CMPT 413
Computational Linguistics

Anoop Sarkar

`http://www.cs.sfu.ca/~anoop`

Context-free Grammars

- Set of rules by which valid sentences can be constructed.
- Example:

  Sentence → Noun Verb Object

  Noun → *trees* | *parsers*

  Verb → *are* | *grow*

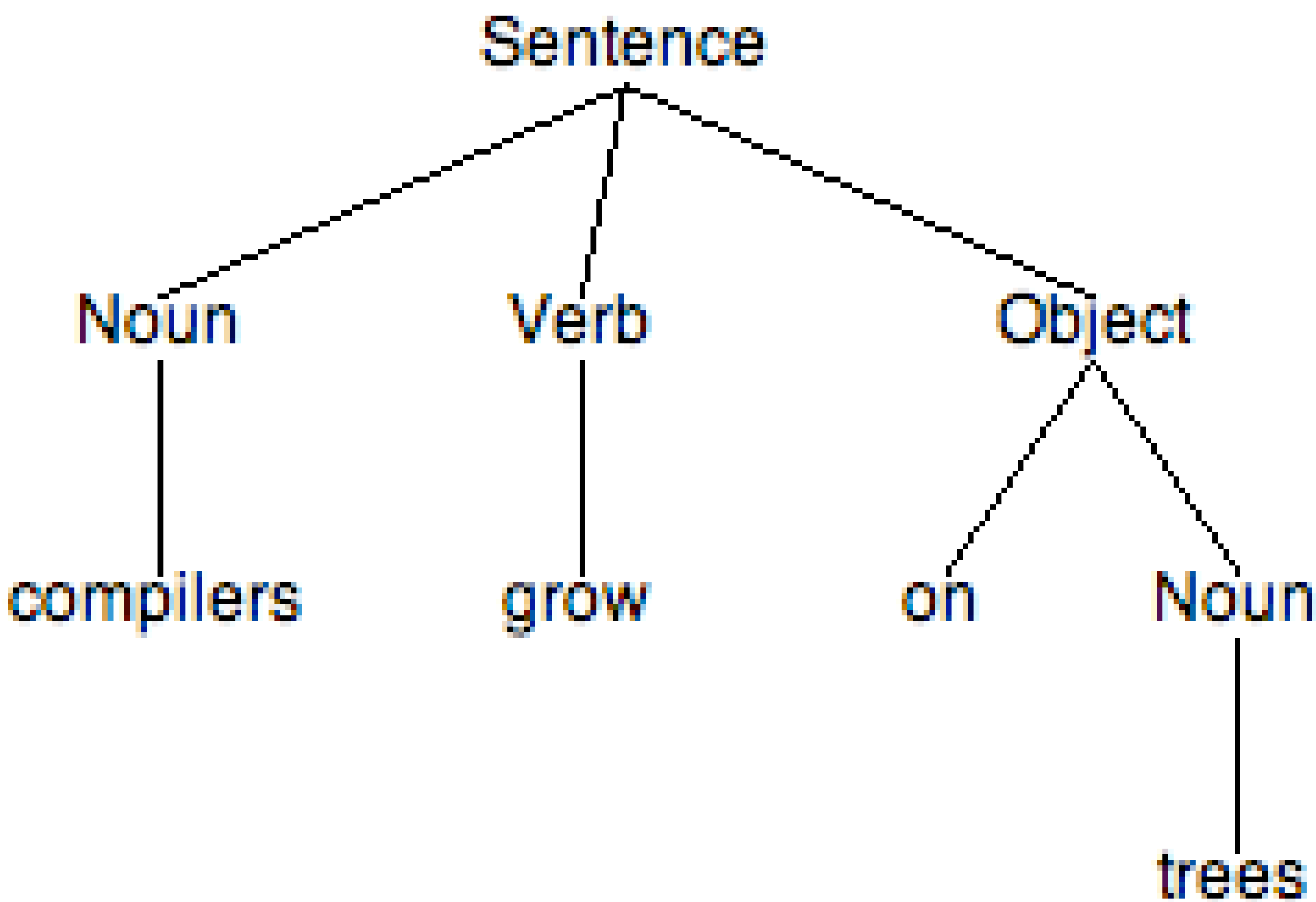
  Object → *on* Noun | Adjective

  Adjective → *slowly* | *interesting*
- What strings can Sentence *derive*?
- Syntax only – no semantic checking

Derivations of a CFG

- *parsers grow on trees*
- *parsers grow on* **Noun**
- *parsers grow* **Object**
- *parsers* **Verb Object**
- **Noun Verb Object**
- **Sentence**

Derivations and parse trees



Arithmetic Expressions

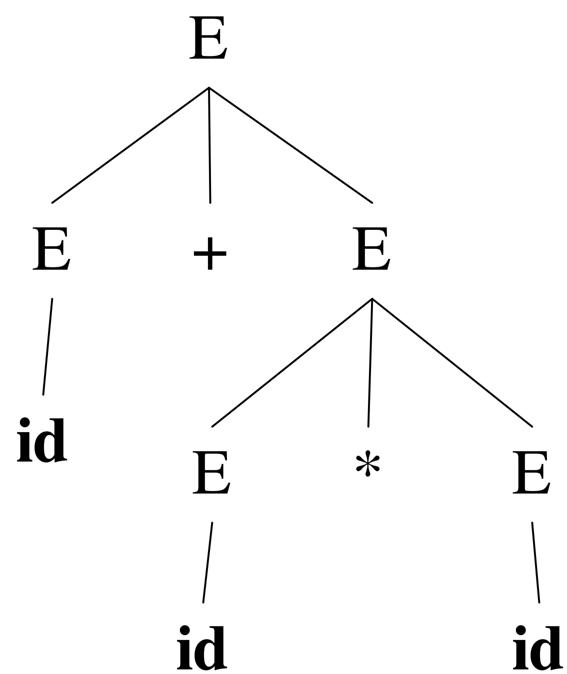
- $E \rightarrow E + E$
- $E \rightarrow E * E$
- $E \rightarrow (E)$
- $E \rightarrow - E$
- $E \rightarrow \textbf{id}$

Leftmost derivations for **id + id * id**

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow - E$
$E \rightarrow \textbf{id}$

- $E \Rightarrow E + E$

$\Rightarrow \textbf{id} + E$
$\Rightarrow \textbf{id} + E * E$
$\Rightarrow \textbf{id} + \textbf{id} * E$
$\Rightarrow \textbf{id} + \textbf{id} * \textbf{id}$



Leftmost derivations for
id + id * id

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow - E$

$E \rightarrow \mathbf{id}$

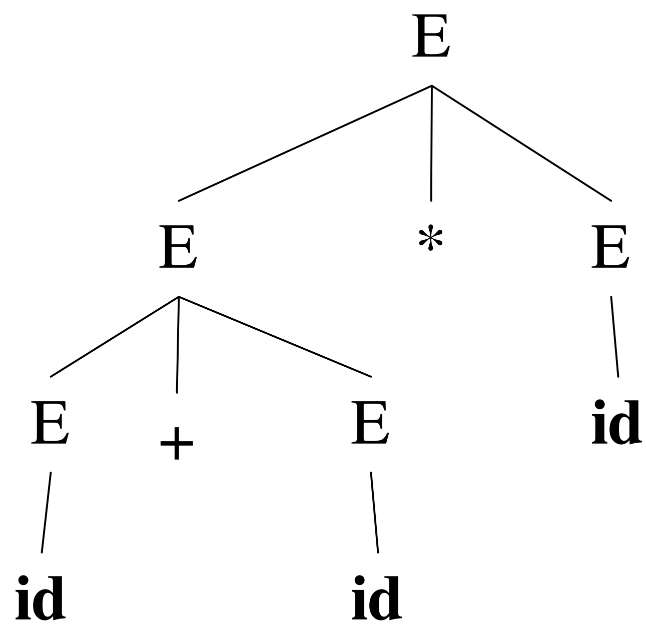
- $E \Rightarrow E * E$

$\Rightarrow E + E * E$

$\Rightarrow \mathbf{id} + E * E$

$\Rightarrow \mathbf{id} + \mathbf{id} * E$

$\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$



Rightmost derivation for
id + id * id

$E \rightarrow E + E$

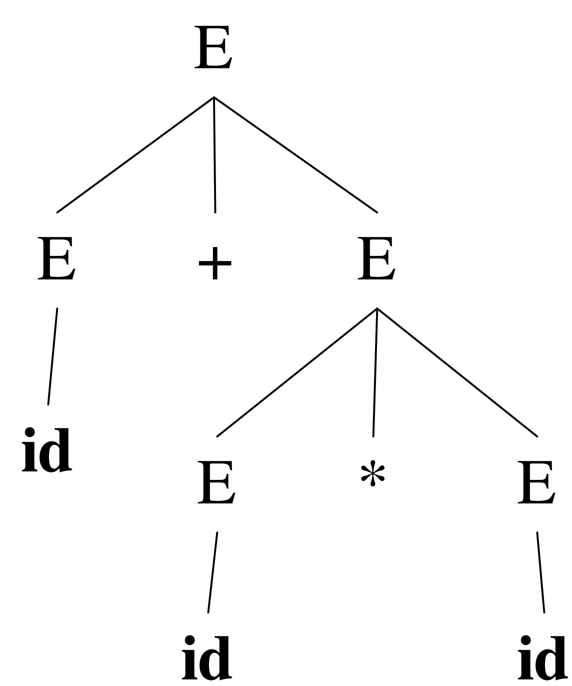
$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow - E$

$E \rightarrow \mathbf{id}$

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow E + E * E \\
&\Rightarrow E + E * \mathbf{id} \\
&\Rightarrow E + \mathbf{id} * \mathbf{id} \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$



Rightmost derivation for **id + id * id**

$E \rightarrow E + E$

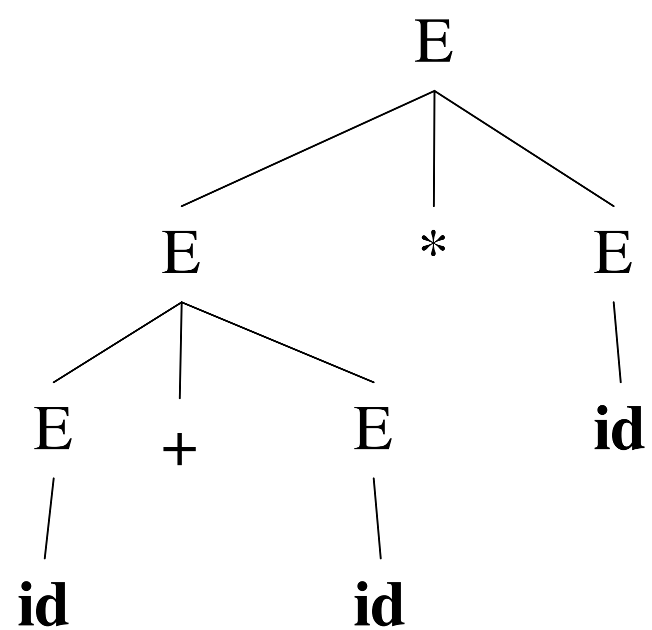
$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow - E$

$E \rightarrow \textbf{id}$

$$
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E * \textbf{id} \\
&\Rightarrow E + E * \textbf{id} \\
&\Rightarrow E + \textbf{id} * \textbf{id} \\
&\Rightarrow \textbf{id} + \textbf{id} * \textbf{id}
\end{aligned}
$$



Parsing - Roadmap

- Parser is a decision procedure: builds a parse tree
- Top-down vs. bottom-up
- Recursive-descent with backtracking
- Bottom-up parsing (CKY)
- Shift-reduce parsing
- Combining top-down and bottom-up: Earley parsing

Top-Down vs. Bottom Up

Grammar: $S \rightarrow A\ B$

$A \rightarrow c \mid \varepsilon$

$B \rightarrow cbB \mid ca$

Input String: ccbca

Top-Down/leftmost		Bottom-Up/rightmost	
$S \Rightarrow AB$	$S \rightarrow AB$	$ccbca \Leftarrow Acbca$	$A \rightarrow c$
$\Rightarrow cB$	$A \rightarrow c$	$\Leftarrow AcbB$	$B \rightarrow ca$
$\Rightarrow ccbB$	$B \rightarrow cbB$	$\Leftarrow AB$	$B \rightarrow cbB$
$\Rightarrow ccbca$	$B \rightarrow ca$	$\Leftarrow S$	$S \rightarrow AB$

Top-Down: Backtracking

$S \rightarrow A\ B$

$A \rightarrow c \mid \varepsilon$

$B \rightarrow cbB \mid ca$

True/False

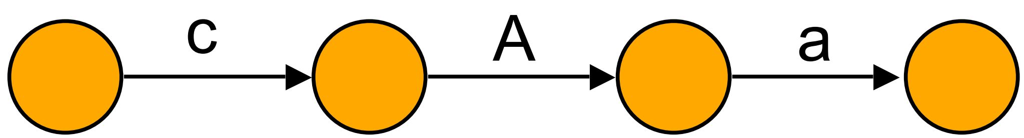
$S \Rightarrow^* cbca$?

S	cbca	try $S \rightarrow AB$
AB	cbca	try $A \rightarrow c$
cB	cbca	match c
B	bca	dead-end, try $A \rightarrow \varepsilon$
εB	cbca	try $B \rightarrow cbB$
cbB	cbca	match c
bB	bca	match b
B	ca	try $B \rightarrow cbB$
cbB	ca	match c
bB	a	dead-end, try $B \rightarrow ca$
ca	ca	match c
a	a	match a, Done!

Transition Diagram

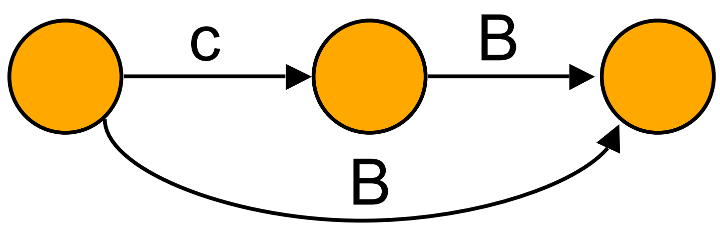
$S \rightarrow cAa$

S:



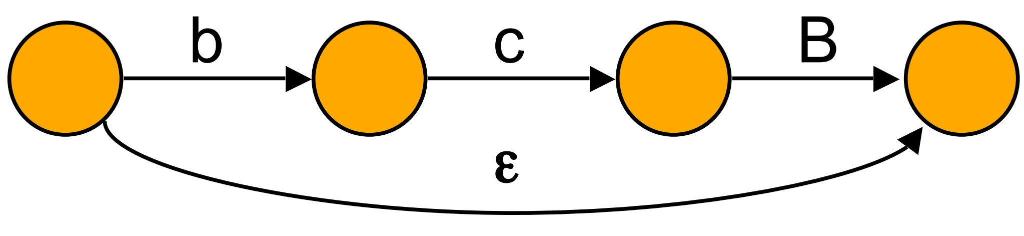
$A \rightarrow cB \mid B$

A:



$B \rightarrow bcB \mid \varepsilon$

B:



Leftmost derivation for **id + id * id**

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow - E$

$E \rightarrow \mathbf{id}$

$E \Rightarrow E + E$

$\Rightarrow \mathbf{id} + E$

$\Rightarrow \mathbf{id} + E * E$

$\Rightarrow \mathbf{id} + \mathbf{id} * E$

$\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$

Bottom-up parsing overview

- Start from terminal symbols, search for a path to the start symbol
- Apply shift and reduce actions: postpone decisions
- LR parsing:
  - L: left to right parsing
  - R: rightmost derivation (in reverse or bottom-up)
- Useful for deterministic parsing (e.g. in compilers for programming languages)

Rightmost derivation for **id + id * id**

$E \rightarrow E + E$	$E \Rightarrow E * E$	
$E \rightarrow E * E$	$\Rightarrow E * \textbf{id}$	
$E \rightarrow (E)$	$\Rightarrow E + E * \textbf{id}$	
$E \rightarrow - E$	$\Rightarrow E + \textbf{id} * \textbf{id}$	reduce with $E \rightarrow \textbf{id}$
$E \rightarrow \textbf{id}$	$\Rightarrow \textbf{id} + \textbf{id} * \textbf{id}$	shift

Ambiguity

- Grammar is ambiguous if more than one parse tree is possible for some sentences
- Examples in English:
  - Two sisters reunited after 18 years in checkout counter
- It is undecidable to check using an algorithm whether a grammar is ambiguous

Parsing CFGs

- Consider the problem of parsing with arbitrary CFGs
- For any input string, the parser has to produce a parse tree
- The simpler problem: print **yes** if the input string is generated by the grammar, print **no** otherwise
- This problem is called *recognition*

CKY Recognition Algorithm

- The Cocke-Kasami-Younger algorithm
- As we shall see it runs in time that is polynomial in the size of the input
- It takes space polynomial in the size of the input
- **Remarkable fact:** it can find all possible parse trees (exponentially many) in polynomial time

Chomsky Normal Form

- Before we can see how CKY works, we need to convert the input CFG into Chomsky Normal Form
- CNF is one of many grammar transformations that *preserve* the language
- CNF means that the input CFG G is converted to a new CFG G' in which all rules are of the form:

  $A \rightarrow B\ C$

  $A \rightarrow a$

Epsilon Removal

- First step, remove epsilon rules

  $A \rightarrow B\ C$

  $C \rightarrow \varepsilon \mid C\ D \mid a$

  $D \rightarrow b \quad B \rightarrow b$

- After ε-removal:

  $A \rightarrow B \mid B\ C\ D \mid B\ a$

  $C \rightarrow D \mid C\ D\ D \mid a\ D \mid a$

  $D \rightarrow b \quad B \rightarrow b$

Removal of Chain Rules

- Second step, remove chain rules

  $A \rightarrow B\ C \mid C\ D\ C$

  $C \rightarrow D \mid a$

  $D \rightarrow d \qquad B \rightarrow b$

- After removal of chain rules:

  $A \rightarrow B\ a \mid B\ D \mid a\ D\ a \mid a\ D\ D \mid D\ D\ a \mid D\ D\ D$

  $D \rightarrow d \qquad B \rightarrow b$

Eliminate terminals from RHS

- Third step, remove terminals from the rhs of rules

  $A \rightarrow B\ a\ C\ d$

- After removal of terminals from the rhs:

  $A \rightarrow B\ N_1\ C\ N_2$

  $N_1 \rightarrow a$

  $N_2 \rightarrow d$

Binarize RHS with Nonterminals

- Fourth step, convert the rhs of each rule to have two non-terminals

  $A \rightarrow B\ N_1\ C\ N_2$

  $N_1 \rightarrow a$

  $N_2 \rightarrow d$

- After converting to binary form:

$A \rightarrow B\ N_3$	$N_1 \rightarrow a$
$N_3 \rightarrow N_1\ N_4$	$N_2 \rightarrow d$
$N_4 \rightarrow C\ N_2$	

CKY algorithm

- We will consider the working of the algorithm on an example CFG and input string
- Example CFG:

  $S \rightarrow A\ X \mid Y\ B$

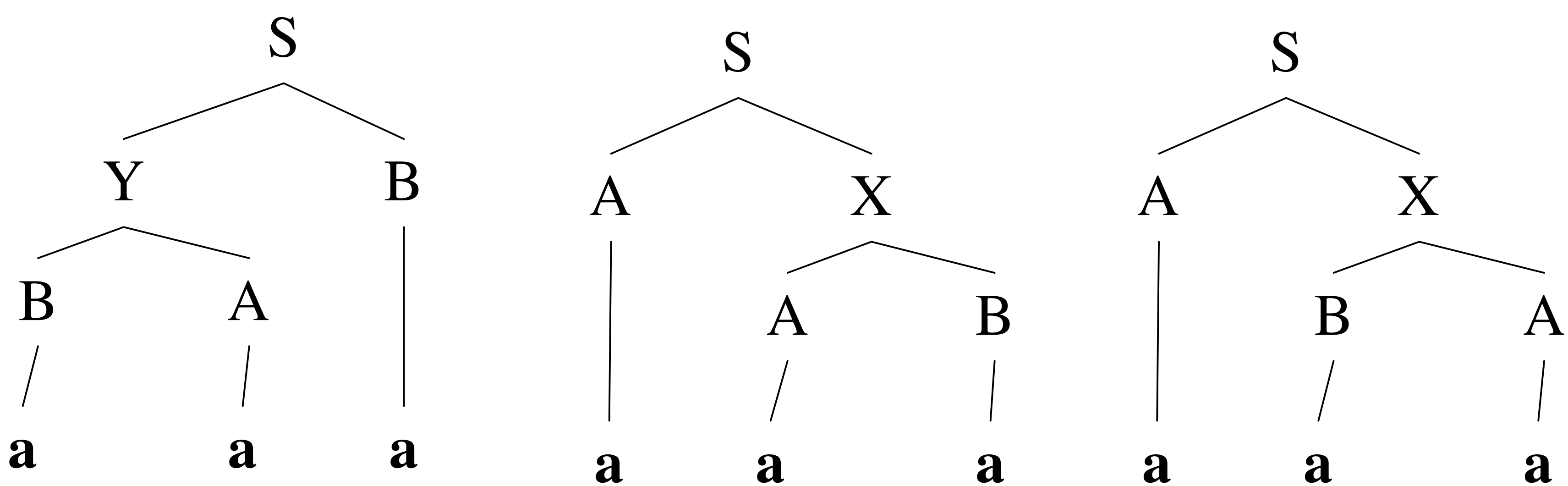
  $X \rightarrow A\ B \mid B\ A \qquad Y \rightarrow B\ A$

  $A \rightarrow a \qquad B \rightarrow a$
- Example input string: *aaa*

CKY Algorithm

	0	1	2	3
0		A, B $A \rightarrow a$ $B \rightarrow a$	X, Y $X \rightarrow A\ B \mid B\ A$ $Y \rightarrow B\ A$	S $S \rightarrow A_{(0,1)}\ X_{(1,3)}$ $S \rightarrow Y_{(0,2)}\ B_{(2,3)}$
1			A, B $A \rightarrow a$ $B \rightarrow a$	X, Y $X \rightarrow A\ B \mid B\ A$ $Y \rightarrow B\ A$
2				A, B $A \rightarrow a$ $B \rightarrow a$
		a	a	a

Parse trees



CKY Algorithm

Input string **input** of size *n*

Create a 2D table **chart** of size n^2

for i=0 **to** n-1

    chart[i][i+1] = A **if** there is a rule A → a and **input**[i]=a

for j=2 **to** N

    for i=j-2 **downto** 0

        for k=i+1 **to** j-1

            chart[i][j] = A **if** there is a rule A → B C **and**
              chart[i][k] = B **and chart**[k][j] = C

return *yes* **if chart**[0][n] has the start symbol

else return *no*

CKY algorithm summary

- Parsing arbitrary CFGs
- For the CKY algorithm, the time complexity is $O(|G|^2 n^3)$
- The space requirement is $O(n^2)$
- The CKY algorithm handles arbitrary ambiguous CFGs
- All ambiguous choices are stored in the chart
- For compilers we consider parsing algorithms for CFGs that do not handle ambiguous grammars

Parsing - Summary

- Parsing arbitrary CFGs: $O(n^3)$ time complexity
- Top-down vs. bottom-up
  - Recursive-descent parsing
  - Shift-reduce parsing
- Earley parsing
- Ambiguous grammars result in parser output with multiple parse trees for a single input string