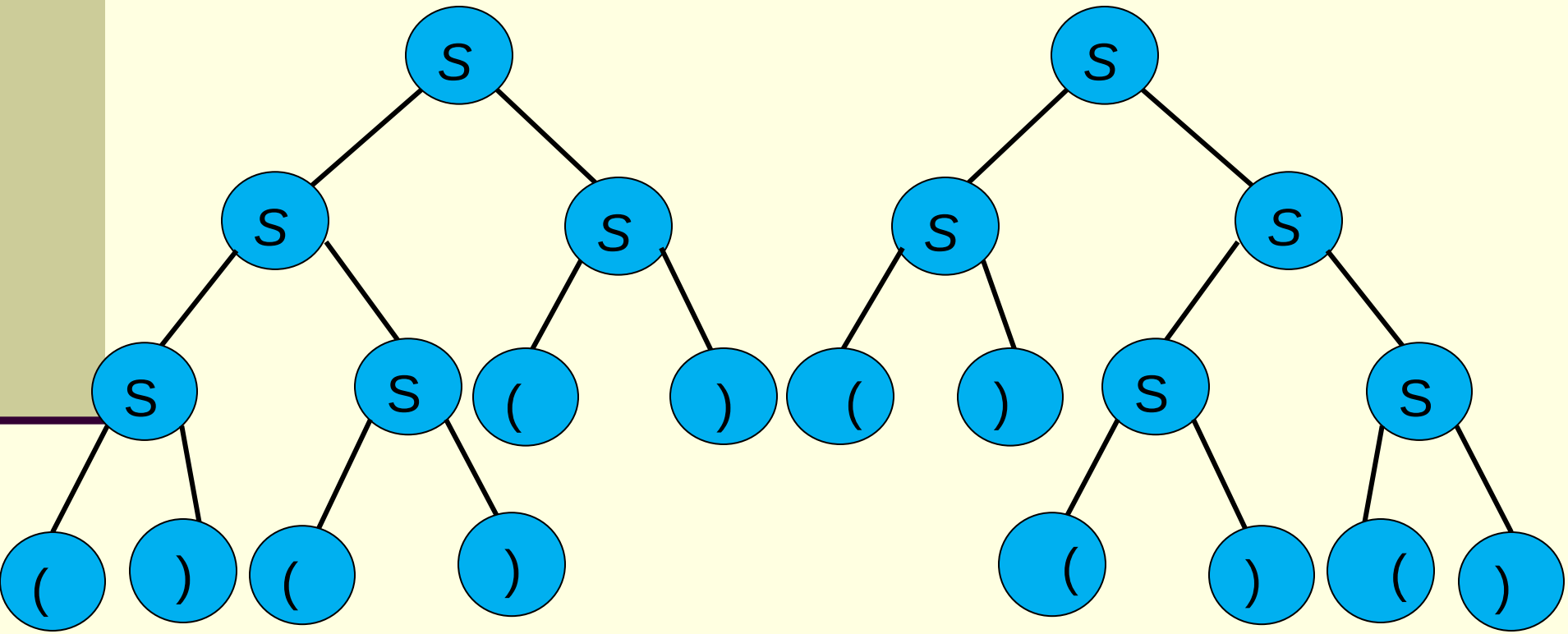
- A CFG is ambiguous if there is a string in the language that is the yield of two or more parse trees.
- Example: S -> SS | (S) | ()



- Two parse trees for ()()() on next slide.

Ambiguous Grammars

Example: ()()()



Facts:

Depending on if identifiers/symbols are included in the languages

- Most programming languages have LL(1) grammars.

  (What if a, aa, aaa are the names of three variables?)

- LL(1) grammars are never ambiguous.

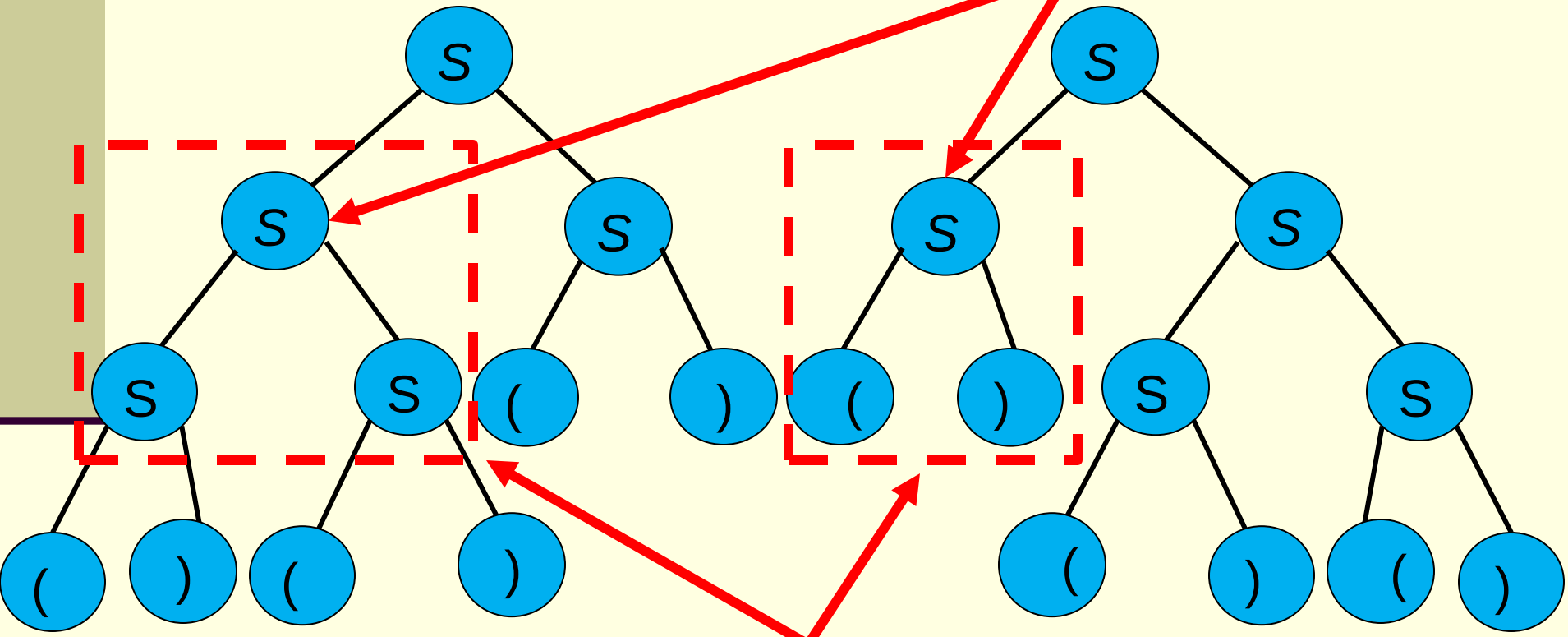
LL(1) grammars are
- not ambiguous and
- not left-recursive.

LL(1) grammars are not ambiguous.

Why?

Example: ()()()

If the grammar is ambiguous, find the first internal node whose child nodes are different.



For that internal node, we have two different production choices for that derivation step, a contradiction.

LL(1) grammars are not left-recursive. Why?

If an LL(1) grammar is left-recursive, then there is at least a production of the form

$$S \rightarrow SA$$

But then there must be a production of the form

$$S \rightarrow B$$

to terminate the recursion. So when ‘S’ is scanned, we would have two options to choose from, a contradiction.

Questions:

Is the grammar

$S \rightarrow AB$; $A \rightarrow aAb \mid \Lambda$; $B \rightarrow bB \mid \Lambda$

for $\{ a^n b^{n+k} \mid n, k \in \mathbf{N} \}$ an LL(1) grammar?

Consider: b b

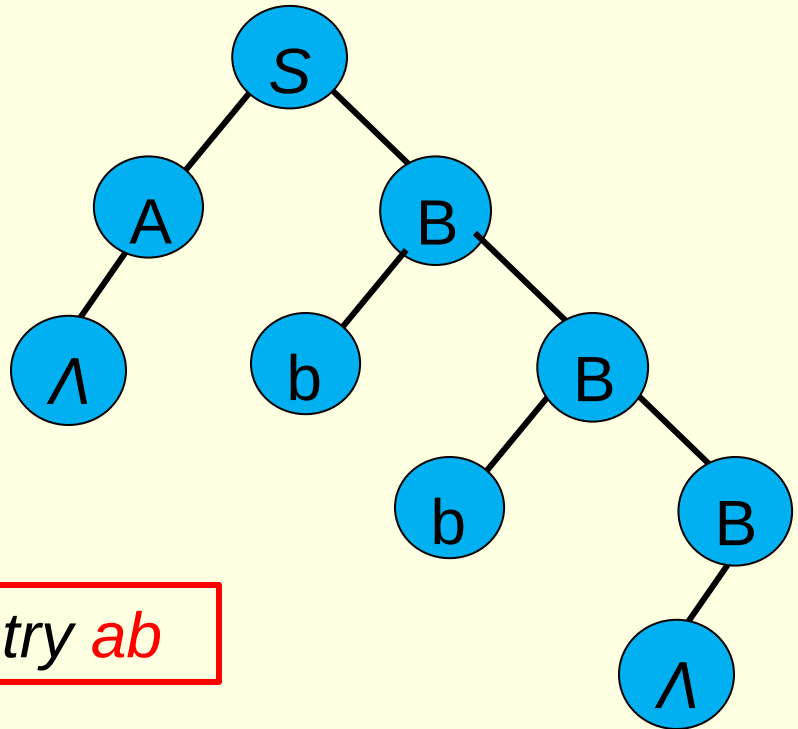
$S \Rightarrow AB$

$\Rightarrow \Lambda B$

$\Rightarrow \Lambda b B$

$\Rightarrow \Lambda b bB$

$\Rightarrow \Lambda b b \Lambda$



Then try ab

Questions:

Is the grammar

$S \rightarrow AB$; $A \rightarrow aAb \mid \Lambda$; $B \rightarrow bB \mid \Lambda$

for $\{ a^n b^{n+k} \mid n, k \in \mathbf{N} \}$ an LL(1) grammar?

Consider: a b b

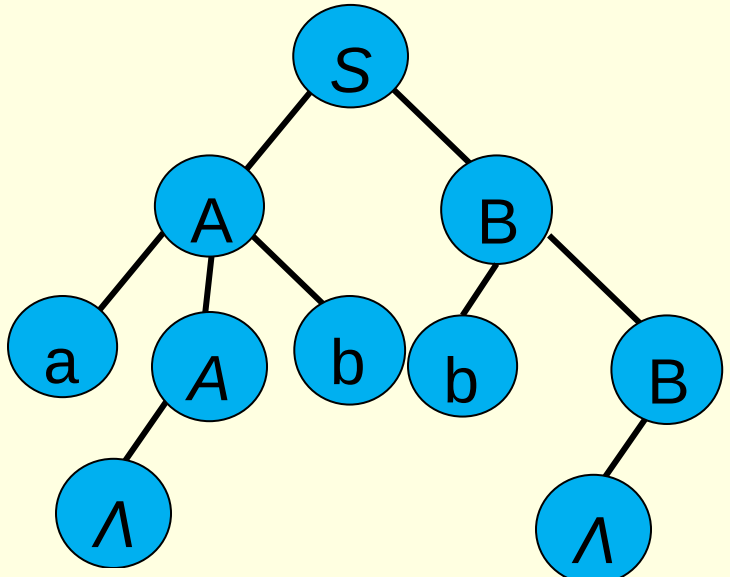
$S \Rightarrow AB$

$\Rightarrow a\,A\,b\,B$

$\Rightarrow a\,\Lambda\,b\,B$

$\Rightarrow a\,\Lambda\,b\,b\,B$

$\Rightarrow a\,\Lambda\,b\,b\,\Lambda$



Yes or No

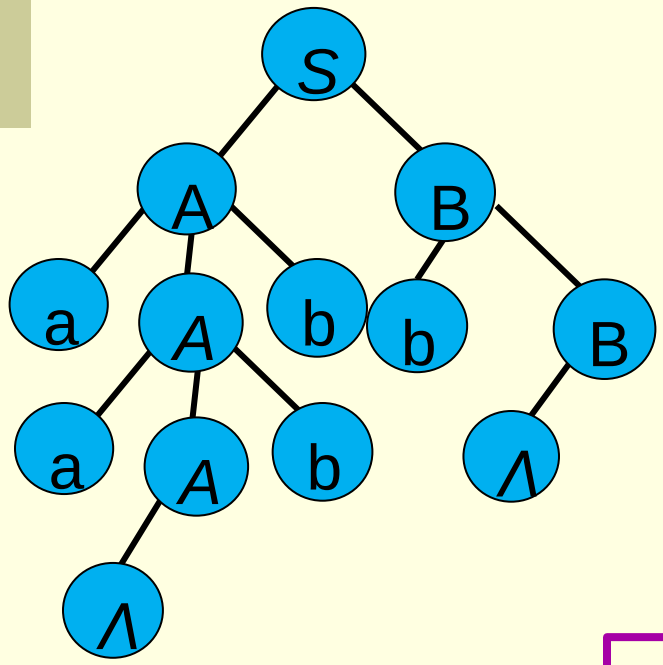
Then try aabb

Questions:

Is the grammar

$S \rightarrow AB$; $A \rightarrow aAb \mid \Lambda$; $B \rightarrow bB \mid \Lambda$

for $\{ a^n b^{n+k} \mid n, k \in \mathbf{N} \}$ an LL(1) grammar?



Consider: a a b b b

$S \Rightarrow AB$

$\Rightarrow a\,A\,b\,B$

$\Rightarrow a\,a\,A\,b\,b\,B$

$\Rightarrow a\,a\,\Lambda\,b\,b\,B$

$\Rightarrow a\,a\,\Lambda\,b\,b\,b\,B$

$\Rightarrow a\,a\,\Lambda\,b\,b\,b\,\Lambda$

Yes or No

Questions:

Is the grammar

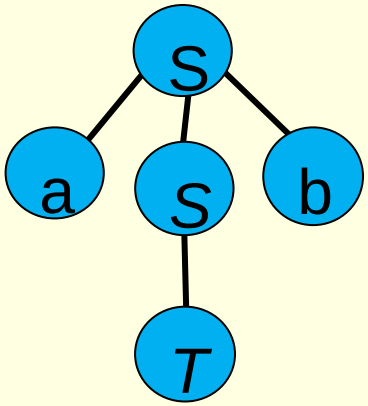
$S \rightarrow aSb \mid T$; $T \rightarrow bT \mid \Lambda$

for $\{ a^n b^{n+k} \mid n, k \in \mathbf{N} \}$ an LL(1) grammar?

Consider: a b

$S \Rightarrow a S b$

$\Rightarrow a T b$



Yes or No

It gets stuck here

Questions:

Is the grammar

$S \rightarrow aSb \mid T$; $T \rightarrow bT \mid \Lambda$

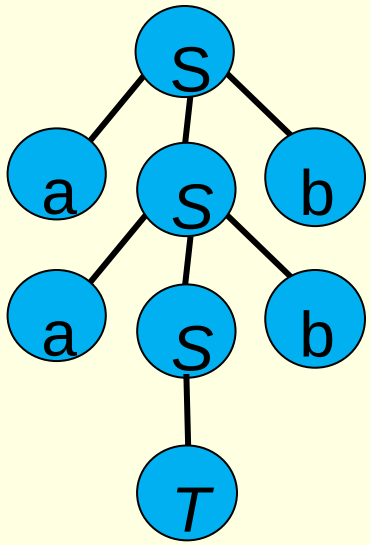
for $\{ a^n b^{n+k} \mid n, k \in \mathbf{N} \}$ an LL(2) grammar?

Consider: a a b b

$S \Rightarrow a\,S\,b$

$\Rightarrow a\,a\,S\,b\,b$

$\Rightarrow a\,a\,T\,b\,b$



Yes or No

It gets stuck here

Questions:

Is the grammar

$S \rightarrow aSb \mid T$; $T \rightarrow bT \mid \Lambda$

for $\{ a^n b^{n+k} \mid n, k \in \mathbf{N} \}$ an LL(k) grammar for k>2?

Quiz on your time

For k=3, check aaabbb

For k=4, check aaaabbbb

For k=n, check $a^n b^n$

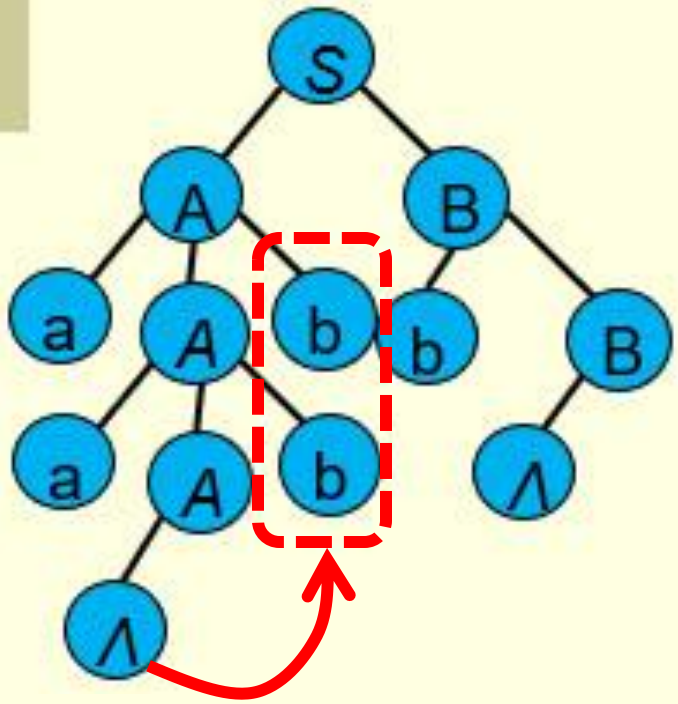
What is the difference here?

Why the grammar { $S \to AB$; $A \to aAb \mid \Lambda$; $B \to bB \mid \Lambda$ }

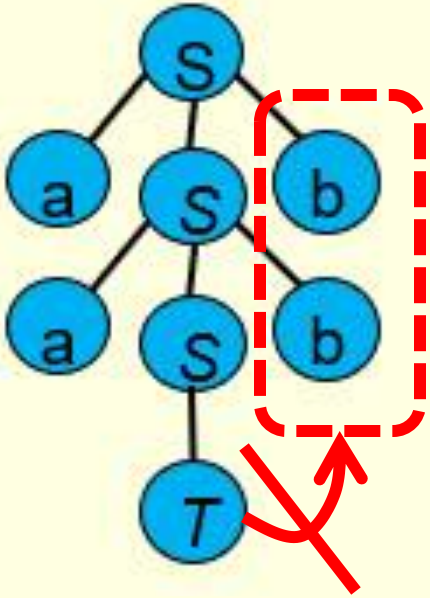
is LL(1) for $\{ a^n b^{n+k} \mid n, k \in \mathbf{N} \}$?

but the grammar { $S \to aSb \mid T$; $T \to bT \mid \Lambda$ }

for the same language is not LL(k) for any k>0?



In the second case, we did not give the parsing process a chance to look back before it moves forward. What does this mean?



How to show a language is not an LL(k) language?

The language $\{ a^n b \mid n \in \mathbf{N} \}$ is LL(1)

$S \rightarrow aS \mid b$

The language $\{ a^n b^{n+k} \mid n, k \in \mathbf{N} \}$ is LL(1)

$S \rightarrow AB \qquad A \rightarrow aAb \mid \Lambda \qquad B \rightarrow bB \mid \Lambda$

How to show a language is not an LL(k) language?

But, even with the following grammar, the language $\{a^{n+k}b^n \mid n,k \in \mathbf{N}\}$ is not LL(k) for any k

$S \rightarrow AB$; $A \rightarrow aA \mid \Lambda$; $B \rightarrow aBb \mid \Lambda$

Why?

Consider ab when k=1

Consider aabb when k=2

Consider aaabbb when k=3

...

But, even with the following grammar, the language $\{a^{n+k}b^n \mid n,k \in \mathbf{N}\}$ is not LL(k) for any k

$S \rightarrow AB$; $A \rightarrow aA \mid \Lambda$; $B \rightarrow aBb \mid \Lambda$

Consider a b *when* k=1

$S \Rightarrow AB$

$\Rightarrow aAB$

$\Rightarrow a\Lambda B$

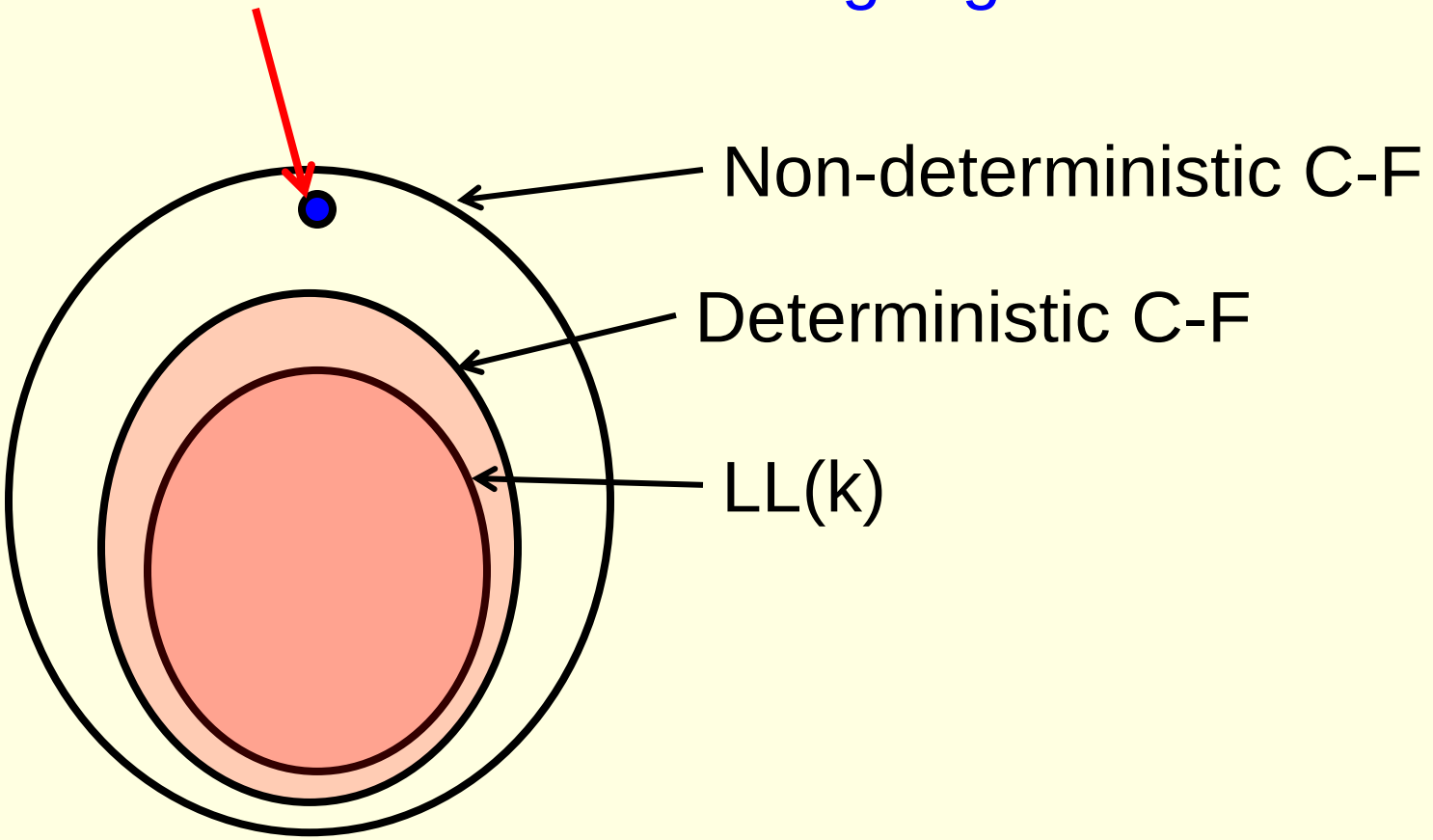
$\Rightarrow a\Lambda aBb$ *or* $\Rightarrow a\Lambda\Lambda$

Either way, we get stuck here

LL(k) languag s

A CF language is called a deterministic CF language if it can be recognized by a deterministic PDA

- LL(*k*) languages are deterministic CF languages,
- hence non-deterministic CF languages are not LL(k).

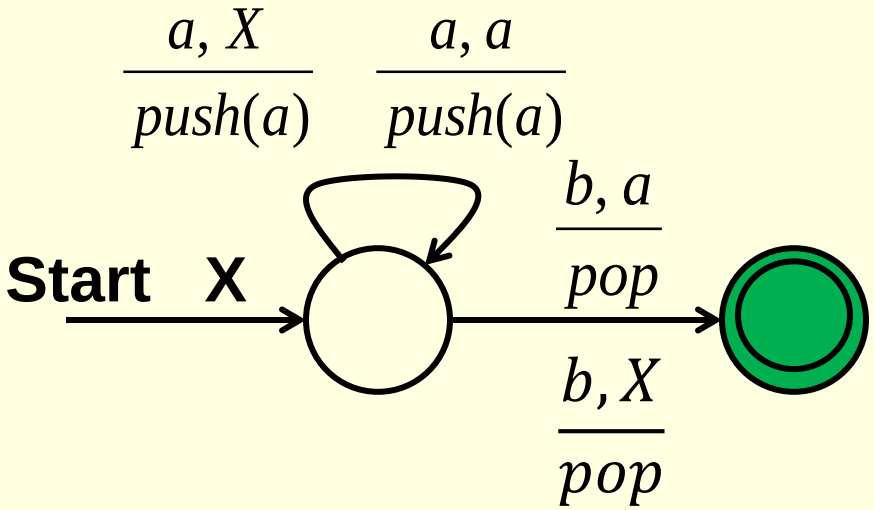


The language $\{ a^n b \mid n \in \mathbf{N} \}$ is LL(1)

$S \rightarrow aS \mid b$

Is the following PDA a PDA for this language?

Deterministic or non-deterministic?



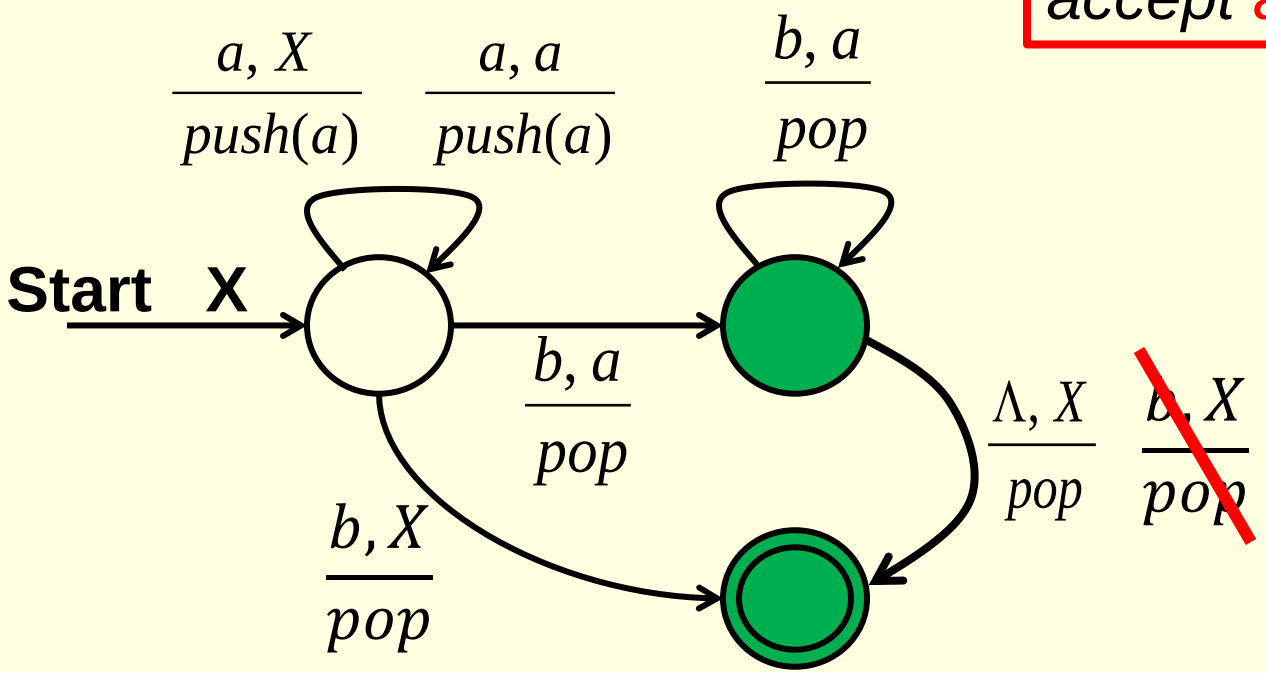
The language $\{ a^n b^{n+k} \mid n, k \in \mathbf{N} \}$ is LL(1)

$S \rightarrow AB \qquad A \rightarrow aAb \mid \Lambda \qquad B \rightarrow bB \mid \Lambda$

Is the following PDA a PDA for this language?

Deterministic or non-deterministic?

Does this PDA accept aabbbbb?



Example:

The language $\{ a^{n+k} b^n \mid k, n \in \mathbf{N} \}$ is nondeterministic context-free. So it has no LL(k) grammar for any k.

Proof:

Any PDA for the language must keep a count of the *a*'s with the stack so that when the *b*'s come along the stack can be popped with each *b*.

But there might still be *a*'s on the stack (when $k > 0$), so there must be a nondeterministic state transition to a final state from the popping state. i.e., we need two instructions like,

(*i*, *b*, *a*, pop, *i*) and (*i*, Λ, *a*, pop, *final*).

Illustration:

So why is it non-deterministic?

The language $\{ a^{n+k} b^n \mid k, n \in \mathbf{N} \}$ is nondeterministic context-free. So it has no LL(*k*) grammar for any *k*.

To pop the remaining a's off the stack

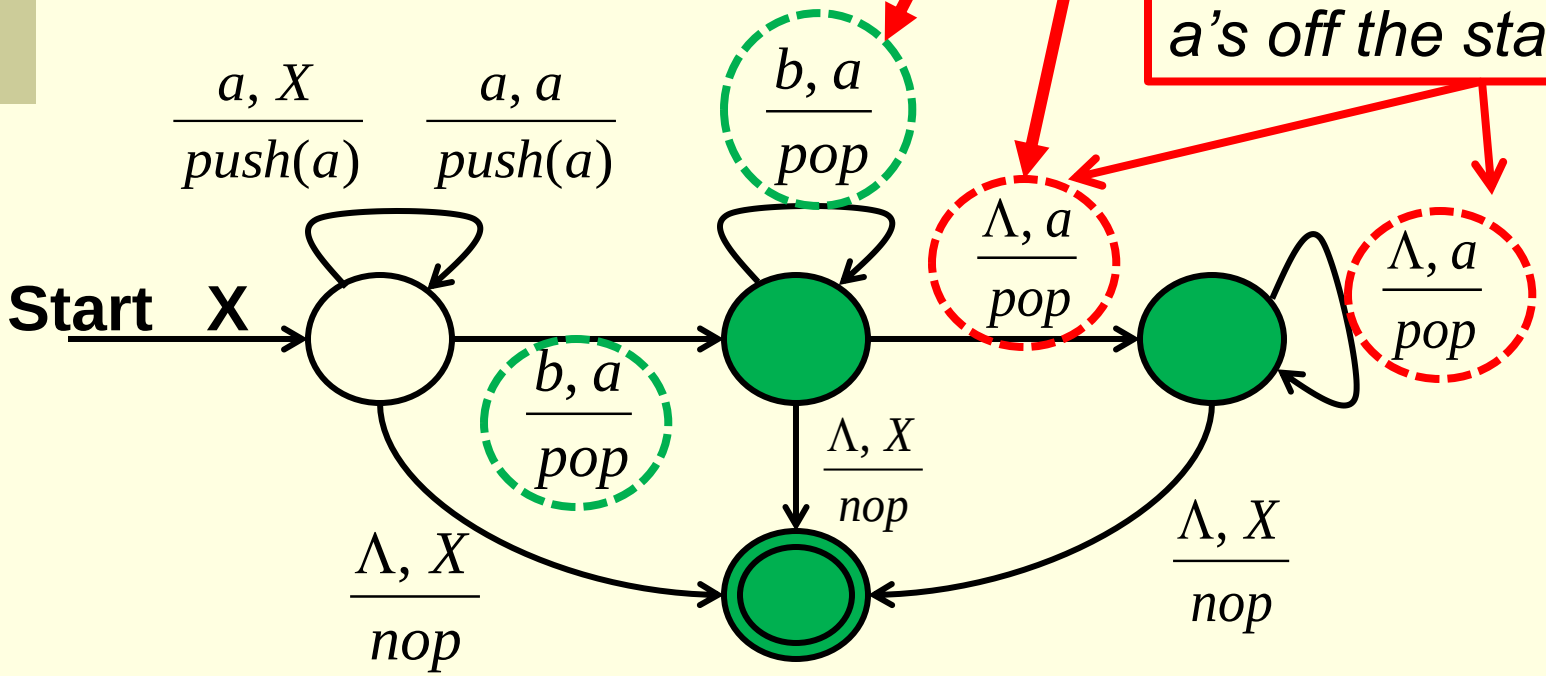
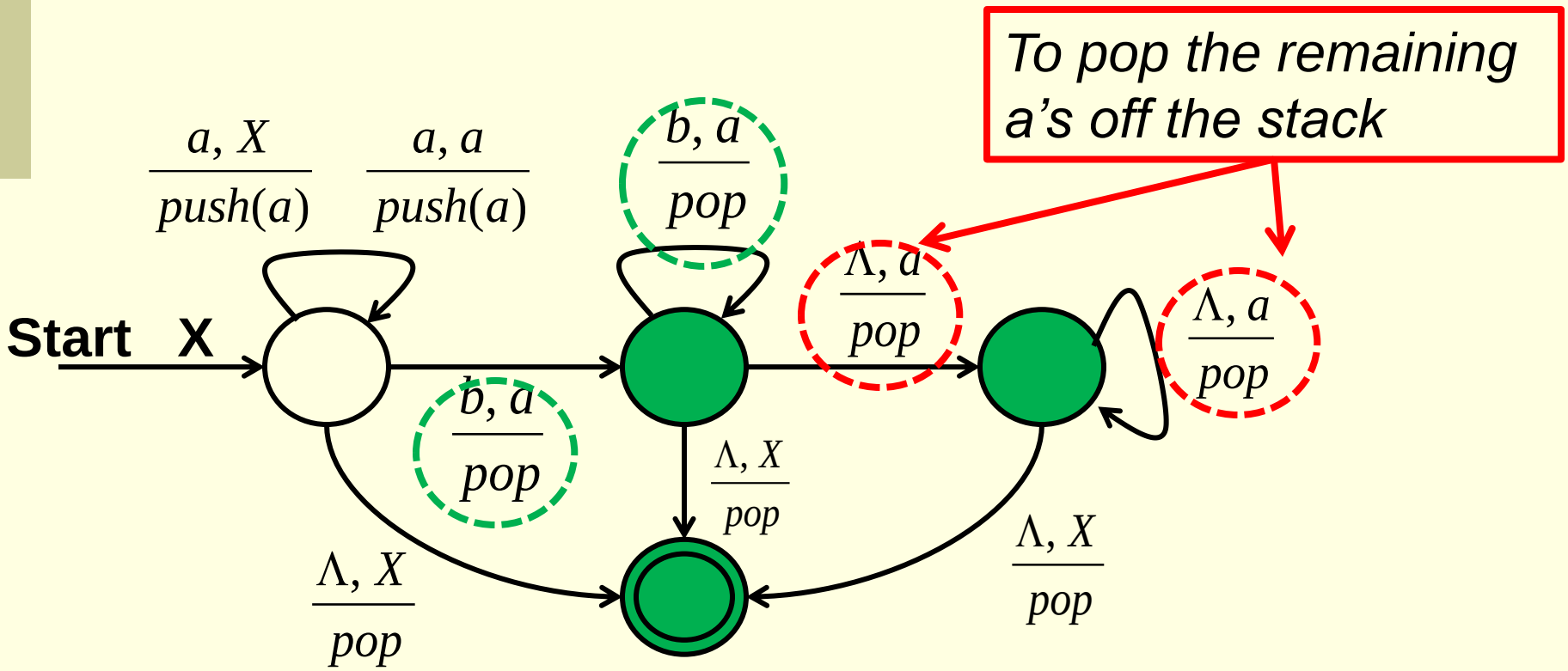


Illustration:

The language $\{ a^{n+k} b^n \mid k, n \in \mathbf{N} \}$ is nondeterministic context-free. So it has no LL(k) grammar for any k.

or



Question:

The language $\{ a^{n+k} b^n \mid k, n \in \mathbf{N} \}$ is nondeterministic context-free. (So it has no LL(k) grammar for any k)

then

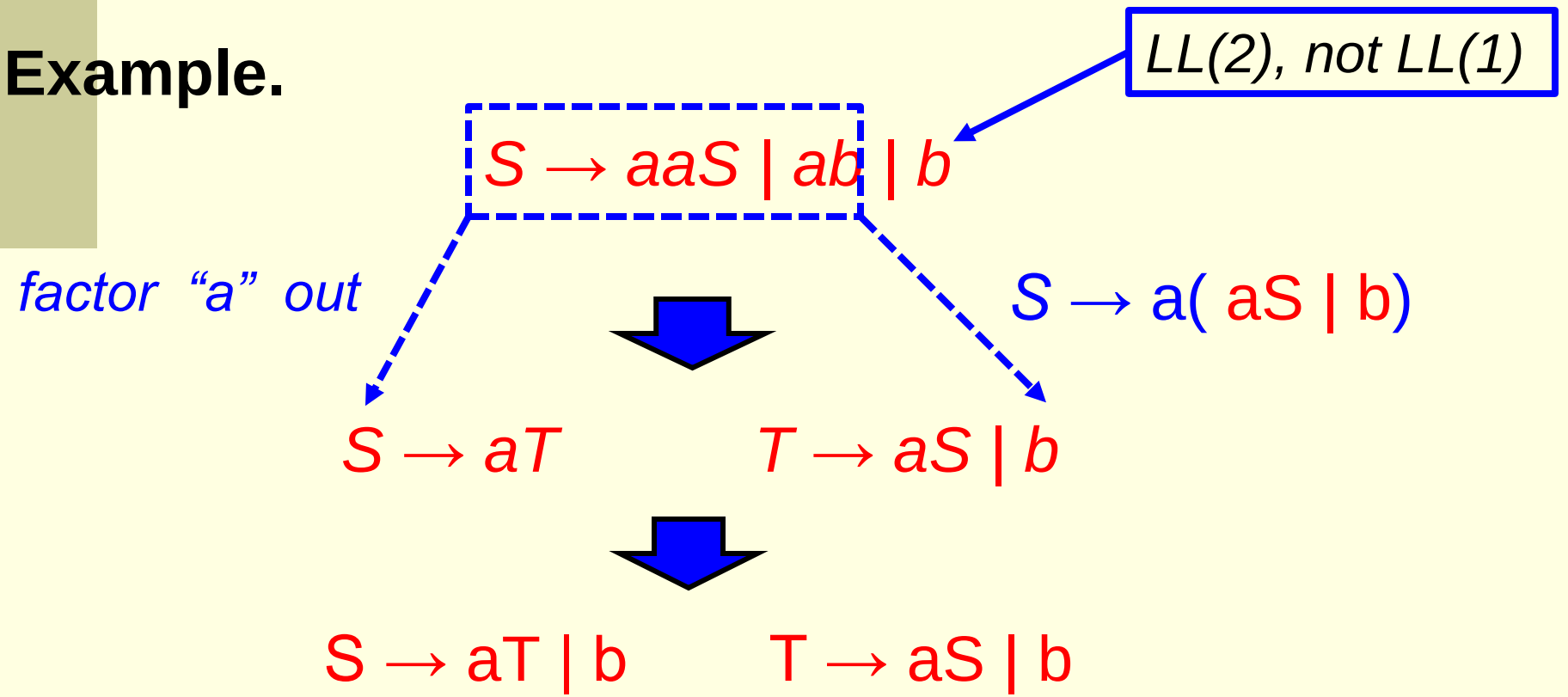
Is the language $\{ a^{n+k} b^n c^k \mid k, n \in \mathbf{N} \}$ nondeterministic context-free?

Keep your answer to yourself

Grammar Transformations:

"Left-factoring" an LL(*k*) grammar to obtain an equivalent LL(*n*) grammar where $n < k$.

Example.



Question:

Is { S → aT | b T → aS | b } an LL(1) grammar for the language { $a^n b$ | n ϵ N} ?

Consider: b

$S \Rightarrow b$

Consider: a b

$S \Rightarrow aT$

$\Rightarrow ab$

Consider: a a b

$S \Rightarrow aT$

$\Rightarrow aaS$

$\Rightarrow aab$

Yes or No

Question:

{ S → aaS | ab | b } is an LL(2) grammar for the language { $a^n b$ | n ϵ N}

{ S → aT | b T → aS | b } is an LL(1) grammar for the language { $a^n b$ | n ϵ N}

Try aaaab and get its parse trees in both cases

End of Context-Free Language and Pushdown Automata III