CFG: Formal Definition

Design the CFG for strings of properly-nested parentheses.

e.g., `()`, `()()`, `(((())()))()`, *etc.*

Present your answer in a formal manner.

A ***context-free grammar (CFG)*** is a 4-tuple (V, Σ, R, S):

- V is a finite set of **variables**.
- Σ is a finite set of ***terminals***.
- R is a finite set of ***rules*** s.t.

$$R \subseteq \{v \to s \mid \}$$

- $S \in V$ is is the **start variable**.

Given strings u, v, w , variable a rule :

- $uAv \Rightarrow uwv$ menas that uAv ***yields*** uwv.
- $u \overset{*}{\Rightarrow} v$ means that u ***derives*** v, if:
  - $u = v$; or
  - $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$ [a ***yield sequence***]

Given a CFG $G = (V, \Sigma, R, S)$, the language of G

$$L(G) = \{\}$$

Context-Free Grammar (CFG): Example Version 3

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

Example: a * a + a

Context-Free Grammar (CFG): Leftmost Derivation

Parse Tree: a + a * a

LMD: a + a * a

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

Order of evaluation?

A **parse tree** may correspond to:

+ multiple **derivations**
+ a unique **LMD**

Context-Free Grammar (CFG): Rightmost Derivation

Parse Tree: a + a * a

RMD: a + a * a

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

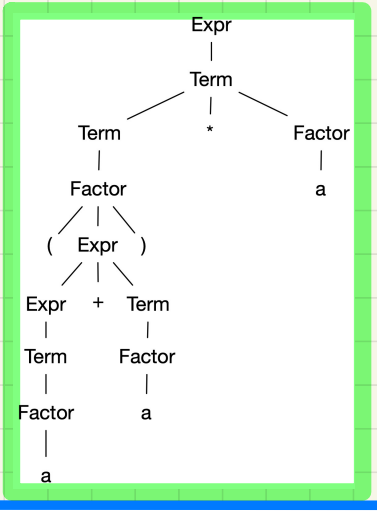
Order of evaluation?

A **parse tree** may correspond to:

+ multiple **derivations**
+ a unique **RMD**

Context-Free Grammar (CFG): Leftmost Derivation

Parse Tree: (a + a) * a



$$
\begin{aligned}
Expr &\rightarrow Expr + Term \mid Term \\
Term &\rightarrow Term * Factor \mid Factor \\
Factor &\rightarrow (Expr) \mid a
\end{aligned}
$$

LMD: (a + a) * a

$$
\begin{aligned}
Expr &\Rightarrow Term \\
&\Rightarrow Term * Factor \\
&\Rightarrow Factor * Factor \\
&\Rightarrow (\ Expr\ ) * Factor \\
&\Rightarrow (\ Expr + Term\ ) * Factor \\
&\Rightarrow (\ Term + Term\ ) * Factor \\
&\Rightarrow (\ Factor + Term\ ) * Factor \\
&\Rightarrow (\ a + Term\ ) * Factor \\
&\Rightarrow (\ a + Factor\ ) * Factor \\
&\Rightarrow (\ a + a\ ) * Factor \\
&\Rightarrow (\ a + a\ ) * a
\end{aligned}
$$

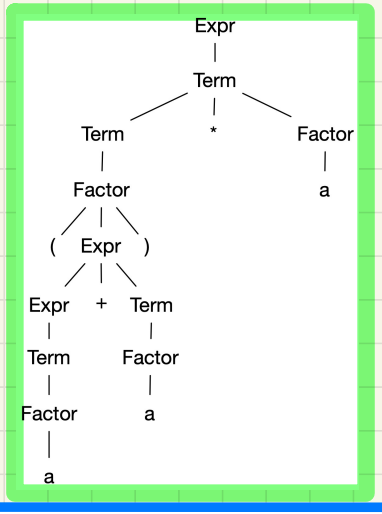
Order of evaluation?

A parse tree may correspond to:

+ multiple derivations
+ a unique LMD

Context-Free Grammar (CFG): Rightmost Derivation

Parse Tree: (a + a) * a



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

RMD: (a + a) * a

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

Order of evaluation?

A **parse tree** may correspond to:

+ multiple **derivations**
+ a unique **RMD**

Context-Free Grammar (CFG): Exercise (1)

Is the following CFG ambiguous?

$$Expr \rightarrow Expr \ + \ Expr \mid Expr \ * \ Expr \mid (\ Expr \ ) \mid a$$

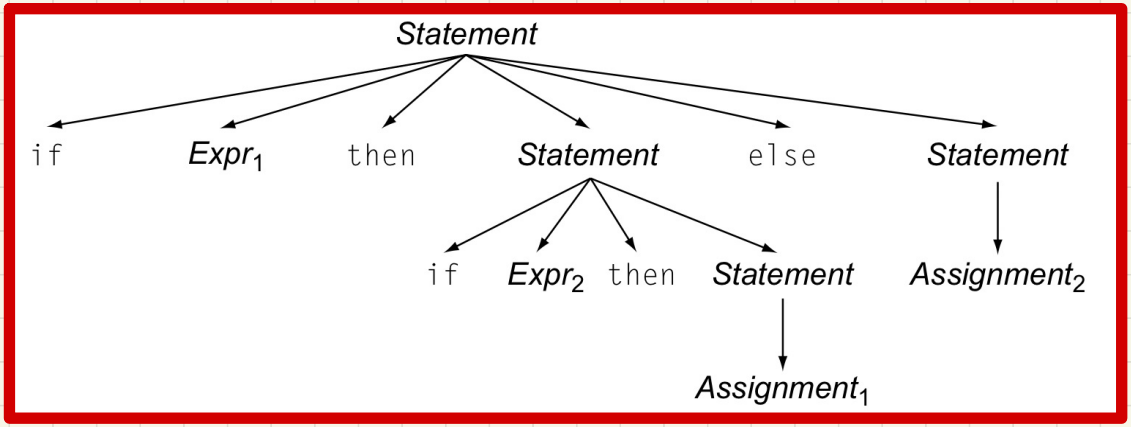
Context-Free Grammar (CFG): Exercise (2.1.1)

Is the following CFG ambiguous?

```
Statement  →  if Expr then Statement
           |  if Expr then Statement else Statement
           |  Assignment
           ...
```

Example: A Possible **Semantic Interpretation**?

if *Expr1* **then** **if** *Expr2* **then** *Assignment1* **else** *Assignment2*



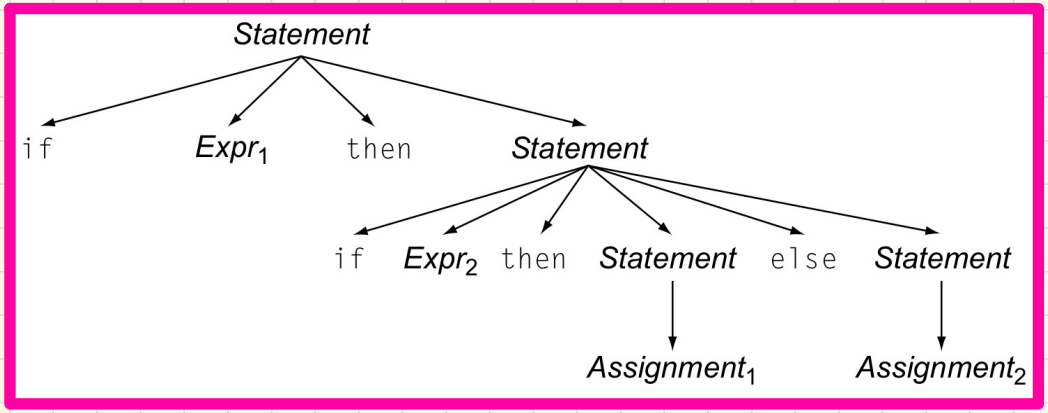
Context-Free Grammar (CFG): Exercise (2.1.2)

Is the following CFG ambiguous?

```
Statement  →  if Expr then Statement
           |  if Expr then Statement else Statement
           |  Assignment
           ...
```

Example: A Possible **Semantic Interpretation**?

if *Expr1* **then** **if** *Expr2* **then** *Assignment1* **else** *Assignment2*



Context-Free Grammar (CFG): Exercise (2.2)

Is the following CFG ambiguous?

Statement → `if` *Expr* `then` *Statement*
| `if` *Expr* `then` *WithElse* `else` *Statement*
| *Assignment*

WithElse → `if` *Expr* `then` *WithElse* `else` *WithElse*
| *Assignment*

Example: How many possible **semantic interpretations**?

if *Expr1* **then if** *Expr2* **then** *Assignment1* **else** *Assignment2*

Can a derivation starting with Statement work?

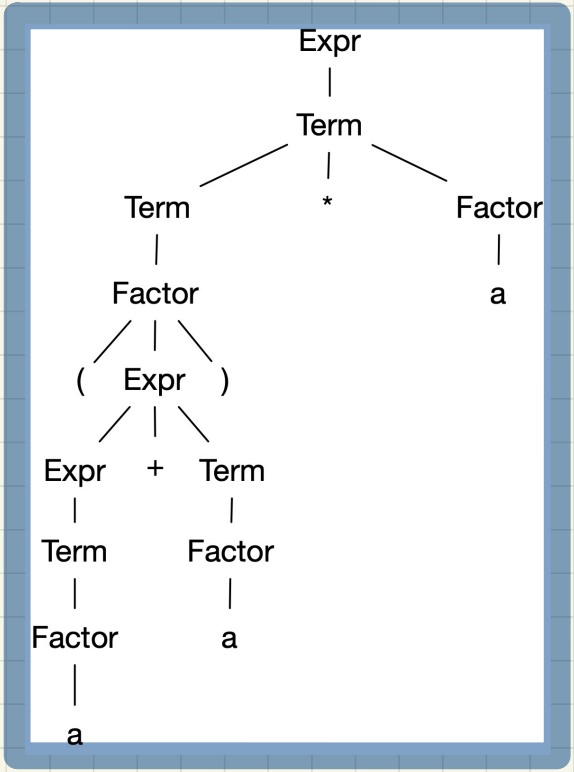
Can a derivation starting with WithElse work?

Discovering Derivations

Input Grammar G

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

AST: (a + a) * a

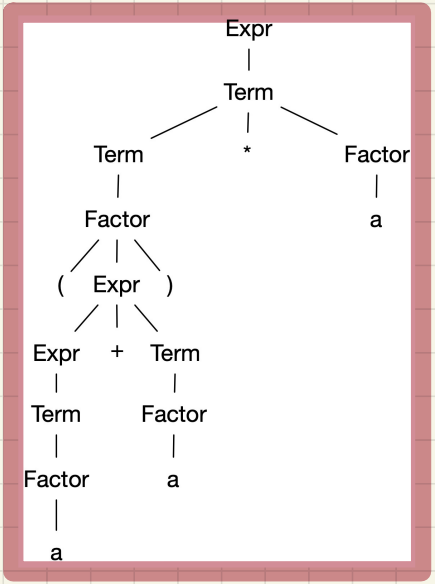


Discovering Derivations: Top-Down vs. Bottom-Up

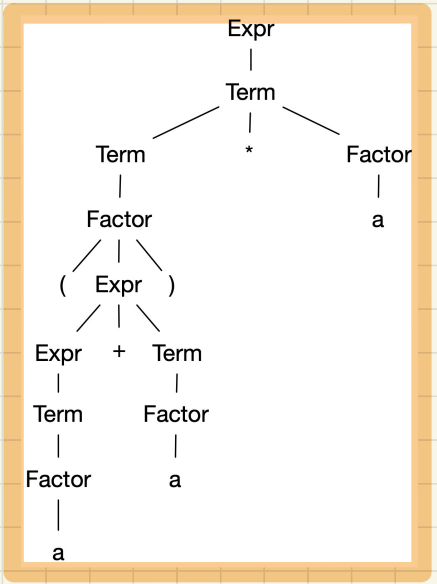
Input Grammar G

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

TDP: (a + a) * a



BUP: (a + a) * a



Top-Down Parsing: Algorithm

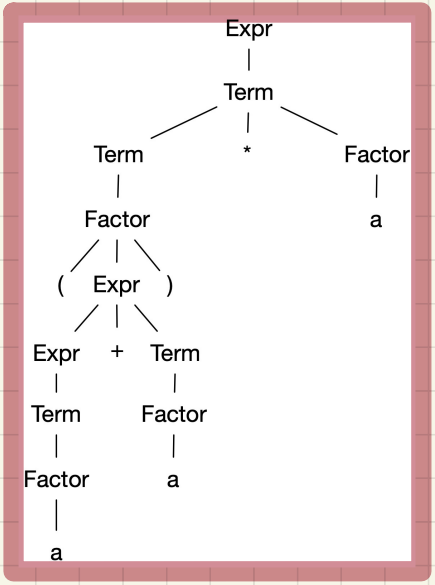
```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited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

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

Input Grammar G

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

TDP: (a + a) * a



Top-Down Parsing: Discovering Leftmost Derivations (1)

```
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
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
```

Parse: a + a * a

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

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

Left-Recursions (LRs): Direct vs. Indirect

Direct Left-Recursions:

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

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

Indirect Left-Recursions:

$$
\begin{array}{lcl}
A & \rightarrow & Br \\
B & \rightarrow & Cd \\
C & \rightarrow & At
\end{array}
$$

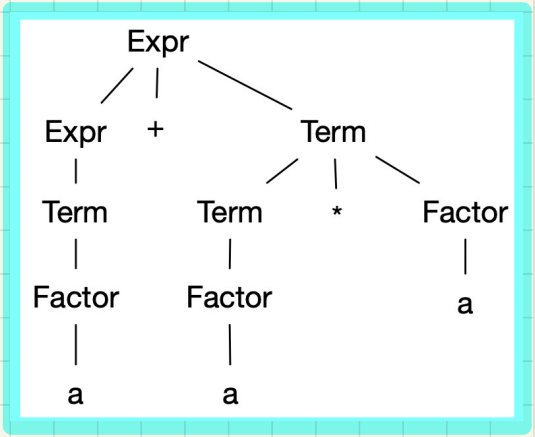
$$
\begin{array}{lcl}
A & \rightarrow & Ba \mid b \\
B & \rightarrow & Cd \mid e \\
C & \rightarrow & Df \mid g \\
D & \rightarrow & f \mid Aa \mid Cg
\end{array}
$$

CFGs: Left-Recursive vs. Right-Recursive

Example: a + a * a

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

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

Top-Down Parsing: Discovering Leftmost Derivations (2)

```
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
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
```

Parse: a + a * a

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

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

Top-Down Parsing: Discovering Leftmost Derivations (3)

```
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
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
```

Parse: (a + a) * a

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

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

Removing Left-Recursions: Algorithm

```
1   ALGORITHM: RemoveLR
2     INPUT: CFG G = (V, Σ, R, S)
3     ASSUME: G has 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′_i, A′_i → α A′_i | ϵ
```

Removing Left-Recursions (1a)

```
1  ALGORITHM: RemoveLR
2    INPUT: CFG G = (V, Σ, R, S)
3    ASSUME: G has 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′i, A′i → αA′i | ϵ
```

Directly Left-Recursive CFG:

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

Removing Left-Recursions (1b)

```
1  ALGORITHM: RemoveLR
2    INPUT: CFG G = (V, Σ, R, S)
3    ASSUME: G has 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′i, A′i → αA′i | ϵ
```

Directly Left-Recursive CFG:

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

Removing Left-Recursions (2a)

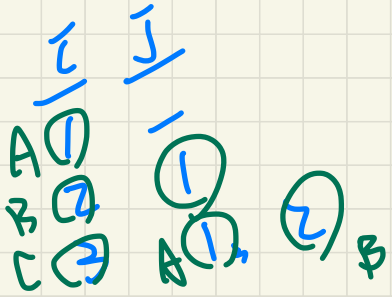
```
1  ALGORITHM: RemoveLR
2    INPUT: CFG G = (V, Σ, R, S)
3    ASSUME: G has no ϵ-productions
4    OUTPUT: G′ s.t. G′ ≡ G, G′ has no
5            indirect & direct left-recursions
6  PROCEDURE:
7    impose an order on V: ⟨(A1, A2, ..., An)⟩
8    for i: 1 .. n:
9      for j: 1 .. i − 1:
10       if ∃ 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′i, A′i → αA′i | ϵ
```

Indirectly Left-Recursive CFG:

1 $A \rightarrow Br$

1 $B \rightarrow Cd$

3 $C \rightarrow At$



Removing Left-Recursions (2b)

Does the order of variables matter?

```
1  ALGORITHM: RemoveLR
2    INPUT: CFG G = (V, Σ, R, S)
3    ASSUME: G has 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_i′, A_i′ → α A_i′ | ϵ
```

Indirectly Left-Recursive CFG:

C → At

B → Cd

A → Br

Removing Left-Recursions (2c)

```
1  ALGORITHM: RemoveLR
2    INPUT: CFG G = (V, Σ, R, S)
3    ASSUME: G has 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′i, A′i → αA′i | ϵ
```

Indirectly Left-Recursive CFG:

A	$\rightarrow$	Ba	$\mid$	b	
B	$\rightarrow$	Cd	$\mid$	e	
C	$\rightarrow$	Df	$\mid$	g	
D	$\rightarrow$	f	$\mid$	Aa	$\mid$ Cg

Eliminating epsilon-Productions

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

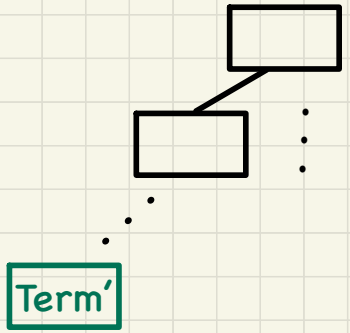
Q: **Nullable** variables?

Top-Down Parsing: Backtrack

```
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
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
```

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

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr′*
2	*Expr′*	→	+ *Term Expr′*
3		|	- *Term Expr′*
4		|	ϵ
5	*Term*	→	*Factor Term′*
6	*Term′*	→	x *Factor Term′*
7		|	÷ *Factor Term′*
8		|	ϵ
9	*Factor*	→	(*Expr*)
10		|	num
11		|	name



FIRST Set

$$\textsc{First}(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \in T \\ \{w \mid w \in \Sigma^* \wedge \alpha \overset{*}{\Rightarrow} w\beta \wedge \beta \in (V \cup \Sigma)^*\} & \text{if } \alpha \in V \end{cases}$$

Right-Recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	× *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST										

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FIRST					

FIRST Set

$$\textsc{First}(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \in T \\ \{w \mid w \in \Sigma^* \wedge \alpha \overset{*}{\Rightarrow} w\beta \wedge \beta \in (V \cup \Sigma)^*\} & \text{if } \alpha \in V \end{cases}$$

Right-Recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

Q. Will FIRST(Expr′) change if we add another rule?

Expr′ → Term′ Factor

FIRST Set: Algorithm

$$\textsc{First}(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \in T \\ \{w \mid w \in \Sigma^* \wedge \alpha \overset{*}{\Rightarrow} w\beta \wedge \beta \in (V \cup \Sigma)^*\} & \text{if } \alpha \in V \end{cases}$$

```
ALGORITHM: GetFirst
  INPUT: CFG G = (V, Σ, R, S)
  T ⊂ Σ* denotes valid terminals
  OUTPUT: First : V ∪ T ∪ {ϵ, eof} ⟶ ℙ(T ∪ {ϵ, eof})
PROCEDURE:
  for α ∈ (T ∪ {eof, ϵ}): First(α) := {α}
  for A ∈ V: First(A) := ∅
  lastFirst := ∅
  while (lastFirst ≠ First):
    lastFirst := First
    for A → β1 β2 ... βk ∈ R s.t. ∀βj : βj ∈ (T ∪ V):
      rhs := First(β1) − {ϵ}
      for (i := 1; ϵ ∈ First(βi) ∧ i < k; i++):
        rhs := rhs ∪ (First(βi+1) − {ϵ})
      if i = k ∧ ϵ ∈ First(βk) then
        rhs := rhs ∪ {ϵ}
      end
      First(A) := First(A) ∪ rhs
```

β_i is nullable

$\beta_1 \ldots \beta_{k-1}$ are nullable

$$A \longrightarrow \beta_1 \; \beta_2 \quad \cdots \quad \beta_{k-1} \; \beta_k$$

Right-Recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

FIRST Set: Tracing

First choose rules whose RHS starts with a terminal

F, E′, T′, T, E

```
ALGORITHM: GetFirst
  INPUT: CFG G = (V, Σ, R, S)
  T ⊂ Σ* denotes valid terminals
  OUTPUT: FIRST : V ∪ T ∪ {ϵ, eof} ⟶ ℙ(T ∪ {ϵ, eof})
PROCEDURE:
  for α ∈ (T ∪ {eof, ϵ}): FIRST(α) := {α}
  for A ∈ V: FIRST(A) := ∅
  lastFirst := ∅
  while (lastFirst ≠ FIRST):
    lastFirst := FIRST
    for A → β1 β2 ... βk ∈ R s.t. ∀βj : βj ∈ (T ∪ V):
      rhs := FIRST(β1) − {ϵ}
      for (i := 1; ϵ ∈ FIRST(βi) ∧ i < k; i++):
        rhs := rhs ∪ (FIRST(βi+1) − {ϵ})
      if i = k ∧ ϵ ∈ FIRST(βk) then
        rhs := rhs ∪ {ϵ}
      end
      FIRST(A) := FIRST(A) ∪ rhs
```

num	name	+	-	×	÷	(	)	eof	ϵ

Expr	*Expr′*	*Term*	*Term′*	*Factor*

FIRST Set

```
ALGORITHM: GetFirst
   INPUT: CFG G = (V, Σ, R, S)
   T ⊂ Σ* denotes valid terminals
   OUTPUT: FIRST : V ∪ T ∪ {ϵ, eof} ⟶ ℙ(T ∪ {ϵ, eof})
PROCEDURE:
   for α ∈ (T ∪ {eof, ϵ}): FIRST(α) := {α}
   for A ∈ V: FIRST(A) := ∅
   lastFirst := ∅
   while (lastFirst ≠ FIRST):
      lastFirst := FIRST
      for A → β1 β2 ... βk ∈ R s.t. ∀βj : βj ∈ (T ∪ V):
         rhs := FIRST(β1) − {ϵ}
         for (i := 1; ϵ ∈ FIRST(βi) ∧ i < k; i++):
            rhs := rhs ∪ (FIRST(βi+1) − {ϵ})
         if i = k ∧ ϵ ∈ FIRST(βk) then
            rhs := rhs ∪ {ϵ}
         end
         FIRST(A) := FIRST(A) ∪ rhs
```

Right-Recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

Q. Will FIRST(Expr′) change if we add another rule?

Expr′ → Term′ Factor

Extended First Set

Q. How about FIRST(Expr′ → Term′ Factor)?

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST	num	name	+	-	x	÷	(	)	eof	ϵ

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FIRST	(, name, num	+, -, ϵ	(, name, num	x, ÷, ϵ	(, name, num

$$\textsc{First}(\beta_1\beta_2\ldots\beta_n) = \left\{ \textsc{First}(\beta_1) \cup \textsc{First}(\beta_2) \cup \ldots \textsc{First}(\beta_k) \;\middle|\; \begin{array}{l} \forall i : 1 \le i < k \bullet \epsilon \in \textsc{First}(\beta_i) \\ \wedge \\ \epsilon \notin \textsc{First}(\beta_k) \end{array} \right\}$$

Right-Recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

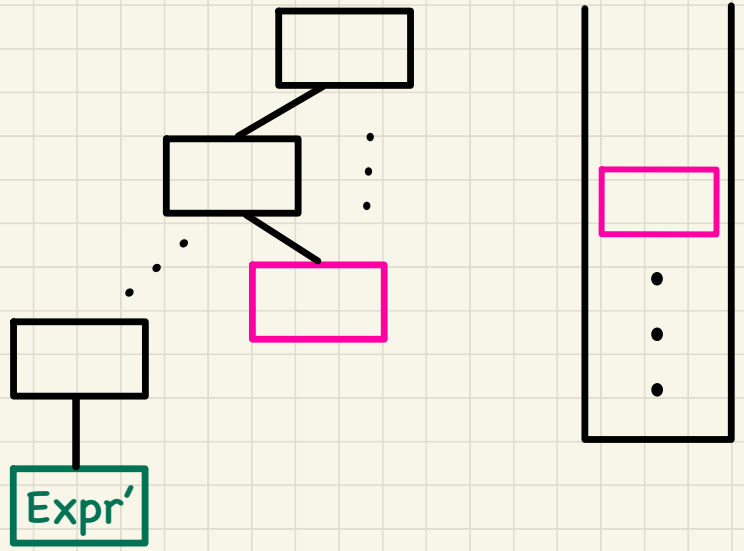
Is the FIRST Set Sufficient?

Expr′	→	+	*Term*	*Term′*	(1)
	|	−	*Term*	*Term′*	(2)
	|	ϵ			(3)

FIRST(+ Term Term′) =

FIRST(- Term Term′) =

FIRST(epsilon) =



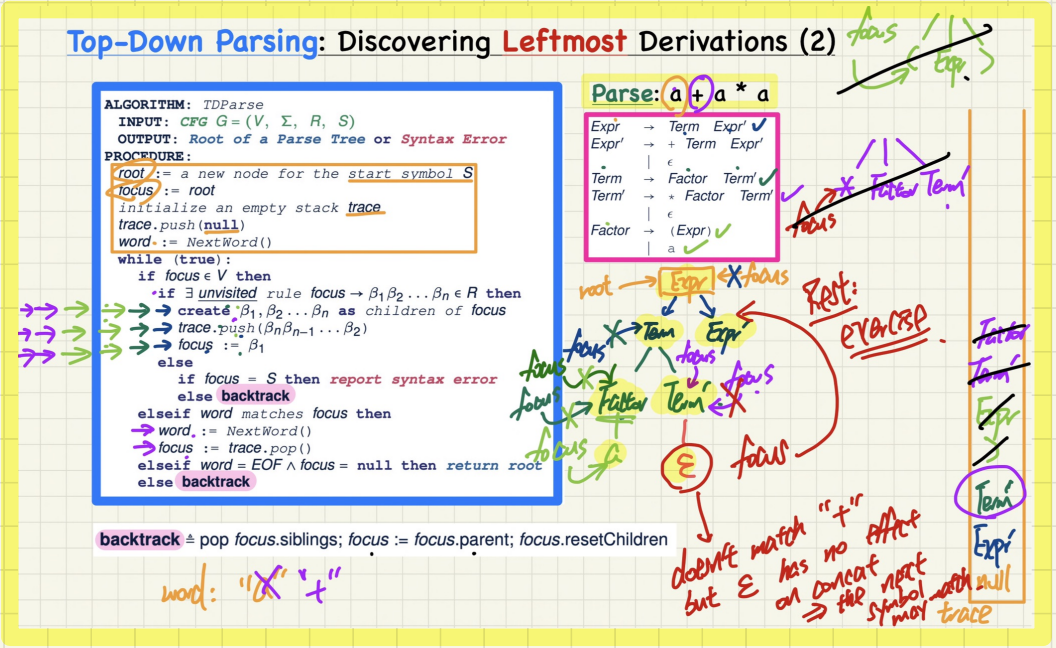
Is the FIRST Set Sufficient?

Expr′	→	+	*Term*	*Term′*	(1)
	|	−	*Term*	*Term′*	(2)
	|	ϵ			(3)

FIRST(+ Term Term′) =

FIRST(- Term Term′) =

FIRST(epsilon) =



FOLLOW Set

$$\text{FOLLOW}(v) = \{w \mid w, x, y \in \Sigma^* \wedge v \overset{*}{\Rightarrow} x \wedge S \overset{*}{\Rightarrow} xwy\}$$

Right-Recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

	Expr	Expr′	Term	Term′	Factor
FIRST	(, name, num	+, -, ϵ	(, name, num	x, ÷, ϵ	(, name, num

	Expr	Expr′	Term	Term′	Factor
FOLLOW	eof,)	eof,)	eof, +, -,)	eof, +, -,)	eof, +, -, x, ÷,)

FOLLOW Set: Algorithm

$$\textsc{Follow}(v) = \{w \mid w, x, y \in \Sigma^* \wedge v \overset{*}{\Rightarrow} x \wedge S \overset{*}{\Rightarrow} xwy\}$$

```
ALGORITHM: GetFollow
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Follow : V ⟶ ℙ(T ∪ {eof})
PROCEDURE:
  for A ∈ V: Follow(A) := ∅
  Follow(S) := {eof}
  lastFollow := ∅
  while (lastFollow ≠ Follow):
    lastFollow := Follow
    for A → β1 β2 ... βk ∈ R:
      trailer := Follow(A)
      for i: k .. 1:
        if βi ∈ V then
          Follow(βi) := Follow(βi) ∪ trailer
          if ε ∈ First(βi)
            then trailer := trailer ∪ (First(βi) − ε)
            else trailer := First(βi)
        else
          trailer := First(βi)
```

$$A \longrightarrow \beta_1 \; \beta_2 \quad \cdots \quad \beta_{k-1} \mid \beta_k$$

FOLLOW(β_k) = ?

When $\varepsilon \in$ FIRST(β_k)

FOLLOW(β_{k-1}) = ?

When $\varepsilon \notin$ FIRST(β_k)

FOLLOW(β_{k-1}) = ?

Computing the FOLLOW Sets: Trailers

$$A \longrightarrow \beta_1\beta_2\beta_3$$

Case 1: $\varepsilon \notin$ FIRST(β_3), $\varepsilon \notin$ FIRST(β_2)

+ FOLLOW(β_3)
+ FOLLOW(β_2)
+ FOLLOW(β_1)

Case 2: $\varepsilon \in$ FIRST(β_3), $\varepsilon \in$ FIRST(β_2)

+ FOLLOW(β_3)
+ FOLLOW(β_2)
+ FOLLOW(β_1)

Right-Recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

G, F, E, T, T′

```
ALGORITHM: GetFollow
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: FOLLOW : V ⟶ ℙ(T ∪ {eof})
PROCEDURE:
  for A ∈ V: FOLLOW(A) := ∅
  FOLLOW(S) := {eof}
  lastFollow := ∅
  while (lastFollow ≠ FOLLOW):
    lastFollow := FOLLOW
    for A → β1 β2 ... βk ∈ R:
      trailer := FOLLOW(A)
      for i: k .. 1:
        if βi ∈ V then
          FOLLOW(βi) := FOLLOW(βi) ∪ trailer
          if ϵ ∈ FIRST(βi)
            then trailer := trailer ∪ (FIRST(βi) − ϵ)
            else trailer := FIRST(βi)
        else
          trailer := FIRST(βi)
```

FOLLOW Set: Tracing

First choose rules whose LHS is processed.
Then rules whose RHS ends with a terminal.

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FIRST	(, name, num	+, -, ϵ	(, name, num	x, ÷, ϵ	(, name, num

Goal	*Expr*	*Expr′*	*Term*	*Term′*	*Factor*

Backtrack-Free Grammar

A ***backtrack-free grammar*** has each of its productions $A \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$ satisfying:

$$\forall i, j : 1 \leq i, j \leq n \wedge i \neq j \bullet \textbf{START}(\gamma_i) \cap \textbf{START}(\gamma_j) = \varnothing$$

$$\textbf{START}(A \to \beta) = \begin{cases} \textbf{FIRST}(\beta) & \text{if } \epsilon \notin \textbf{FIRST}(\beta) \\ \textbf{FIRST}(\beta) \cup \textbf{FOLLOW}(A) & \text{otherwise} \end{cases}$$

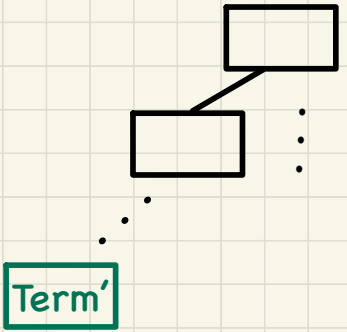
FIRST(β) is the extended version where β may be $\beta_1 \beta_2 \ldots \beta_n$

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

Top-Down Parsing: Algorithm with lookahead

```
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 ∧ word ∈ START(β) 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
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
```

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr′*
2	*Expr′*	→	+ *Term Expr′*
3		|	- *Term Expr′*
4		|	ϵ
5	*Term*	→	*Factor Term′*
6	*Term′*	→	x *Factor Term′*
7		|	÷ *Factor Term′*
8		|	ϵ
9	*Factor*	→	(*Expr*)
10		|	num
11		|	name



Backtrack-Free Grammar: Exercise

A ***backtrack-free grammar*** has each of its productions $A \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$ satisfying:

$$\forall i, j : 1 \le i, j \le n \wedge i \neq j \bullet \textsc{Start}(\gamma_i) \cap \textsc{Start}(\gamma_j) = \varnothing$$

$$\textsc{Start}(A \to \beta) = \begin{cases} \textsc{First}(\beta) & \text{if } \epsilon \notin \textsc{First}(\beta) \\ \textsc{First}(\beta) \cup \textsc{Follow}(A) & \text{otherwise} \end{cases}$$

$\textsc{First}(\beta)$ is the extended version where β may be $\beta_1 \beta_2 \ldots \beta_n$

Is the following CFG ***backtrack free***?

11	*Factor*	→	`name`
12		|	`name [` *ArgList* `]`
13		|	`name (` *ArgList* `)`
15	*ArgList*	→	*Expr MoreArgs*
16	*MoreArgs*	→	`,` *Expr MoreArgs*
17		|	ϵ

Left-Factoring: Removing Common Prefixes

Identify a common prefix α:

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_j$$

[each of $\gamma_1, \gamma_2, \ldots, \gamma_j$ does not begin with α]

Rewrite that production rule as:

$$\begin{aligned} A &\to \alpha B \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_j \\ B &\to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \end{aligned}$$

11	*Factor*	→	`name`
12		|	`name` `[` *ArgList* `]`
13		|	`name` `(` *ArgList* `)`
15	*ArgList*	→	*Expr MoreArgs*
16	*MoreArgs*	→	`,` *Expr MoreArgs*
17		|	ϵ

Implementing a Recursive-Descent Parser

	Production			FIRST$^+$
2	*Expr′*	→	+ *Term Expr′*	{+}
3		|	- *Term Expr′*	{-}
4		|	ϵ	{ϵ, eof,)}

```
ExprPrim()
  if word = + ∨ word = - then /* Rules 2, 3 */
    word := NextWord()
    if(Term())
      then return ExprPrim()
      else return false
  elseif word = ) ∨ word = eof then /* Rule 4 */
    return true
  else
    report a syntax error
    return false
  end

Term()
  ...
```