Lecture 9

Monday February 3

Top-Down Parsing: Algorithm

backtrack ≜ pop *focus*.children; *focus* := *focus*.parent; *focus*.resetChildren

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R then
        create β1, β2 ... βn as children of focus
        trace.push(βn βn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
        end
      end
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then  return root
    else backtrack
    end
```

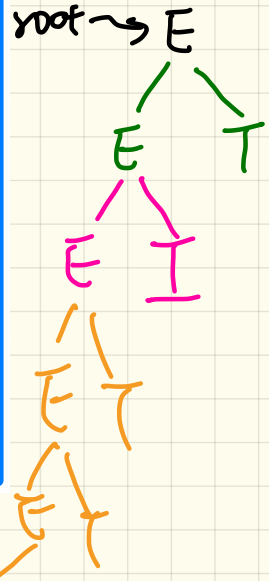
Top-Down Parsing: Discovering Leftmost Derivations (1)

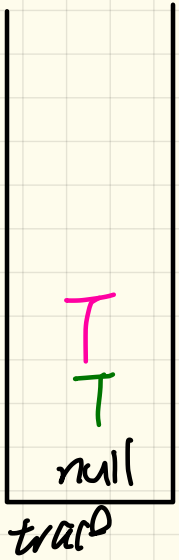
backtrack ≜ pop *focus*.children; *focus* := *focus*.parent; *focus*.resetChildren

Parse: a + a * a

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R then
        create β1, β2 ... βn as children of focus
        trace.push(βn βn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
        end
      end
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
    end
```

Expr E	→	Expr + Term
	|	Term
Term T	→	Term * Factor
	|	Factor
Factor F	→	(Expr)
	|	a





word : a

focus : E E E

Left-Recursions (LRs): Direct vs. Indirect

Direct Left-Recursions:

Expr	→	Expr + Term
	|	Term
Term	→	Term * Factor
	|	Factor
Factor	→	(Expr)
	|	a

Expr	→	Expr + Term
	|	Expr − Term
	|	Term
Term	→	Term * Factor
	|	Term / Factor
	|	Factor

Indirect Left-Recursions:

A	→	Br
B	→	Cd
C	→	At

① $A \to Br$ ② $B \Rightarrow^{*} Atd$

② $A \to Ba$, $B \Rightarrow^{*} Aafd$

A	→	Ba	|	b	
B	→	Cd	|	e	
C	→	Df	|	g	
D	→	f	|	Aa	| Cg

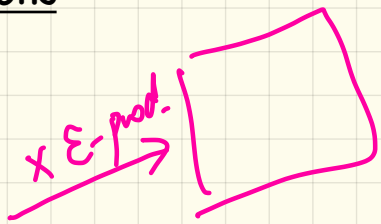
CFGs: Left-Recursive vs. Right-Recursive

CFG with Left Recursions

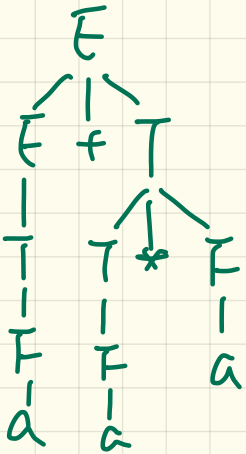
$$
\begin{array}{lcl}
Expr & \rightarrow & Expr + Term \\
 & | & Term \\
Term & \rightarrow & Term * Factor \\
 & | & Factor \\
Factor & \rightarrow & (Expr) \\
 & | & a
\end{array}
$$

CFG with Right Recursions

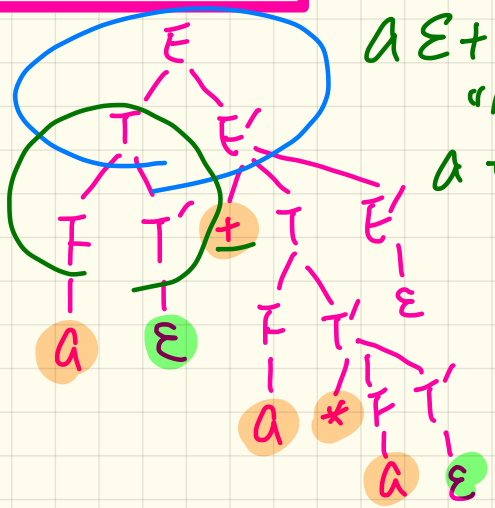
$$
\begin{array}{lcl}
Expr & \rightarrow & Term \quad Expr' \\
Expr' & \rightarrow & + \quad Term \quad Expr' \\
 & | & \epsilon \\
Term & \rightarrow & Factor \quad Term' \\
Term' & \rightarrow & * \quad Factor \quad Term' \\
 & | & \epsilon \\
Factor & \rightarrow & (Expr) \\
 & | & a
\end{array}
$$



Example: a + a * a



algo. semantically preserving.



a ε + a * a ε
or
a + a * a

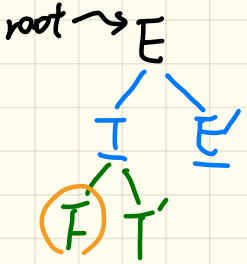
Top-Down Parsing: Discovering Leftmost Derivations (2)

backtrack ≜ pop *focus*.children; *focus* := *focus*.parent; *focus*.resetChildren

Parse: a + a * a

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R then
        create β1 β2 ... βn as children of focus
        trace.push(βn βn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
        end
      end
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
    end
```

E	Expr	→ Term Expr′
E′	Expr′	→ + Term Expr′
		| ϵ
T	Term	→ Factor Term′
T′	Term′	→ * Factor Term′
		| ϵ
F	Factor	→ (Expr)
		| a



look ahead

word: a

focus: ~~E~~ ~~T~~ ~~E′~~ ~~T′~~ F

Top-Down Parsing: Discovering Leftmost Derivations (3)

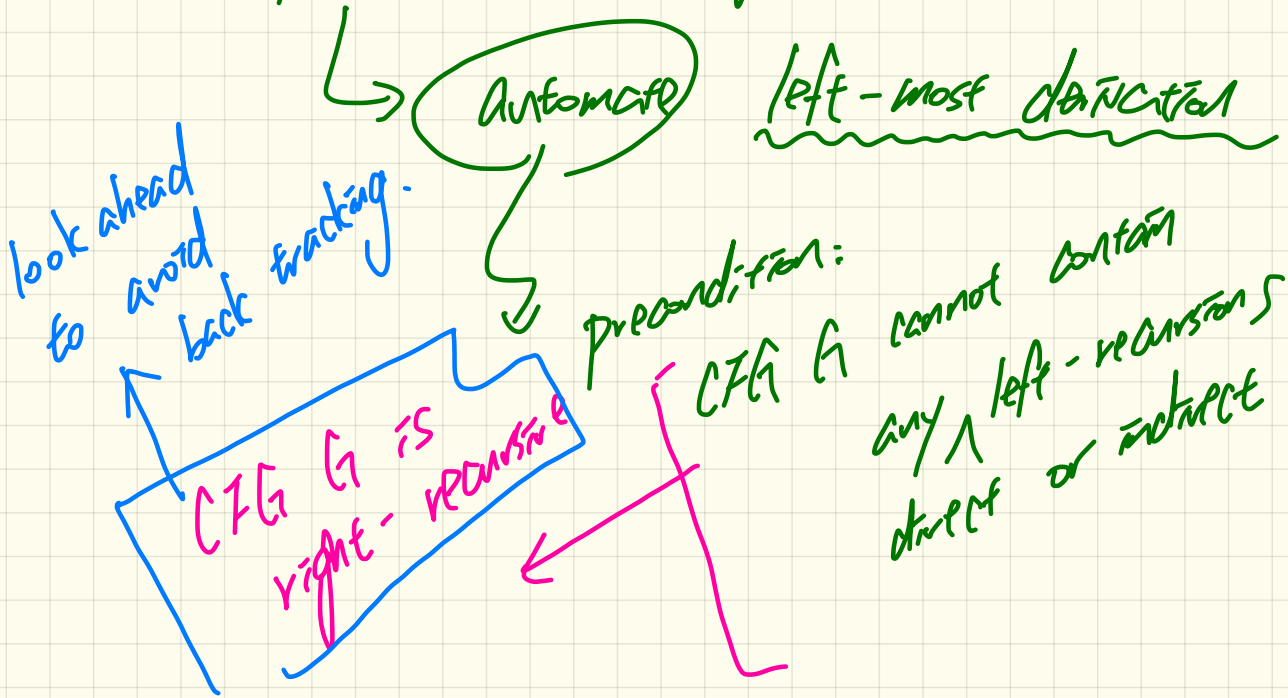
backtrack ≜ pop *focus*.children; *focus* := *focus*.parent; *focus*.resetChildren

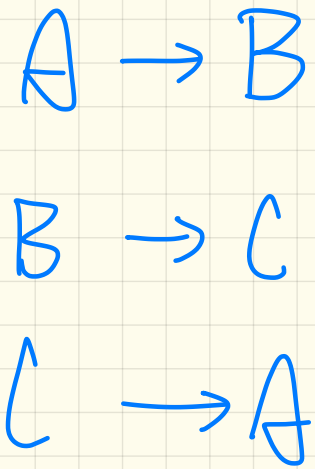
Parse: (a + a) * a

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R then
        create β1, β2 ... βn as children of focus
        trace.push(βn βn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
        end
      end
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then  return root
    else backtrack
    end
```

Expr	→	*Term* *Expr′*
Expr′	→	+ *Term* *Expr′*
	|	ϵ
Term	→	*Factor* *Term′*
Term′	→	* *Factor* *Term′*
	|	ϵ
Factor	→	(*Expr*)
	|	a

Top-Down Parsing




A → B
B → C
C → A

Removing Left-Recursions: Algorithm

```
1  ALGORITHM: RemoveLR
2    INPUT: CFG G = (V, Σ, R, S)
3    ASSUME: G acyclic ∧ with no ε-productions
4    OUTPUT: G' s.t. G' ≡ G, G' has no
5            indirect & direct left-recursions
6  PROCEDURE:
7    impose an order on V: ⟨⟨A_1, A_2, ..., A_n⟩⟩
8    for i: 1 .. n:
9      for j: 1 .. i − 1:
10       if ∃ A_i → A_j γ ∈ R ∧ A_j → δ_1 | δ_2 | ... | δ_m ∈ R then
11         replace A_i → A_j γ with A_i → δ_1 γ | δ_2 γ | ... | δ_m γ
12       end
13     for A_i → A_i α | β ∈ R:
14       replace it with: A_i → β A', A' → α A' | ε
```

remove indirect LRs

remove direct LRs.

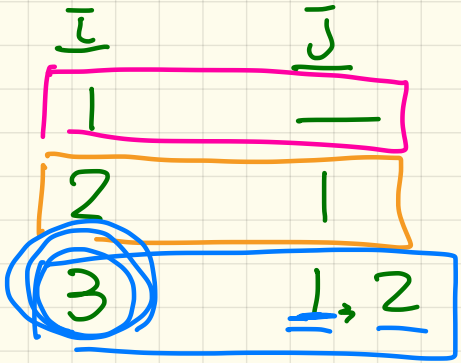
$A_i \to A_j \gamma$

$A_i \to \delta_1 \gamma \mid \delta_2 \gamma \mid \vdots \mid \delta_m \gamma$

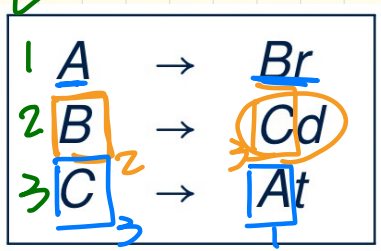
$\delta_1 \mid \delta_2 \cdots \delta_m$

Removing Left-Recursions (1)

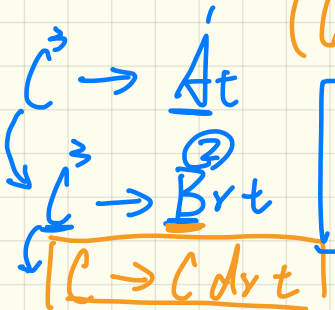
```
1  ALGORITHM: RemoveLR
2    INPUT: CFG G = (V, Σ, R, S)
3    ASSUME: G acyclic ∧ with no ε-productions
4    OUTPUT: G' s.t. G' ≡ G, G' has no
5            indirect & direct left-recursions
6  PROCEDURE:
7    impose an order on V: ⟨(A1, A2, ..., An)⟩
8    for i: 1 .. n:
9      for j: 1 .. i − 1:
10       if ∃ Ai → Ajγ ∈ R ∧ Aj → δ1 | δ2 | ... | δm ∈ R then
11         replace Ai → Ajγ with Ai → δ1γ | δ2γ | ... | δmγ
12       end
13     for Ai → Aiα | β ∈ R:
14       replace it with: Ai → βA', A' → αA' | ε
```

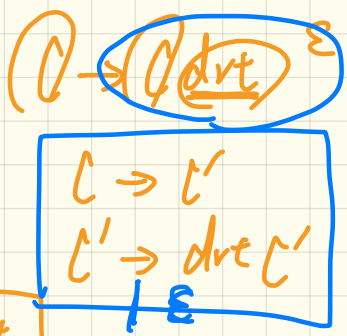


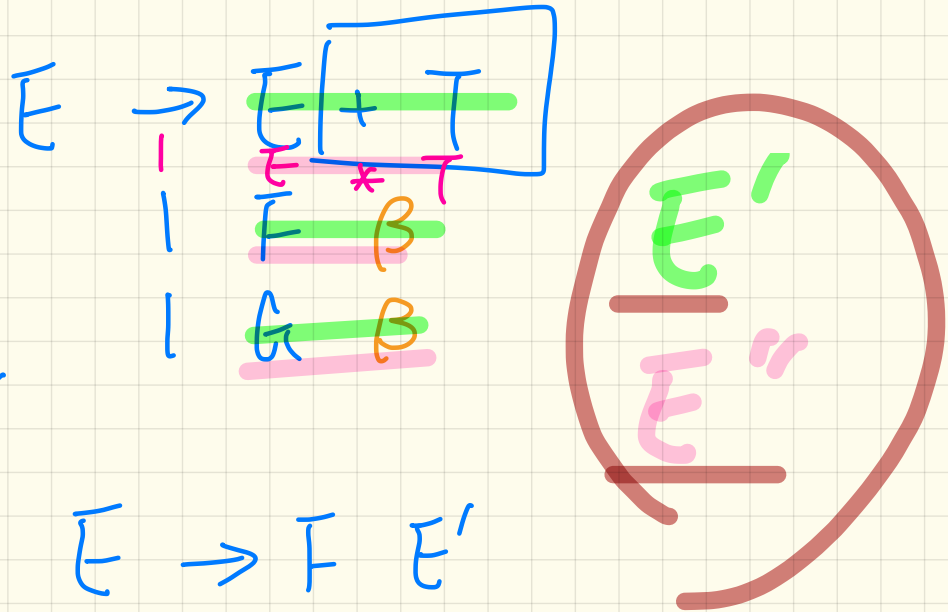
Indirectly Left-Recursive CFG:



i = 2
j = 1






E → E + T
E * T
F β
G β
E'
E''
E → F E'
G E'
E' → + T E'
ε

Removing Left-Recursions (2)

```
1  ALGORITHM: RemoveLR
2    INPUT: CFG G = (V, Σ, R, S)
3    ASSUME: G acyclic ∧ with no ϵ-productions
4    OUTPUT: G′ s.t. G′ ≡ G, G′ has no
5            indirect & direct left-recursions
6  PROCEDURE:
7    impose an order on V: ⟨⟨A1, A2, ..., An⟩⟩
8    for i: 1 .. n:
9      for j: 1 .. i − 1:
10       if ∃ Ai → Ajγ ∈ R ∧ Aj → δ1 | δ2 | ... | δm ∈ R then
11         replace Ai → Ajγ with Ai → δ1γ | δ2γ | ... | δmγ
12       end
13     for Ai → Aiα | β ∈ R:
14       replace it with: Ai → βA′, A′ → αA′ | ϵ
```

Indirectly Left-Recursive CFG:

$$
\begin{array}{lcl}
A & \rightarrow & Ba \mid b \\
B & \rightarrow & Cd \mid e \\
C & \rightarrow & Df \mid g \\
D & \rightarrow & f \mid Aa \mid Cg
\end{array}
$$

Removing Left-Recursions (3)

```
1  ALGORITHM: RemoveLR
2    INPUT: CFG G = (V, Σ, R, S)
3    ASSUME: G acyclic ∧ with no ϵ-productions
4    OUTPUT: G′ s.t. G′ ≡ G, G′ has no
5            indirect & direct left-recursions
6  PROCEDURE:
7    impose an order on V: ⟨⟨A1, A2, ..., An⟩⟩
8    for i: 1 .. n:
9      for j: 1 .. i − 1:
10       if ∃ Ai → Ajγ ∈ R ∧ Aj → δ1 | δ2 | ... | δm ∈ R then
11         replace Ai → Ajγ with Ai → δ1γ | δ2γ | ... | δmγ
12       end
13     for Ai → Aiα | β ∈ R:
14       replace it with: Ai → βA′, A′ → αA′ | ϵ
```

Directly Left-Recursive CFG:

$$
\begin{array}{lcl}
\mathit{Expr} & \rightarrow & \mathit{Expr} \; + \; \mathit{Term} \\
 & | & \mathit{Term} \\
\mathit{Term} & \rightarrow & \mathit{Term} \; * \; \mathit{Factor} \\
 & | & \mathit{Factor} \\
\mathit{Factor} & \rightarrow & (\mathit{Expr}) \\
 & | & \mathtt{a}
\end{array}
$$

Removing Left-Recursions (4)

```
1  ALGORITHM: RemoveLR
2    INPUT: CFG G = (V, Σ, R, S)
3    ASSUME: G acyclic ∧ with no ϵ-productions
4    OUTPUT: G′ s.t. G′ ≡ G, G′ has no
5            indirect & direct left-recursions
6  PROCEDURE:
7    impose an order on V: ⟨⟨A_1, A_2, ..., A_n⟩⟩
8    for i: 1 .. n:
9      for j: 1 .. i − 1:
10       if ∃ A_i → A_j γ ∈ R ∧ A_j → δ_1 | δ_2 | ... | δ_m ∈ R then
11         replace A_i → A_j γ with A_i → δ_1 γ | δ_2 γ | ... | δ_m γ
12       end
13     for A_i → A_i α | β ∈ R:
14       replace it with: A_i → β A′, A′ → α A′ | ϵ
```

Directly Left-Recursive CFG:

$$
\begin{array}{lcl}
\mathit{Expr} & \rightarrow & \mathit{Expr} + \mathit{Term} \\
 & | & \mathit{Expr} - \mathit{Term} \\
 & | & \mathit{Term} \\
\mathit{Term} & \rightarrow & \mathit{Term} \star \mathit{Factor} \\
 & | & \mathit{Term} \ / \ \mathit{Factor} \\
 & | & \mathit{Factor}
\end{array}
$$

A is not nullable

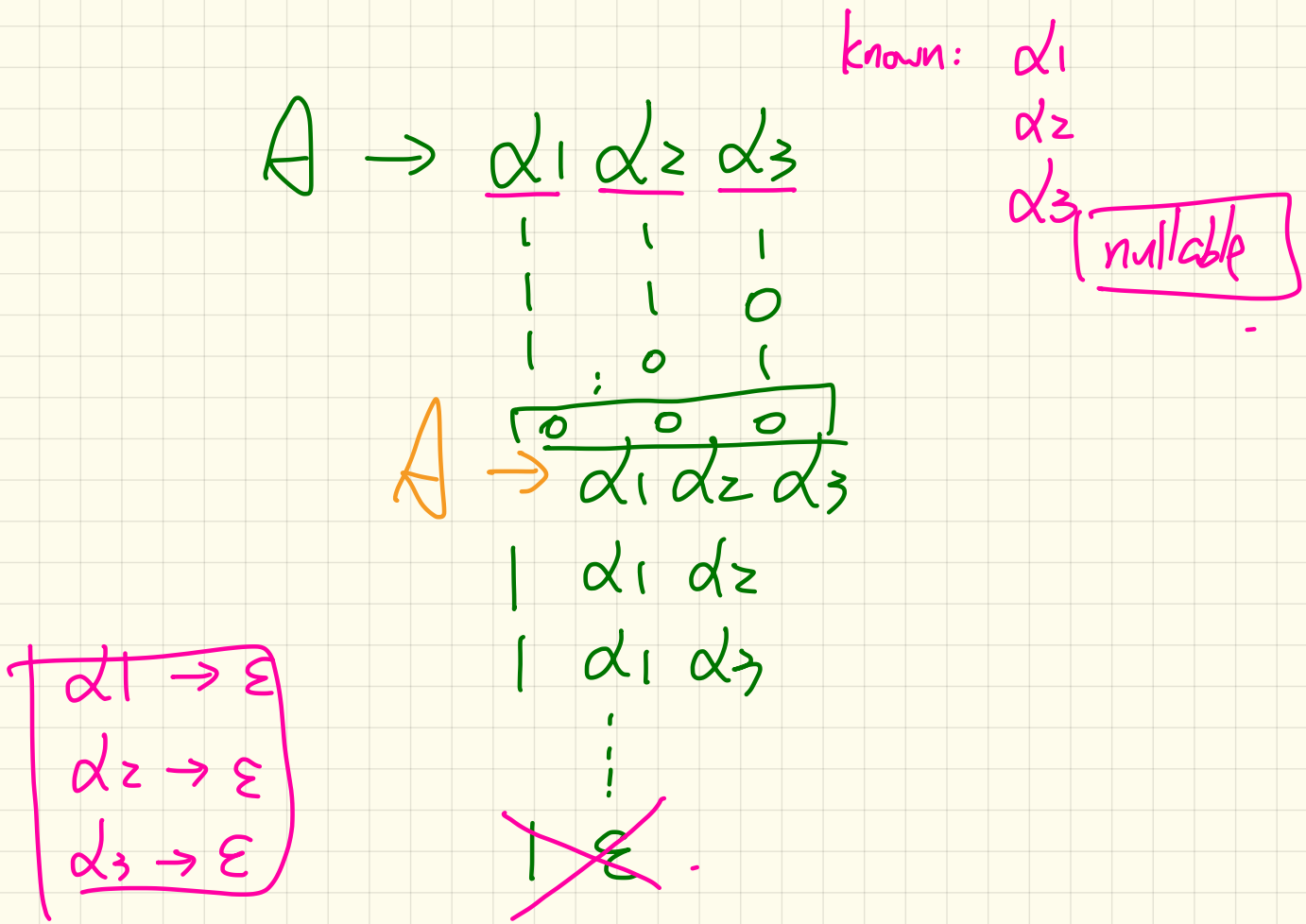
$A \to \varepsilon$ nullable

$B \to C$
$C \to D$
$D \to \varepsilon$

B, C, D nullable

$B \to \alpha_1 \alpha_2 \cdots \alpha_n$

B is nullable if? $\alpha_1 \alpha_2 \alpha_3 \cdots \alpha_n$ all nullable


known: α_1
α_2
α_3
nullable
$A \to \alpha_1 \alpha_2 \alpha_3$
1 1 0
1 0 1
0 0 0
$A \to \alpha_1 \alpha_2 \alpha_3$
$\alpha_1 \alpha_2$
$\alpha_1 \alpha_3$
$\alpha_1 \to \varepsilon$
$\alpha_2 \to \varepsilon$
$\alpha_3 \to \varepsilon$

Eliminating epsilon-Productions

$G1 \neq G2$

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow aAA \mid \epsilon \\
B &\rightarrow bBB \mid \epsilon
\end{aligned}
$$

→ G1

$\varepsilon \in L(G1)$

Q: Nullable variables? A B S

G2

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

→ $\varepsilon \notin L(G2)$