CMPT 413
Computational Linguistics

Anoop Sarkar
http://www.cs.sfu.ca/~anoop

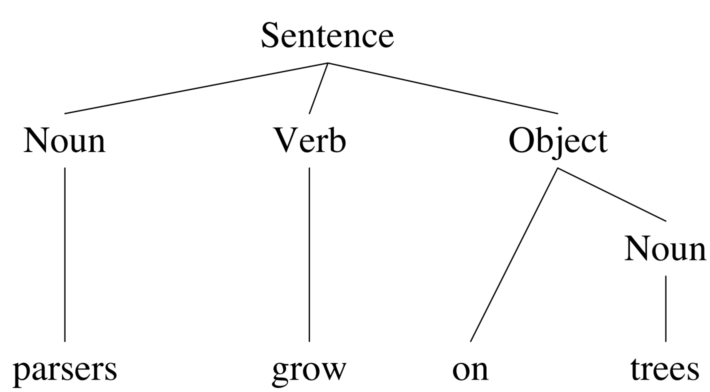
Context-free Grammars

- 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



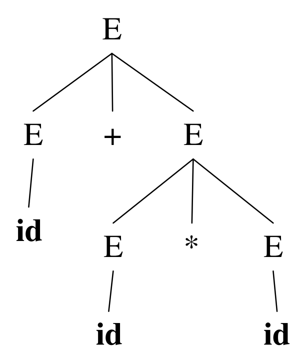
Arithmetic Expressions

- E → E + E
- E → E * E
- E → (E)
- E → - E
- E → **id**

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

E → E + E
E → E * E
E → (E)
E → - E
E → **id**

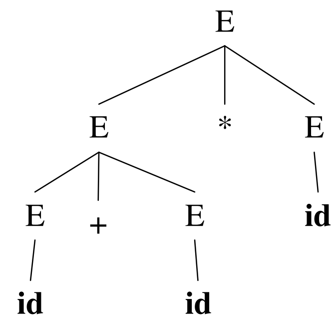
- E ⇒ E + E
  ⇒ **id** + E
  ⇒ **id** + E * E
  ⇒ **id** + **id** * E
  ⇒ **id** + **id** * **id**



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

E → E + E
E → E * E
E → (E)
E → - E
E → **id**

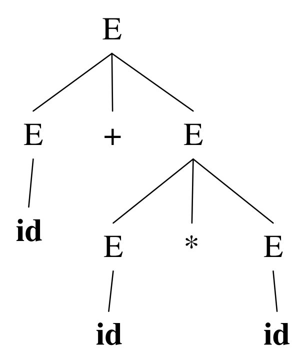
- E ⇒ E * E
⇒ E + E * E
⇒ **id** + E * E
⇒ **id** + **id** * E
⇒ **id** + **id** * **id**



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

E → E + E
E → E * E
E → (E)
E → - E
E → **id**

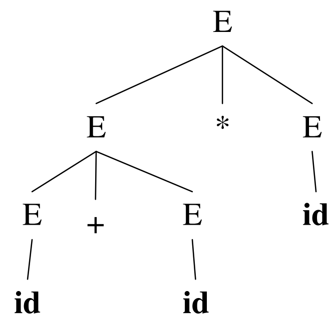
E ⇒ E + E
⇒ E + E * E
⇒ E + E * **id**
⇒ E + **id** * **id**
⇒ **id** + **id** * **id**



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

E → E + E
E → E * E
E → (E)
E → - E
E → **id**

E ⇒ E * E
⇒ E * **id**
⇒ E + E * **id**
⇒ E + **id** * **id**
⇒ **id** + **id** * **id**



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: $S \rightarrow A\ B$ Input String: ccbca

$A \rightarrow c \mid \varepsilon$

$B \rightarrow cbB \mid ca$

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

$S \Rightarrow^* cbca?$

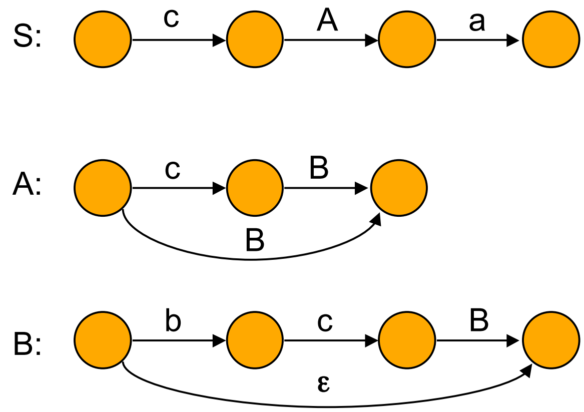
S	cbca	try $S \rightarrow AB$
AB	cbca	try $A \rightarrow c$
cB	cbca	match c
B	bca	dead-end, try $A \rightarrow \varepsilon$
εB	cbca	try $B \rightarrow cbB$
cbB	cbca	match c
bB	bca	match b
B	ca	try $B \rightarrow cbB$
cbB	ca	match c
bB	a	dead-end, try $B \rightarrow ca$
ca	ca	match c
a	a	match a, Done!

Transition Diagram

$S \rightarrow cAa$

$A \rightarrow cB \mid B$

$B \rightarrow bcB \mid \varepsilon$



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 → E + E	E ⇒ E * E	
E → E * E	⇒ E * **id**	
E → (E)	⇒ E + E * **id**	
E → - E	⇒ E + **id** * **id**	reduce with E → **id**
E → **id**	⇒ **id** + **id** * **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 → B C

  A → a

Epsilon Removal

- First step, remove epsilon rules

  A → B C

  C → ε | C D | a

  D → b   B → b

- After ε-removal:

  A → B | B C D | B a | BC

  C → D | C D D | a D | C D | a

  D → b   B → b

Removal of Chain Rules

- Second step, remove chain rules

  A → B C | C D C

  C → D | a

  D → d B → b

- After removal of chain rules:

  A → B a | B D | a D a | a D D | D D a | D D D

  D → d B → b

Eliminate terminals from RHS

- Third step, remove terminals from the rhs of rules

  A → B a C d

- After removal of terminals from the rhs:

  A → B N_1 C N_2

  N_1 → a

  N_2 → 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:

  $A \rightarrow B\ N_3$ $\quad$ $N_1 \rightarrow a$

  $N_3 \rightarrow N_1\ N_4$ $\quad$ $N_2 \rightarrow d$

  $N_4 \rightarrow C\ N_2$

CKY algorithm

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

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

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

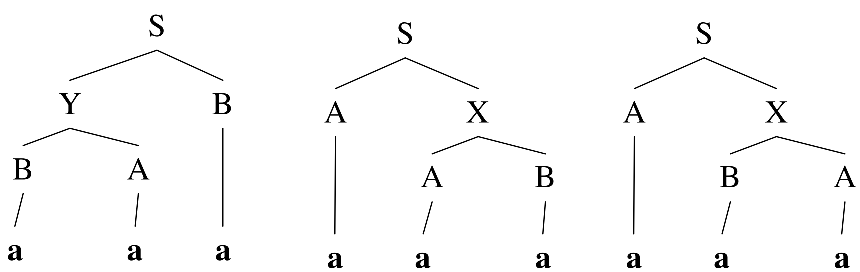
  $A \rightarrow a$ $\quad$ $B \rightarrow a$

- Example input string: *aaa*

CKY Algorithm

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

Parsing - Additional Results

- $O(n^2)$ time complexity for linear grammars
  - All rules are of the form S → aSb or S → a
  - Reason for $O(n^2)$ bound is the linear grammar normal form: A → aB, A → Ba, A → B, A → a
- Left corner parsers
  - extension of top-down parsing to arbitrary CFGs
- Earley's parsing algorithm
  - $O(n^3)$ worst case time for arbitrary CFGs just like CKY
  - $O(n^2)$ worst case time for unambiguous CFGs
  - $O(n)$ for specific unambiguous grammars (e.g. S → aSa | bSb | ε)

Non-CF Languages

$$L_1 = \{wcw \mid w \in (a|b)*\}$$

$$L_2 = \{a^n b^m c^n d^m \mid n \geq 1, m \geq 1\}$$

$$L_3 = \{a^n b^n c^n \mid n \geq 0\}$$

CF Languages

$$L_4 = \{wcw^R \mid w \in (a|b)*\}$$

$$S \rightarrow aSa \mid bSb \mid c$$

$$L_5 = \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

$$S \rightarrow aSd \mid aAd$$

$$A \rightarrow bAc \mid bc$$

Context-free languages and Pushdown Automata

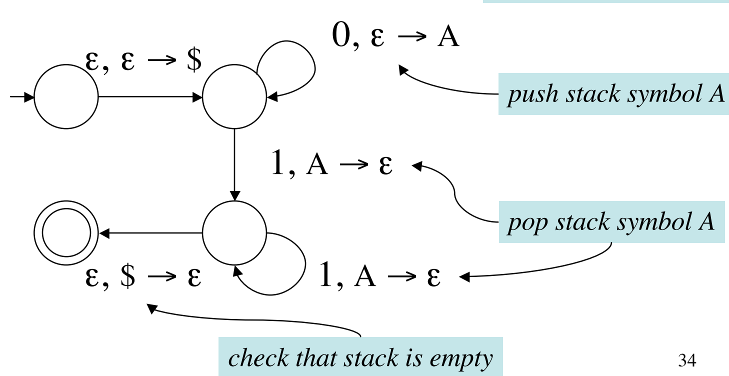
- Recall that for each regular language there was an equivalent finite-state automaton
- The FSA was used as a recognizer of the regular language
- For each context-free language there is also an automaton that recognizes it: called a **pushdown automaton (pda)**

Pushdown Automata

- PDA has
  - an alphabet (terminals) and
  - stack symbols (like non-terminals),
  - a finite-state automaton, and
  - stack

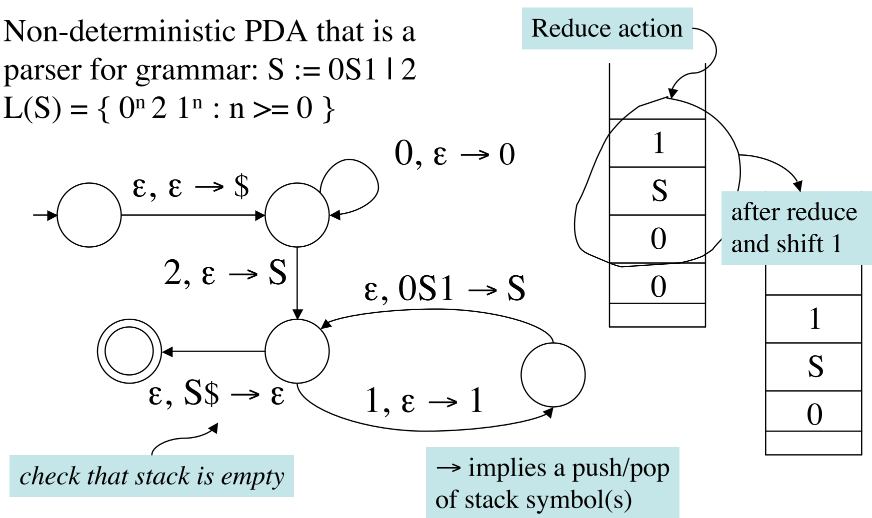
e.g. PDA for language
L = { 0^n1^n : n >= 0 }

→ implies a push/pop of stack symbol(s)



Shift-reduce parser as a pda

Non-deterministic PDA that is a parser for grammar: S := 0S1 | 2
L(S) = { $0^n 2\ 1^n$: n >= 0 }



Context-free languages and Pushdown Automata

- Similar to FSAs there are non-deterministic pda and deterministic pda
- Unlike in the case of FSAs we cannot always convert a npda to a dpda
- The construction of a pda will provide us with the algorithm for parsing (take in strings and provide the parse tree)

CKY algorithm for PCFGs

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

  S → A X (0.3) | Y B (0.7)

  X → A B (0.1) | B A (0.9) Y → B A (1.0)

  A → a (1.0) B → a (1.0)
- Example input string: *aaa*

CKY Algorithm

Max(0.1, 0.9)

0.3 * 0.9 = 0.27
Max(0.27, 0.7)

	0	1	2	3
0		A 1.0, B 1.0 A → a $_{1.0}$ B → a $_{1.0}$	X 0.9, Y 1.0 X → A B $_{0.1}$ | B A $_{0.9}$ Y → B A $_{1.0}$	S 0.7 S → $A_{(0,1)}$ $X_{(1,3)}$ $_{0.3}$ S → $Y_{(0,2)}$ $B_{(2,3)}$ $_{0.7}$
1			A 1.0, B 1.0 A → a $_{1.0}$ B → a $_{1.0}$	X 0.9, Y 1.0 X → A B $_{0.1}$ | B A $_{0.9}$ Y → B A $_{1.0}$
2				A 1.0, B 1.0 A → a $_{1.0}$ B → a $_{1.0}$
		a	a	a

Parse trees

PCFG is consistent:
0.7 + 0.27 + 0.03 = 1.0