Lecture 18 - July 22

Syntactic Analysis

Discovering Derivations: TD vs. BU
TDP Algorithm: Left-Recursive
TDP Algorithm: Tracing Exercise

Announcements/Reminders

- WrittenTest being graded
- Project Milestone 2 released
- Project Report Template to be released
- Assignment 3 (Top-Down Parsing)
- Exam: 9 AM, Wednesday, August 13 (DB 0007)
- Makeup and Exam Study lectures (Bottom-Up Parsing) be released

Discovering Derivations

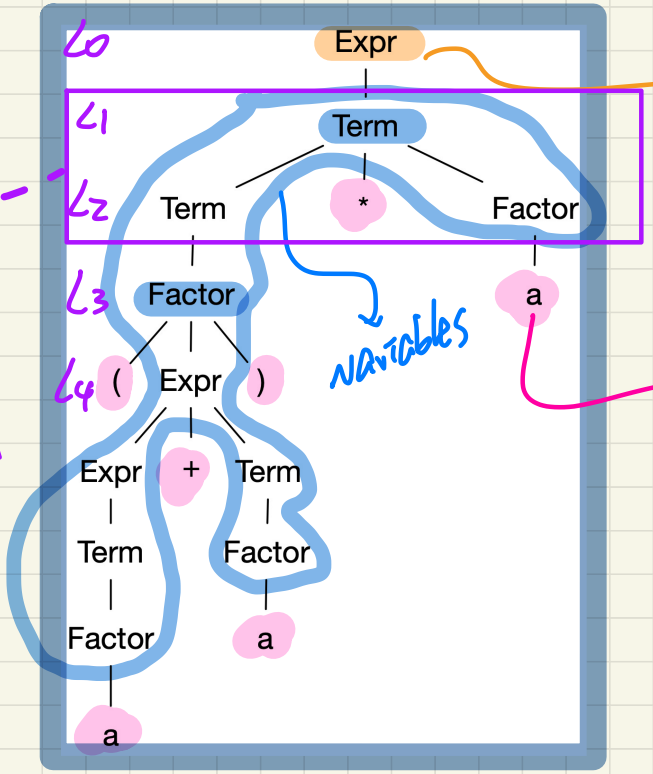
Expr $\overset{*}{\Rightarrow}$ (a+a) * a

start variable — derivation steps to be discovered — input string

Input Grammar G

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

AST: (a + a) * a



start variable

variables

grammatical connection (production rule)

terminals (some traversal on leaves should give back the input string).

- root
- internal nodes
- external nodes

production : $V \to W$

Discovering Derivations: Top-Down vs. Bottom-Up

$V \rightsquigarrow W$ $W \rightsquigarrow V$

left-most derivation.

Input Grammar G

Expr	→	① Expr	+	Term
	|	② Term		
Term	→	Term	*	Factor
	|	Factor		
Factor	→	(Expr)		
	|	a		

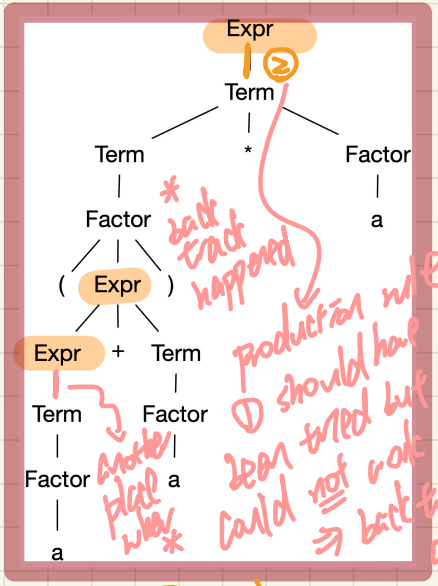
start variable

Expr

for each occurrence of Expr, keep track of whether production rules ① or ② is applied.

↳ e.g. if ① doesn't work, back track and try ②

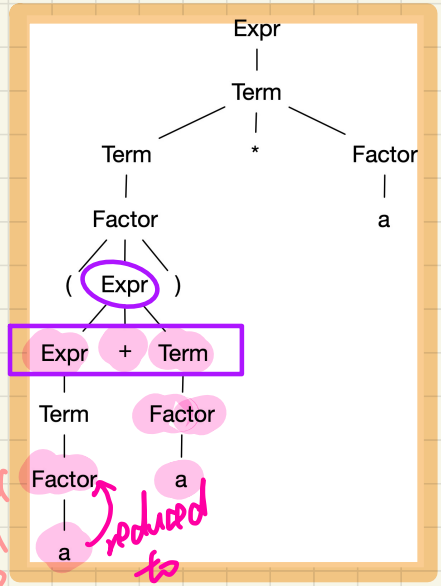
TDP: (a + a) * a



back track happened

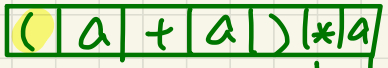
production rule ① should have been tried but could not work ⇒ back track and use ②

BUP: (a + a) * a



reduced to

Top-Down Parsing: Algorithm



At each level, keep track of the rule being used.

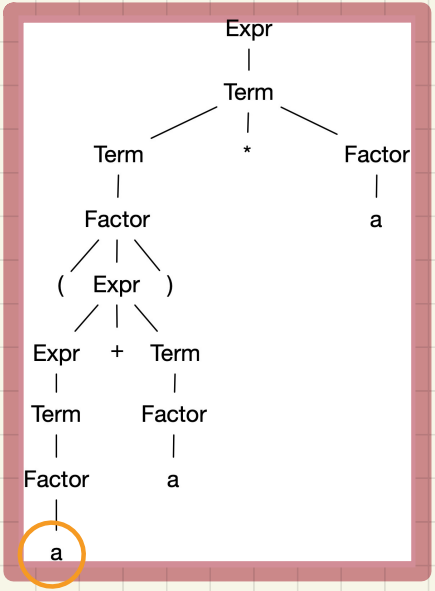
```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited 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	→	① Term * Factor
	|	② Factor
Factor	→	(Expr)
	|	a

TDP: (a + a) * a



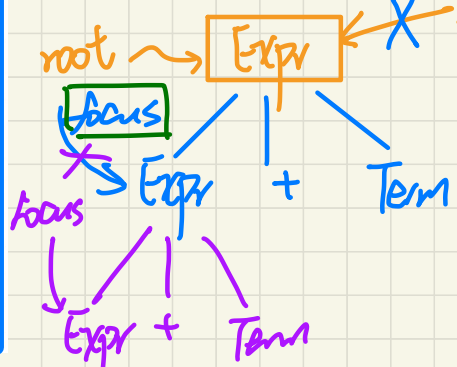
Top-Down Parsing: Discovering Leftmost Derivations (1)

```
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
```

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

Parse: a + a * a

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



word: "a"

infinite loop!

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:

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

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

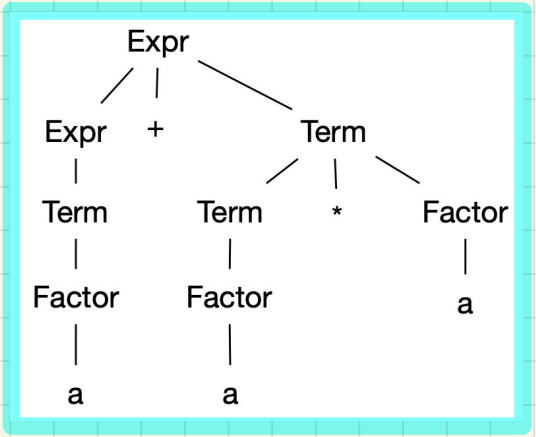
CFGs: Left-Recursive vs. Right-Recursive

Example: a + a * a

a ε

CFG with Left Recursions

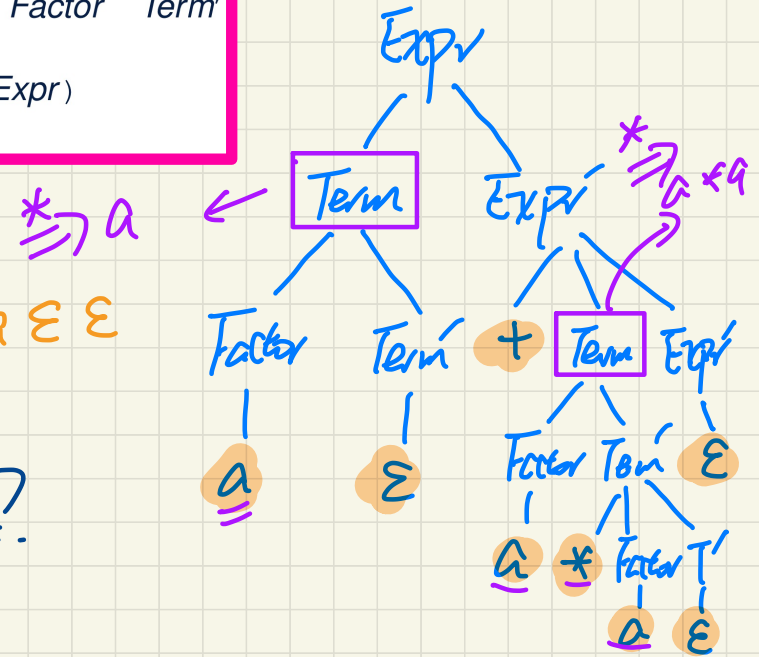
$$
\begin{aligned}
Expr &\rightarrow Expr + Term \mid Term \\
Term &\rightarrow Term * Factor \mid Factor \\
Factor &\rightarrow (Expr) \mid a
\end{aligned}
$$



CFG with Right Recursions

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

Draw Parse Tree



a ε + a * a ε ε

Exercise: LMD

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