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 2-5: Regular Languages, Finite Automata (Chapter 11)**
- **Weeks 6-8: Context-Free Languages, Pushdown Automata (Chapters 12)**
- **Weeks 9-11: Turing Machines (Chapter 13)**

Table of Contents (conti):

- **Weeks 12-13: Propositional Logic (Chapter 6), Predicate Logic (Chapter 7), Computational Logic (Chapter 9), Algebraic Structures (Chapter 10)**

Factorization has two effects:

*(1) Changes the **1st derivation step** to a **unique step** (in most of the cases);*

*(2) it does not change the number of options for the 2nd step, but it removes the common factor in all the options so the **lookahead box** can be a smaller box.*

Another example of Grammar Transformation:

LL(3)?

Find an LL(k) grammar where k is as small as possible that is equivalent to the following grammar.

S → abS | abcT | ab   T → cT | c

S → abS | abcT | ab   T → cT | c

S → ab(S | cT | Λ)   T → c(T | Λ)

S → abR   R→ S | cT | Λ   T → cU   U → T | Λ

LL(1) ?

Another example of Grammar Transformation:

LL(3)?

Find an LL(*k*) grammar where *k* is as small as possible that is equivalent to the following grammar.

S → abS | abcT | ab   T → cT | c

⇩

S → abR   R→ S | cT | Λ   T → cU   U → T | Λ

LL(1) ?

Step 1   *Step 2*

Find an LL(*k*) grammar where *k* is as small as possible that is equivalent to the following grammar.

S → abS | abcT | ab   T → cT | c

LL(3)?

First, the language generated by the grammar is

{ $(ab)^n$, $(ab)^n c^m$ | n ≥ 1, m ≥ 2 }

Is this grammar LL(3)?

$S \Rightarrow abS \Rightarrow ababS \Rightarrow \cdots \Rightarrow (ab)^{n-1}S \Rightarrow (ab)^{n-1}ab = (ab)^n$

$S \Rightarrow abS \Rightarrow ababS \Rightarrow \cdots \Rightarrow (ab)^{n-1}S \Rightarrow (ab)^{n-1}abcT = (ab)^n cT$
$\Rightarrow (ab)^n ccT \Rightarrow \cdots \Rightarrow (ab)^n c^{m-1}T \Rightarrow (ab)^n c^{m-1}c = (ab)^n c^m$

Is $\begin{pmatrix} S \to abR & T \to cU \\ R \to S \mid cT \mid \Lambda & U \to T \mid \Lambda \end{pmatrix}$ LL(1)

for $\{ (ab)^n, (ab)^n c^m \mid n \geq 1, m \geq 2 \}$?

YES

Convert $\begin{pmatrix} S \to abR & T \to cU \\ R \to S \mid cT \mid \Lambda & U \to T \mid \Lambda \end{pmatrix}$

to

$$\begin{pmatrix} S \to abR & T \to cU \\ R \to abR \mid cT \mid \Lambda & U \to cU \mid \Lambda \end{pmatrix}$$

then prove

Convert

$$\begin{pmatrix} S \to abR & T \to cU \\ R \to S \mid cT \mid \Lambda & U \to T \mid \Lambda \end{pmatrix}$$

to

$$\begin{pmatrix} S \to abR & T \to cU \\ R \to abR \mid cT \mid \Lambda & U \to cU \mid \Lambda \end{pmatrix}$$

then prove

Factorization has two effects:

(1) changes the 1st derivation step to a unique step;

(2) it does not change the # of options for the 2nd step, but it removes the common factor in all the options so lookahead box can be a smaller box.

Most importantly:

If the size of the factor is ‘n’, then the grammar is reduced from an LL(k) grammar to an LL(k-n) grammar.

Question: Since each string of the language $\{ (ab)^n, (ab)^n c^m \mid n \geq 1, m \geq 2 \}$ would have two different parse trees now, one with respect to the old grammar, one with respect to the new grammar, does this mean the language is ambiguous?

There is no such thing as an ambiguous language, but an ambiguous grammar.

If the old grammar is not ambiguous, then the new grammar would still be un-ambiguous.

If the old grammar is ambiguous, then the new grammar would also be ambiguous.

Why?

For instance:

LL(3)?

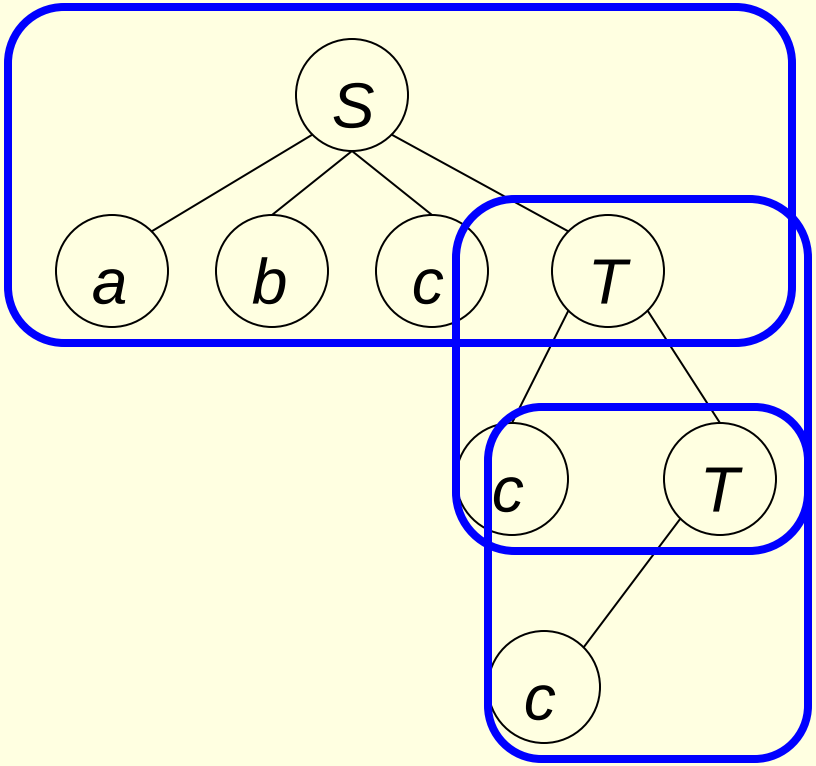
$S \rightarrow abS \mid abcT \mid ab \qquad T \rightarrow cT \mid c$

For abccc we have

$S \Rightarrow abcT$

$\Rightarrow abccT$

$\Rightarrow abccc$



For instance:

LL(1)?

S → abR   R → S | cT | Λ   T → cU   U → T | Λ

For abccc, we have

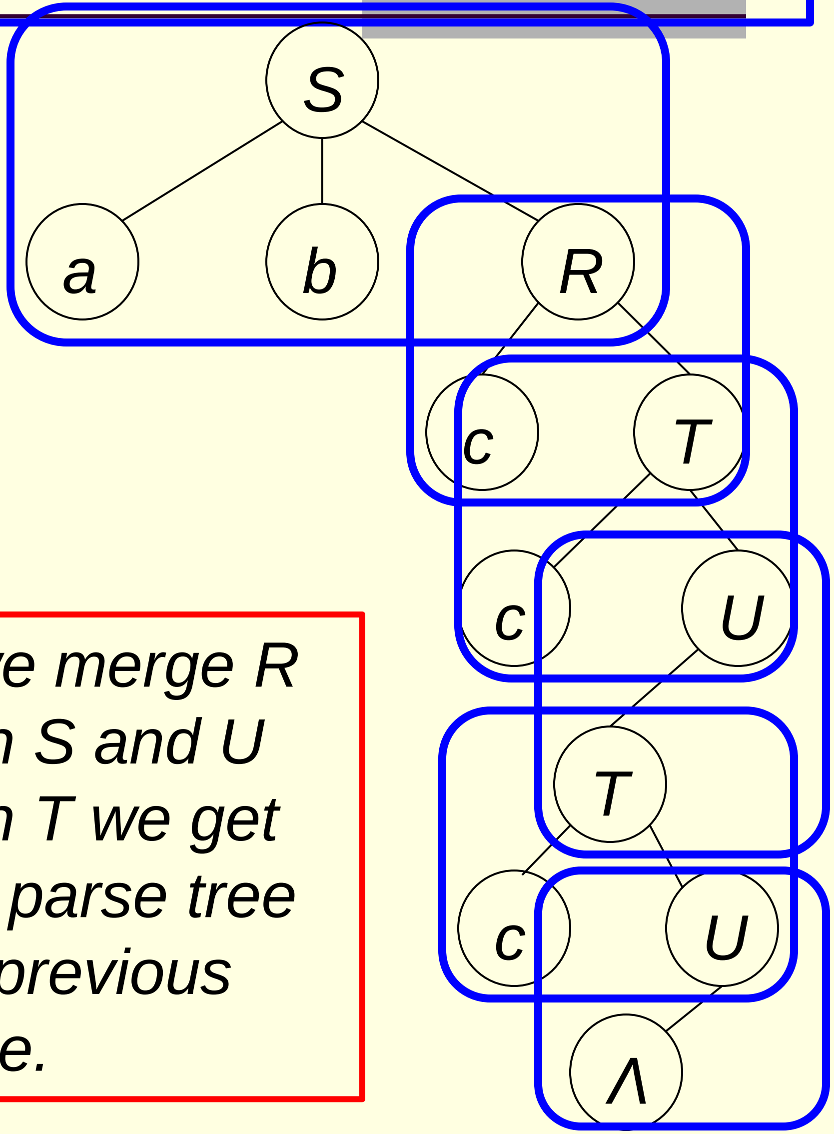
S ⇒ abR

⇒ abcT

⇒ abccU

⇒ abccT

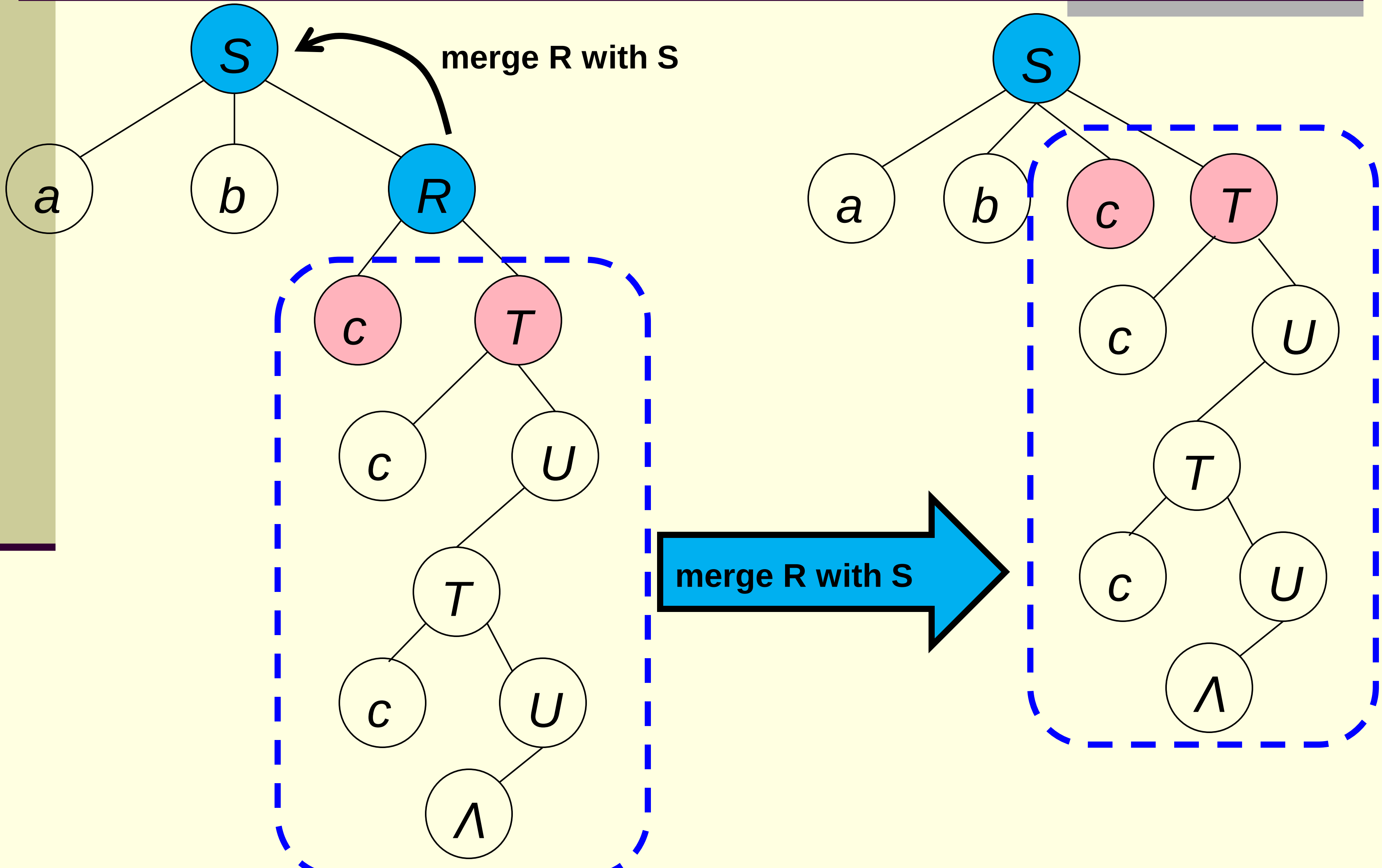
⇒ abcccU ⇒ abcccΛ



If we merge R with S and U with T we get the parse tree on previous slide.

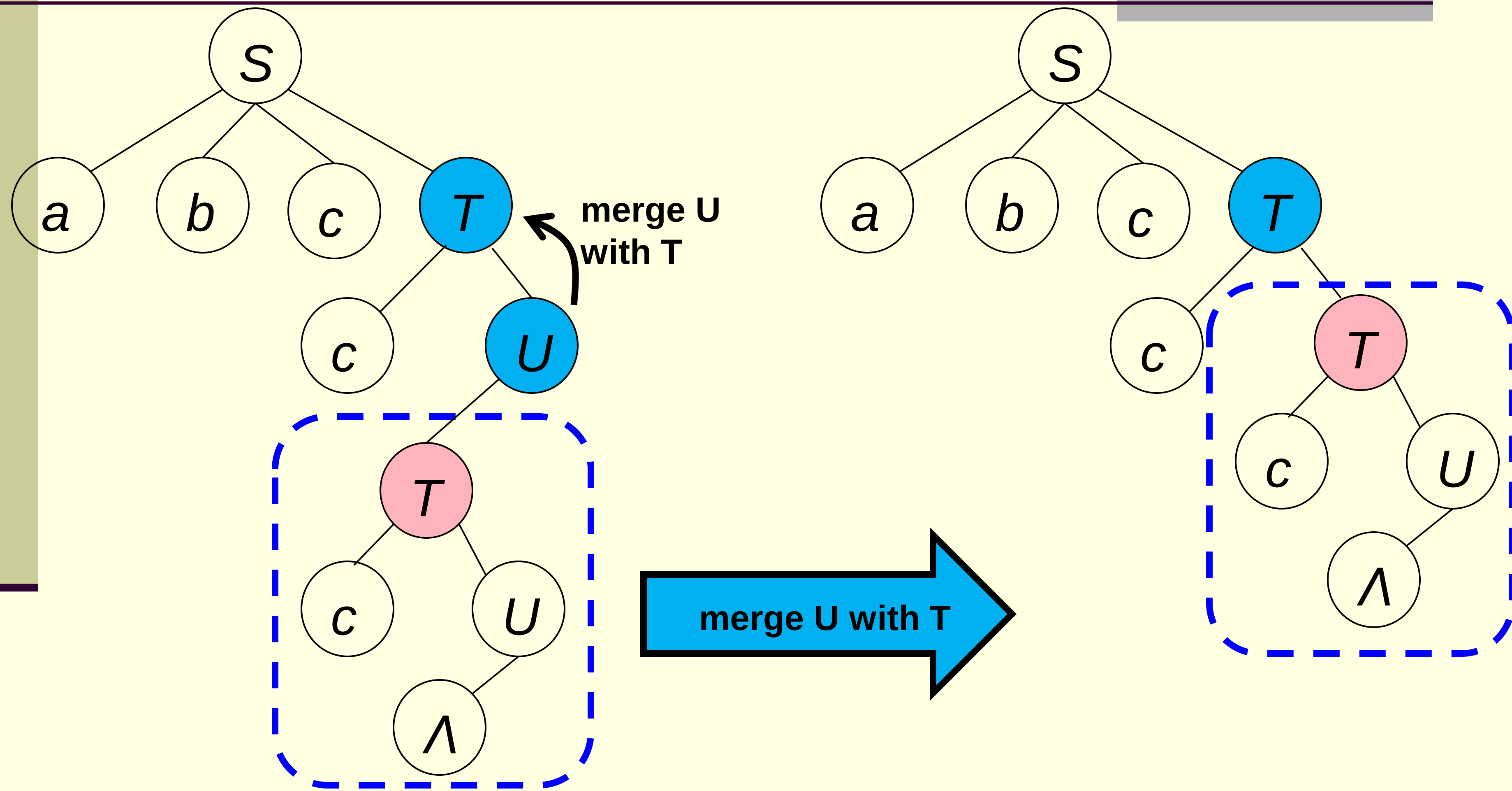
S → abR   R→ S | cT | Λ   T → cU   U → T | Λ

For abccc, we have



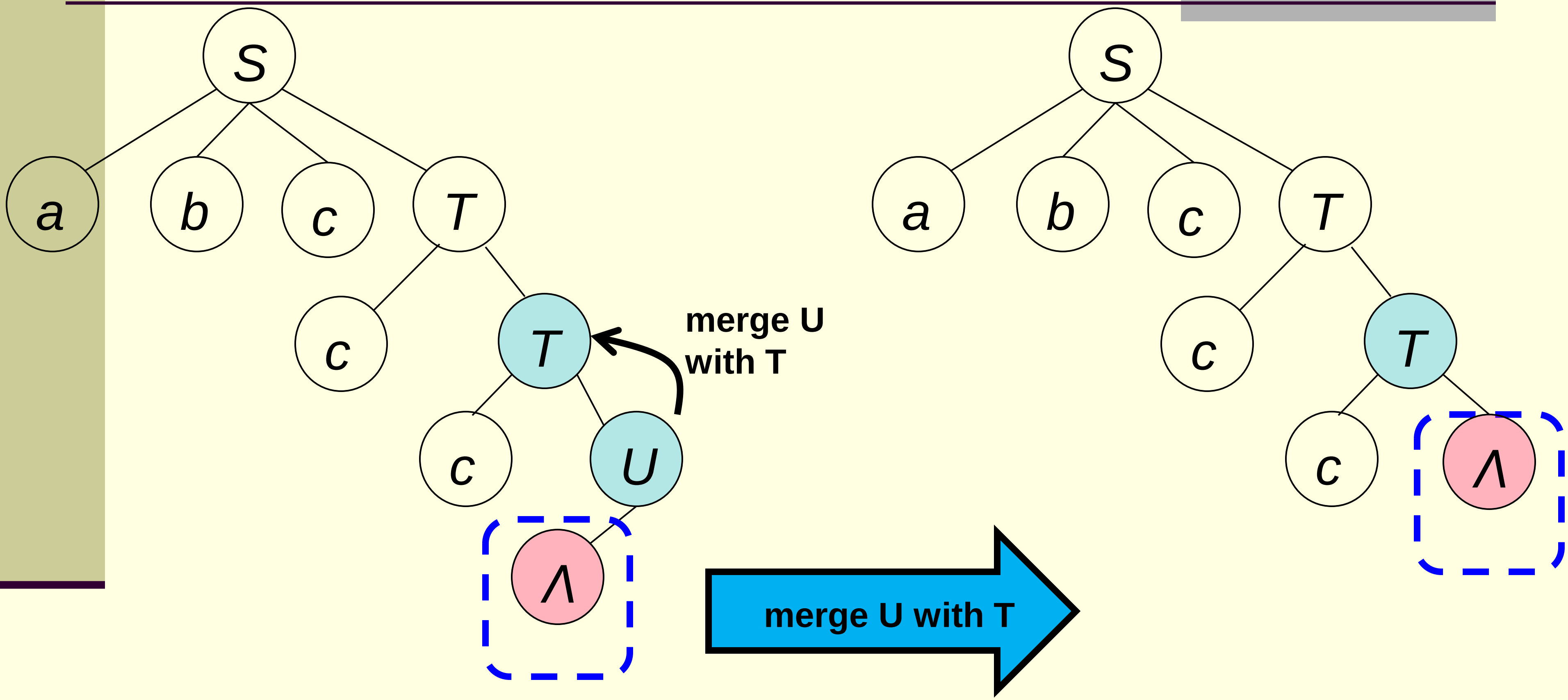
S → abR   R → S | cT | Λ   T → cU   U → T | Λ

For abccc, we have



S → abR   R→ S | cT | Λ   T → cU   U → T | Λ

For abccc, we have



New grammar

S → abR　　R → S | cT | Λ

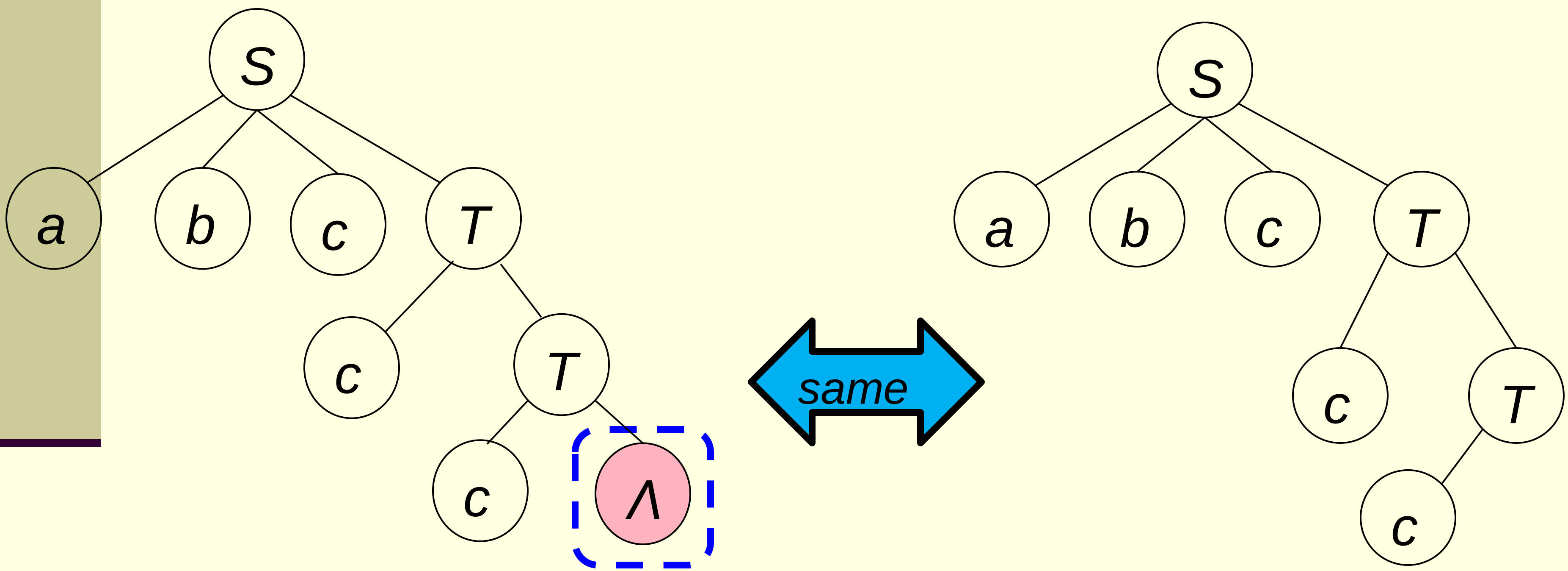
T → cU　　U → T | Λ

Old grammar

S → abS | abcT | ab

T → cT | c

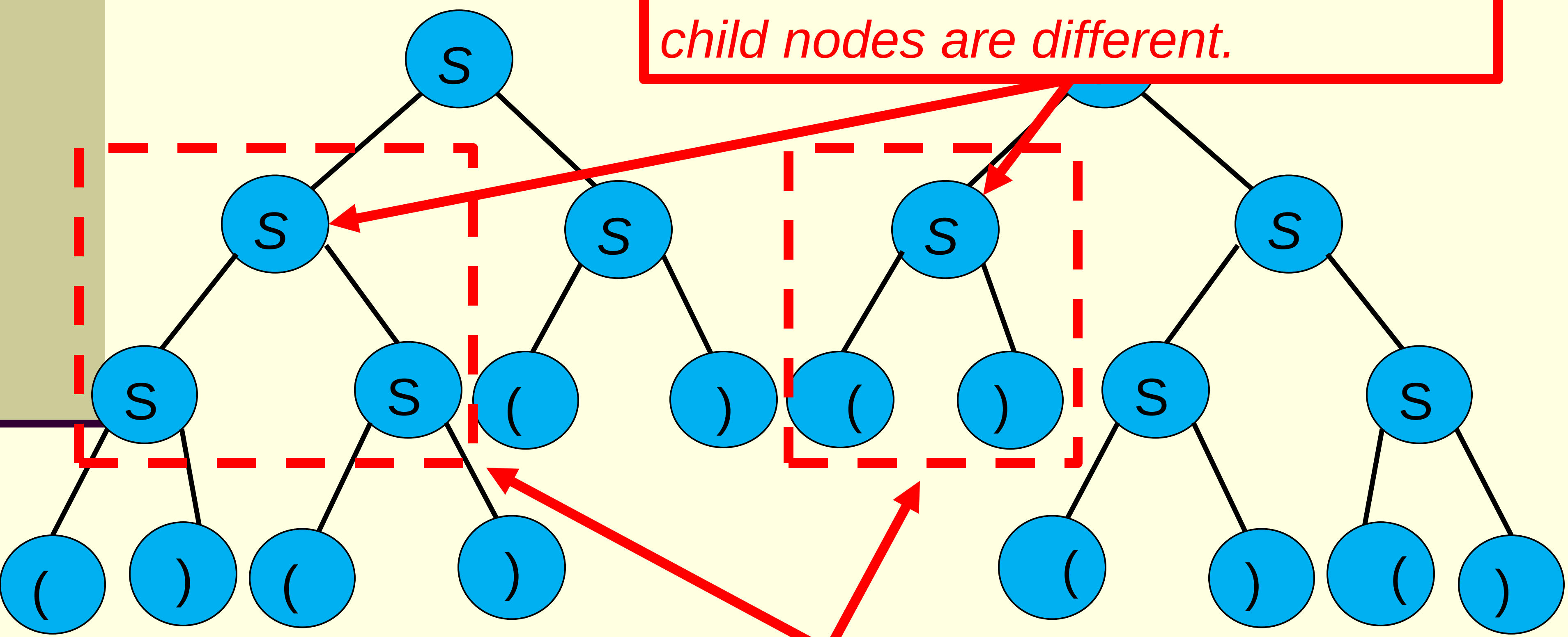
After merging, we have　　　　*For abccc, we have*



So, what does this mean? it means the parse tree of a string with respect to the old grammar can be converted to the parse tree of that string with respect to the new grammar (and vice versa)

If old grammar is ambiguous then the new grammar is ambiguous too. Why?

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



Then the corresponding internal nodes (the node itself or one of its non-leaf child nodes) in the parse trees generated by the new grammar would be different too.

Why?

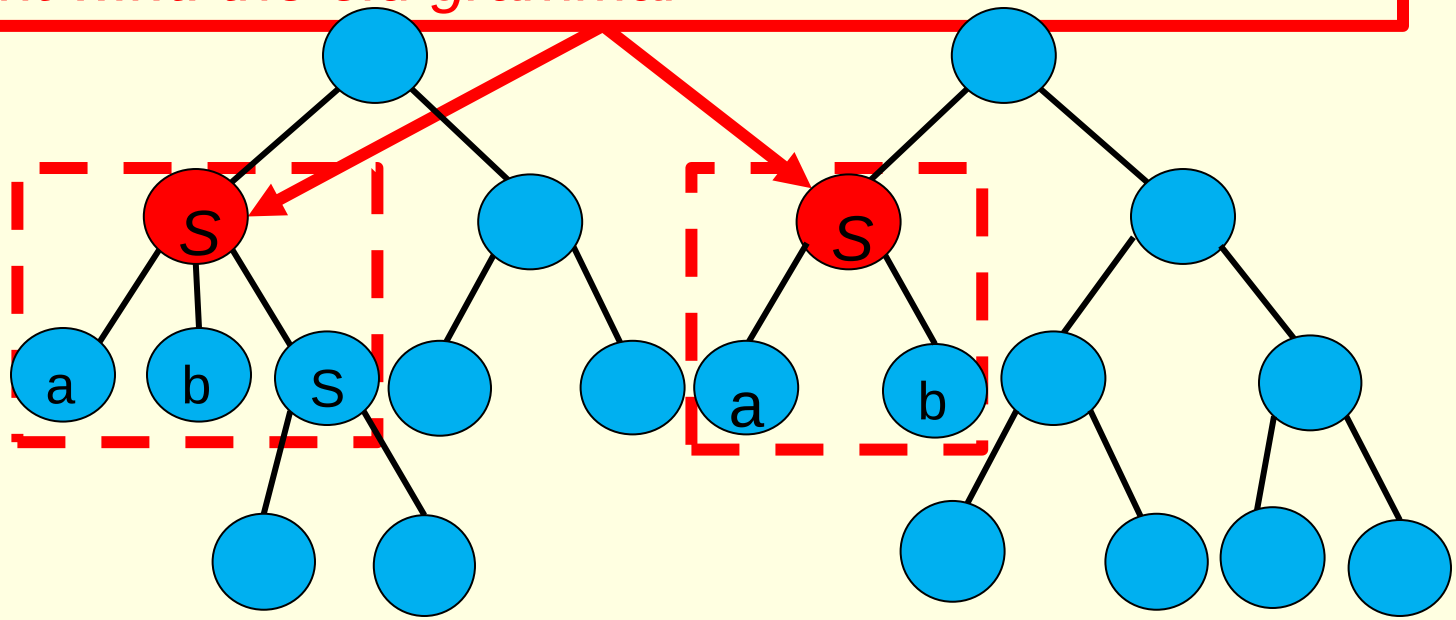
Consider the case that

$S \rightarrow abS|abcT|ab \quad T \rightarrow cT|\Lambda$

After factorization, we get

$S \rightarrow abR \quad R \rightarrow S|cT|\Lambda \quad T \rightarrow cT|\Lambda$

If S is the first internal node whose child nodes are different w.r.t. the old grammar



Why?

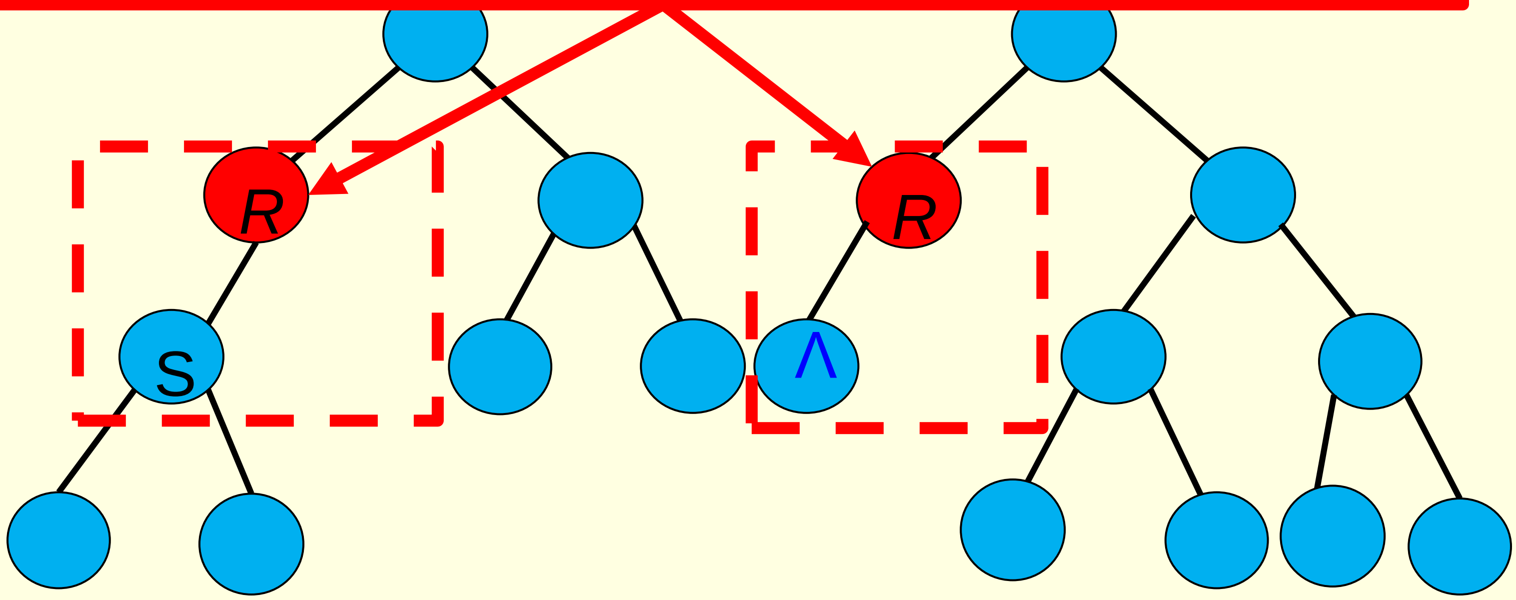
Consider the case that

$S \rightarrow abS|abcT|ab \quad T \rightarrow cT|\Lambda$

After factorization, we get

$S \rightarrow abR \quad R \rightarrow S|cT|\Lambda \quad T \rightarrow cT|\Lambda$

Then R would be the first internal node whose child nodes are different w.r.t. the new grammar



Remove Left Recursion:

A grammar is left-recursive if it has a derivation of the form

$$A \Rightarrow^{+} Ax$$

for some nonterminal A and sentential form x.

***Example*.** The language $\{ ba^n \mid n \in \mathbf{N} \}$ has a grammar

$$S \to Sa \mid b$$

that is left-recursive.

$$S \Rightarrow^{+} Sa^n$$

Remove Left Recursion:

Left-recursive grammars are not LL(k) for any *k*

For instance, the grammar $S \to Sa \mid b$ for the language $\{ ba^n \mid n \in \mathbf{N} \}$ is not LL(k) for any k.

WHY?

LL(1) case: Consider: b a

$S \Rightarrow ?$

LL(2) case: Consider: b a a

$S \Rightarrow Sa$

$\Rightarrow ?$

Are there any a's to the right of this 'a'?

Remove Left Recursion:

Left-recursive grammars are not LL(k) for any *k*

For instance, the grammar S → Sa | b for the language $\{ ba^n \mid n \in \mathbf{N} \}$ is not LL(k) for any k.

WHY?

LL(3) case:

Consider: b a a a

Are there any a's to the right of this 'a'?

$S \Rightarrow Sa$

$\Rightarrow Saa$

$\Rightarrow$?

Left-recursive grammars are not LL(k) for any *k*

WHY?

Growth direction ←

Left-recursive:

$$S \Rightarrow Sx_1 \Rightarrow Sx_2x_1 \Rightarrow Sx_3x_2x_1 \Rightarrow \cdots$$
$$\Rightarrow Sx_7\ x_6 \cdots x_3\ x_2\ x_1$$

Scan/matching direction →

Input string: $x_7\ x_6\ x_5\ x_4\ x_3\ x_2\ x_1$ *LL(2)*

How would you be able to tell what production(s) to use for the generation of x_7x_6 while we don't have information on x_5x_4 yet.

Left-recursive grammars are not LL(k) for any *k*

WHY?

Growth direction

Left-recursive:

$$S \Rightarrow Sx_1 \Rightarrow Sx_2x_1 \Rightarrow Sx_3x_2x_1 \Rightarrow \cdots$$
$$\Rightarrow S\, x_7\, x_6 \cdots x_3\, x_2\, x_1$$

Scan/matching direction

Input string: $x_7\, x_6\, x_5\, x_4\, x_3\, x_2\, x_1$ *LL(7)*

But what if the length of the input string is 9, 10 or 20?

Left-recursive grammars are **bad** for parsing

WHY?

Left-recursive grammar: $S \rightarrow Sa \mid b$ *for* $\{ ba^n \mid n \in \mathbf{N} \}$

Input string: *baaa*

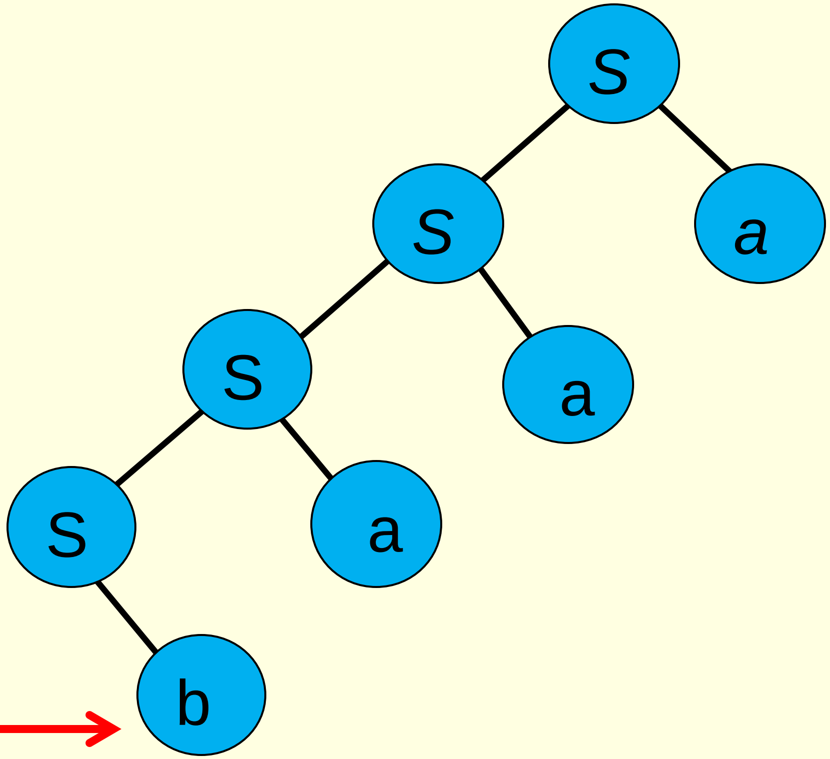
String generation:

$S \Rightarrow Sa$

$\Rightarrow Saa$

$\Rightarrow Saaa$

$\Rightarrow baaa$



When 'b' is scanned, how do we know it is a 'b' from baaa or from baa, or from ba?

Left-recursive grammars are not LL(k) for any *k*

Left-recursive grammars are **bad** for parsing

Root, left, right

The parse tree is supposed to be built in a ***top-down*** *fashion (or, the symbols in the input string are supposed to be matched with the leaf nodes in a pre-order fashion) and, yet, for a left-recursive grammar, the order is reversed.*

Bottom-up

Remove Direct Left Recursion:

Obtain an LL(k) grammar by removing left-recursion

Consider: $A \to Aw \mid Au \mid Av \mid a \mid b$

One gets avuw through the following derivation:

$A \Rightarrow Aw \Rightarrow Auw \Rightarrow Avuw \Rightarrow avuw$

So you obtain avuw this way = ((((a) v) u) w)

Since (a (v (u (w))))= ((((a) v) u) w)

One can also get avuw the following way:

$A \Rightarrow aB \Rightarrow avB \Rightarrow avuB \Rightarrow avuwB \Rightarrow avuw\Lambda = avuw$

Remove Direct Left Recursion:

(Change end point to start point, change start point to end point)

Algorithm for removing Direct Left Recursion:

Terminating tools

Left recursion

Transform: $A \rightarrow Aw \mid Au \mid Av \mid a \mid b$

To: $A \rightarrow aB \mid bB$

$B \rightarrow wB \mid uB \mid vB \mid \Lambda$

Right recursion

Terminating tool

Remove Direct Left Recursion:

Example: removing left-recursion of $S \rightarrow Sa \mid b$

Transform:

$S \rightarrow b$

$S \rightarrow Sa$

To:

$S \rightarrow bB$

$B \rightarrow aB \mid \Lambda$

Remove Direct Left Recursion:

$S \to Sa \mid b$ is not LL(1),

but $S \to bB \quad B \to aB \mid \Lambda$ is LL(1)

Consider: b a a a

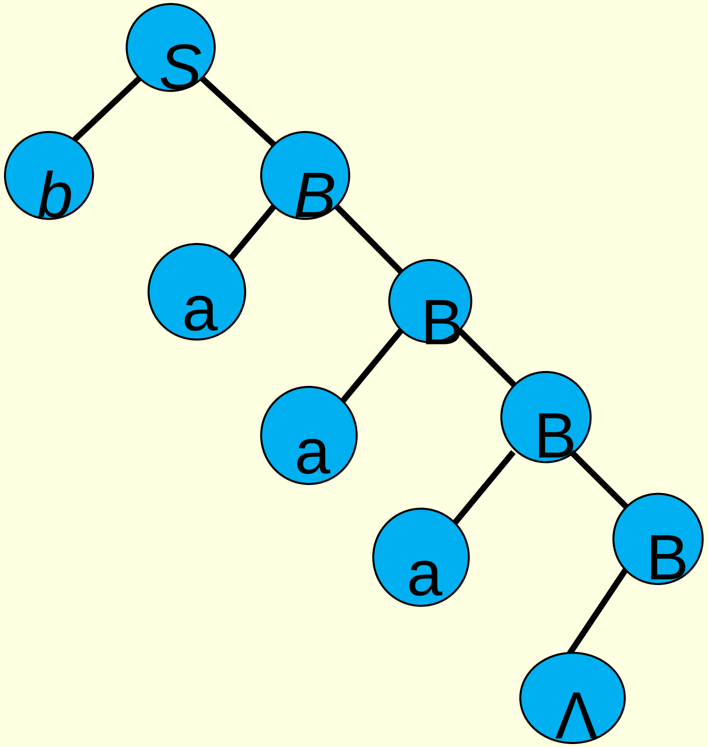
$S \Rightarrow bB$

$\Rightarrow b\,a\,B$

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

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

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



Fundamental difference between left-recursive grammars and right-recursive grammars:

Right-recursive:

Growth direction

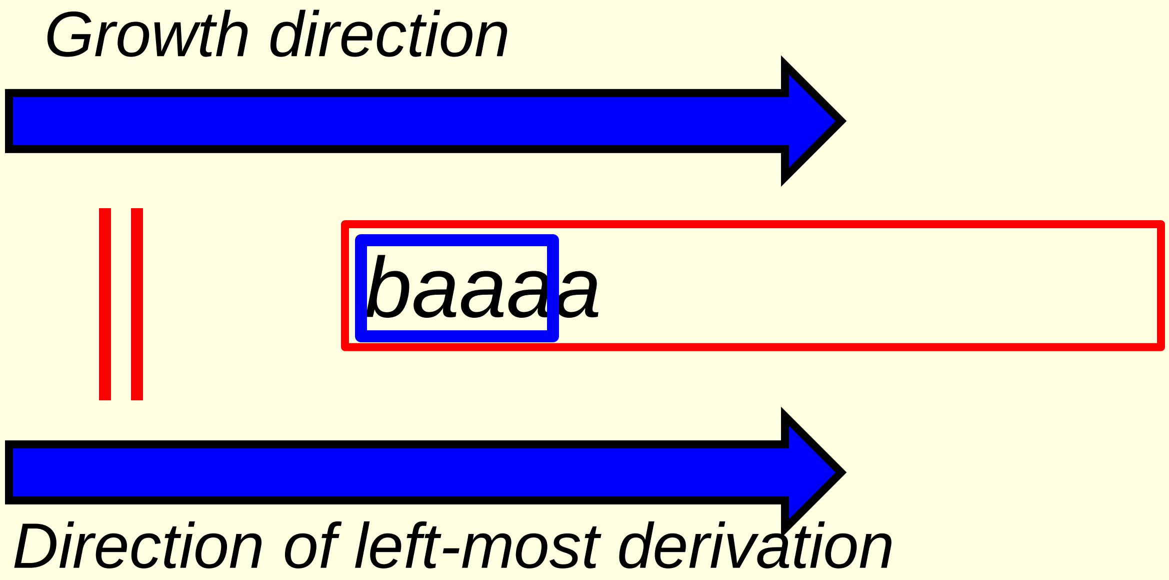
baaaa…

$$S \Rightarrow b\color{red}{B}$$
$$\Rightarrow ba\color{red}{B}$$
$$\Rightarrow baa\color{red}{B}$$

Direction of left-most derivation

Fundamental difference between left-recursive grammars and right-recursive grammars:

Right-recursive:



The left-most derivation process knows all the previous foot steps of the string growing process

Fundamental difference between left-recursive grammars and right-recursive grammars:

Left-recursive:

Growth direction ←

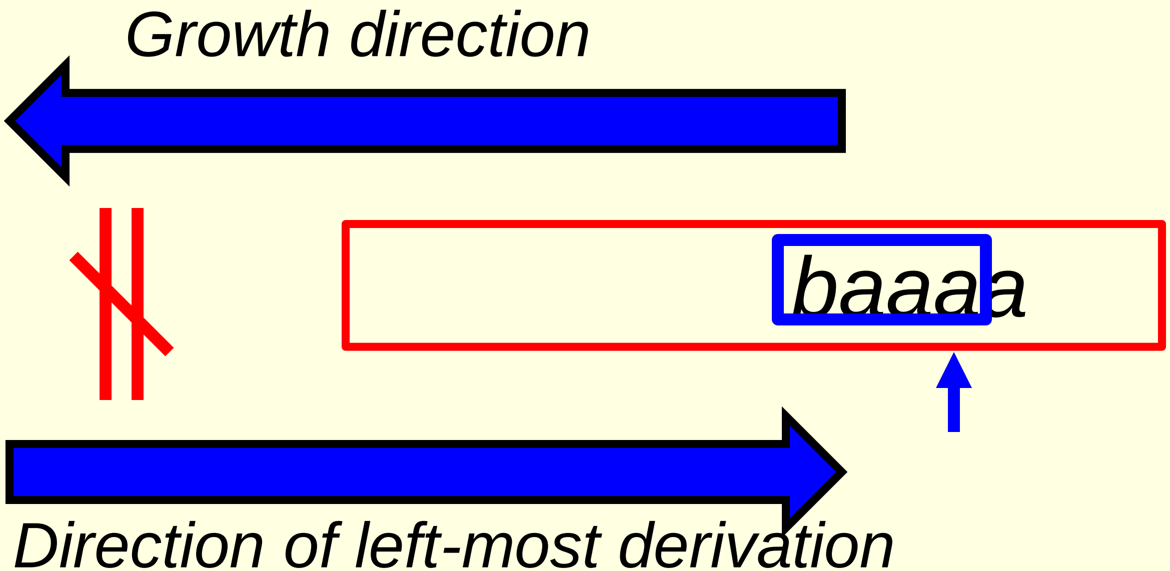
baa

$$S \Rightarrow A\,a$$
$$\Rightarrow A\,aa$$
$$\Rightarrow baa$$

→ *Direction of left-most derivation*

Fundamental difference between left-recursive grammars and right-recursive grammars:

Left-recursive:



The left-most derivation process does not know the previous foot steps of the string growing process.

Remove Direct Left Recursion:

Example: removing left-recursion of

S → Saa | aab |aac

Transform:

S → aab | aac

S → Saa

To:

S → aabB | aacB

B → aaB | Λ

Convert productions that are used as terminating tools first

Remove Direct Left Recursion:

$S \rightarrow Saa \mid aab \mid aac$ is not LL(3),

but $S \rightarrow aabB \mid aacB$ $\quad B \rightarrow aaB \mid \Lambda$ is LL(3)

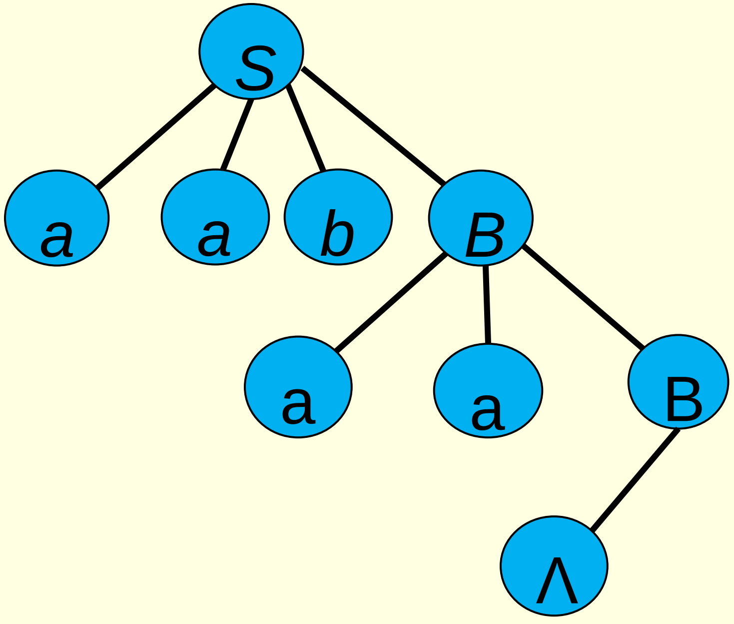
Consider: a a b a a

$S \Rightarrow aabB$

$\Rightarrow$ a a b a a B

$\Rightarrow$ a a b a a Λ

= a a b a a



A right-recursive grammar is always LL(k) for some *k*

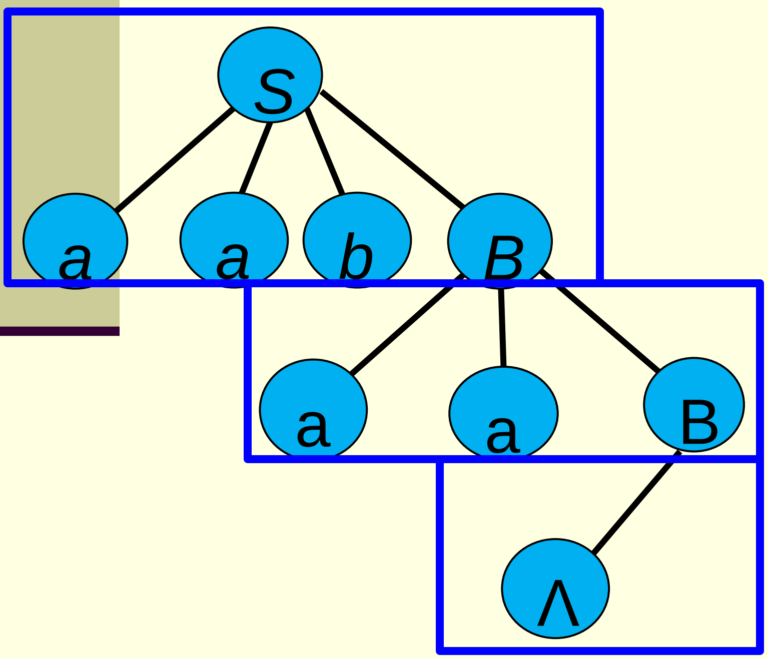
$S \rightarrow aabB \mid aacB \qquad B \rightarrow aaB \mid \Lambda$

WHY?

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

Right-recursive:

$S \Rightarrow x_1 S \Rightarrow x_1 x_2 S \Rightarrow x_1 x_2 x_3 S \Rightarrow \cdots$
$\Rightarrow x_1\, x_2\, x_3 \cdots x_{n-1}\, x_n\, S$



So no matter how big the input string is, one can always find a k large enough so that if k symbols are read each time, one can always tell if the right most symbol of the lookahead box is the result of some m-th derivation step

Use factorization to make a right recursive grammar more efficient:

Rewrite $S \rightarrow aabB \mid aacB \quad B \rightarrow aaB \mid \Lambda$ to be LL(1)

Note that $S \rightarrow aabB \mid aacB$ = $S \rightarrow aa(bB \mid cB)$

Transform: $S \rightarrow aabB \mid aacB \quad B \rightarrow aaB \mid \Lambda$

To:

$S \rightarrow aaA$

$A \rightarrow bB \mid cB$

$B \rightarrow aaB \mid \Lambda$

← Show this is LL(1)

Use factorization to make a right recursive grammar more efficient:

Transform: $S \rightarrow aabB \mid aacB \qquad B \rightarrow aaB \mid \Lambda$

To:

$S \rightarrow aaA$

$A \rightarrow bB \mid cB$ ← LL(1)

$B \rightarrow aaB \mid \Lambda$

Factorization makes ‘$S \rightarrow aaA$’ a unique first production. This production will always be used as the first step in the derivation step (you don’t need to scan anything), and the common factor on the right side of this production provides an automatic match on the first part of the input string (e.g., for aabaa, you automatically get a match on aa from the first production), so you can start your first scan on the third or fourth symbol and yet with a smaller lookahead box.

Remove Indirect Left Recursion:

$S \to Ab \mid a \qquad A \to Sa \mid b$ is left recursive

(Because $S \Rightarrow Ab \Rightarrow Sab$)

To remove indirect left recursion:

1. Replace A in $S \to Ab$ by the right side of $A \to Sa \mid b$
2. Then remove the left recursion

Remove Indirect Left Recursion:

Step 1:

$S \rightarrow Ab \mid a \qquad A \rightarrow Sa \mid b$

$S \rightarrow Sab \mid bb \mid a$

Step 2:

$S \rightarrow bbB \mid aB$

$B \rightarrow abB \mid \Lambda$

Remove Indirect Left Recursion:

Example: remove left recursion from

$S \rightarrow Ab \mid a \qquad A \rightarrow SAa \mid b$

Step 1:

$S \rightarrow SAab \mid bb \mid a \qquad A \rightarrow SAa \mid b$

Step 2:

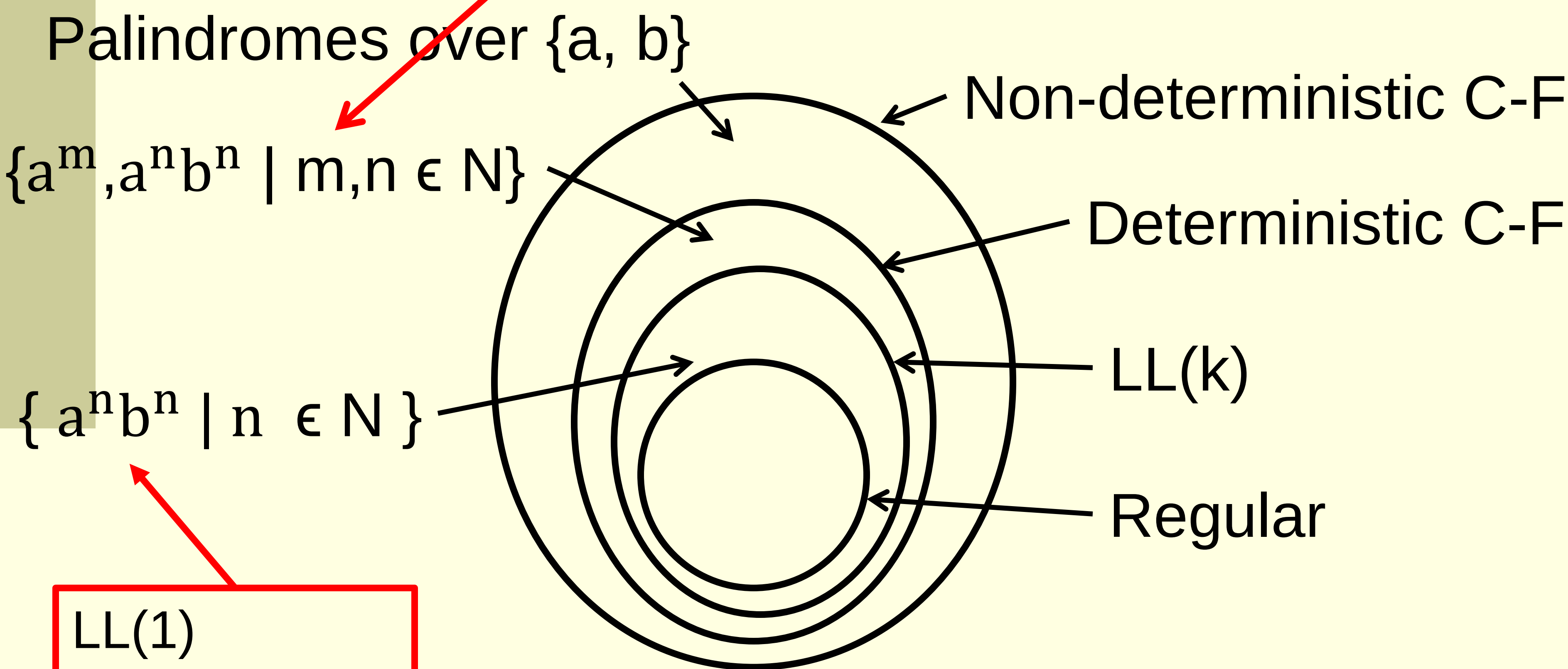
$S \rightarrow bbB \mid aB \qquad B \rightarrow AabB \mid \Lambda \qquad A \rightarrow SAa \mid b$

The Picture:

Non-deterministic

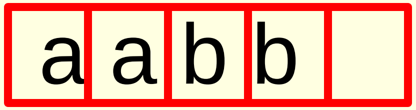
S → A | B
B → aB | Λ
A → aAb | Λ

Skip slides 44-55



LL(1)
S → aSb | Λ

The grammar $\{ S \to aSb \mid \Lambda \}$ for the language $\{ a^n b^n \mid n \in \mathbf{N}\}$ is LL(1)

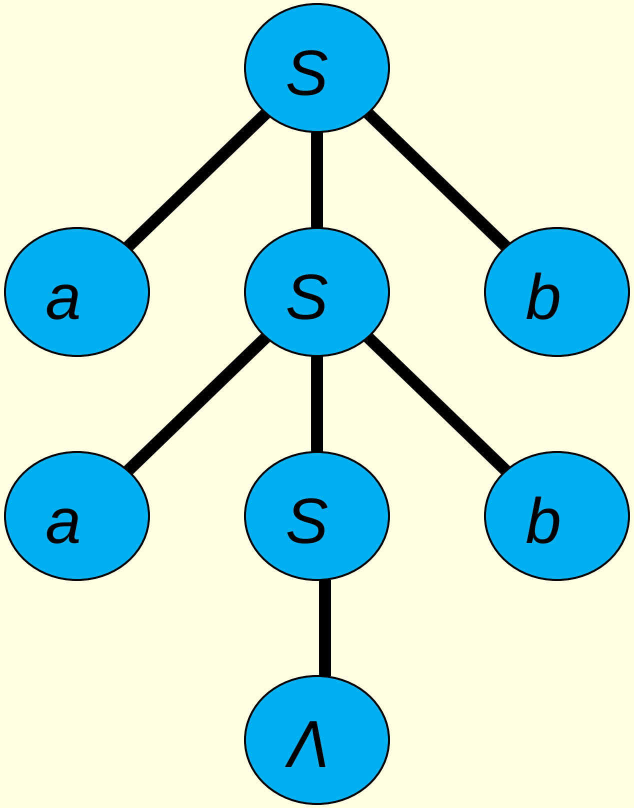
Consider 

$S \Rightarrow aSb$

$\Rightarrow aaSbb$

$\Rightarrow aa\Lambda bb$

Q.E.D.



The grammar

$\{ S \rightarrow A \mid B \quad A \rightarrow aAb \mid \Lambda \quad B \rightarrow aB \mid \Lambda \}$

for the language $\{ a^m, a^n b^n \mid m, n \in \mathbf{N} \}$

is not LL(k) for any k

For k=1, consider: a

For k=2, consider: aa

For k=3, consider: aaa

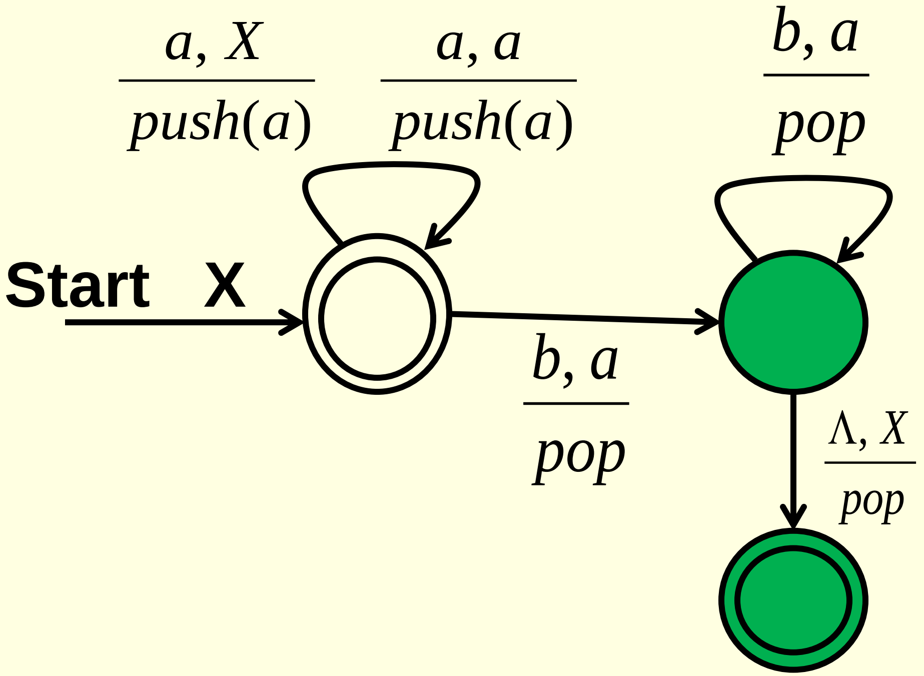
…

The grammar

{ $S \rightarrow A \mid B$ $\quad A \rightarrow aAb \mid \Lambda$ $\quad B \rightarrow aB \mid \Lambda$ }

for the language $\{ a^m, a^n b^n \mid m, n \in \mathbf{N} \}$

is not LL(k) for any k, but is deterministic



End of Context-Free Language and Pushdown Automata IV