# EE 583
# PATTERN RECOGNITION

Syntactic Pattern Recognition via Parsing

Parsing

CYK Parsing Algorithm

Higher Dimensional (Tree) Grammars

Grammatical Inference

## Introduction

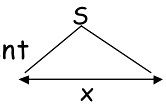
- After a grammar is constructed to generate a language, next step is to design a recognizer that will recognize patterns (represented by strings) generated by this grammar
- One way of recognition is as follows : Given the description of a pattern as a string produced by a class-specific grammar, the objective is to determine which $L(G_i)$ i=1,2,…,c the string belongs
  - One way of recognition is by matching the string against each pattern in each library. The class membership of this string can be found with a huge computational complexity
  - Another way of recognition is *parsing*

# Parsing (1/2)

- Parsing is a fundamental concept which determines whether the input pattern (string) is syntactically well formed in the context of theprespecified grammars (parsing=recognizing=syntax analyzing)
- Given a string (sentence) x and a grammar G, a parser should construct a derivation of x and find a corresponding derivation tree of the tree (complete description of patterns andsubpatterns)
- Usually different parsing methods are associated with restricted classes of grammars.
- The constraints in restricted classes yield efficient parser complexities at the expense of losing representational flexibility

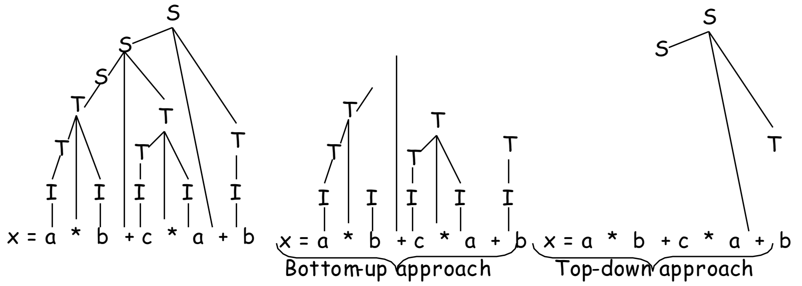
# Parsing (2/2)

- Given a string x and a grammar G, if self-consistent tree of derivations can fill the triangle, x is element of the language obtained by this class




- Example : $G=\{V_T,V_N,P,S\}$ where $V_N=\{S,T\}$, $V_T=\{I=\{a,b,c\},+,*\}$, P as
  (1) $S \rightarrow T$ (2) $T \rightarrow T*I$ (3) $S \rightarrow S+T$ (4) $T \rightarrow I$

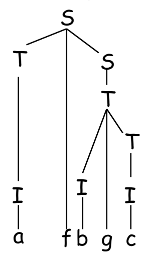
  Derivation tree for the sentence : a * b + c * a + b



# Top-down Parsing

- Beginning from right part of production $A \to X_1X_2...X_n$, the goal is
  - If $X_1$ is a terminal symbol → string must begin with this symbol
  - If $X_1$ is a nonterminal symbol → a subgoal is obtained : whether the head of the string may be reduced to $X_1$
  - If a match is found for $X_1$ → continue with $X_2$ ...
- If no match is found for some $X_i$ → report to higher level and apply an alternative production $A \to X'_1X'_2...X'_n$

Example: $G=\{V_T,V_N,P,S\}$ where
$V_N=\{S,T,I\}$, $V_T=\{a,b,c,f,g\}$, P as
(1) S→T (2) S→TfS (3) T→IgT
(4) T→I (5) I→a (6) I→b (7) I→c
Check x=afbgc

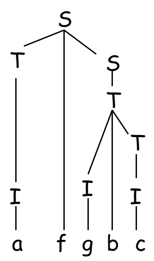



(2)
(4)
(5)
(1)
(3)
(6)
(4)
(7)

# Bottom-up Parsing

- Start from the string (sentence), s, apply productions backward
- No general rule but some rule-of-thumbs :
  - Begin from leftmost symbol of the string
  - Process A→b cases where A, b are elements of $V_N$ and $V_T$, respectively

Example: $G=\{V_T,V_N,P,S\}$ where
$V_N=\{S,T,I\}$, $V_T=\{a,b,c,f,g\}$, P as
(1) S→T (2) S→TfS (3) T→IgT
(4) T→I (5) I→a (6) I→b (7) I→c

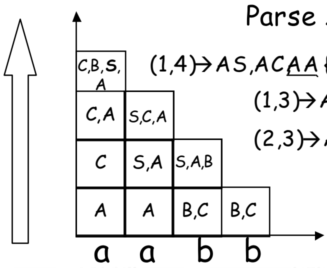



(2)
(1)
(4)
(7)
(6)
(4)
(5)

- Bottom-up procedure is not efficient because a large number of false trails or errors may be made

## Cocke-Younger-Kasami (CYK) Parsing Algorithm (1/2)

- CYK algorithm requires the grammar to be
  - Context-free (CFG) : $V_N \rightarrow (V_N \cup V_T)^* - \varepsilon$
  - Chomsky Normal Form (CNF) : $A \rightarrow BC$, $A \rightarrow a$
- Example : S→AB|BB A→CC|AB|a B→BB|CA|b C→BA|AA|b
  Parse x=aabb



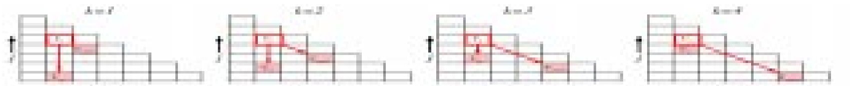

(1,4)→AS,AC*AA* {1+3} (1,4)→CS,*CA*,CB {2+2} (1,4)→CB,*AB*,CC,AC {3+1}

(1,3)→AS,*AA* {1+2} (1,3)→CB,*CC* {2+1}

(2,3)→AS,*AA*,*AB* {1+2} (2,3)→SB,SC,*AB*,AC {2+1}

(1,2)→*AA* (2,2)→*AB*,AC (3,2)→*BB*,CB,BC,*CC*

A→a A→a B,C→b B,C→b

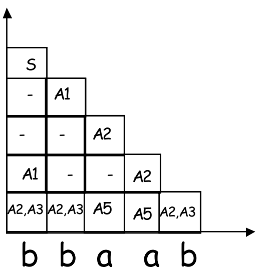


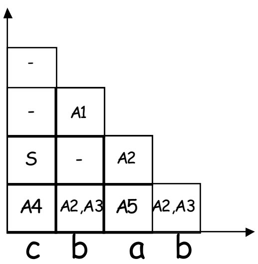
## Cocke-Younger-Kasami (CYK) Parsing Algorithm (2/2)

- Convert a FSG to CFG in order to utilize CYK algorithm
- Example : FSG : $S \rightarrow bA_1 | cA_2$ $A_1 \rightarrow bA_2$ $A_2 \rightarrow b | a\ A_2$

FSG→CFG :

$S \rightarrow A_3A_1$ $A_3 \rightarrow b$ $S \rightarrow A_4A_2$ $A_4 \rightarrow c$ $A_1 \rightarrow A_3A_2$ $A_2 \rightarrow b$ $A_2 \rightarrow A_5A_2$ $A_5 \rightarrow a$



## Transition Networks in Parsing (1/2)

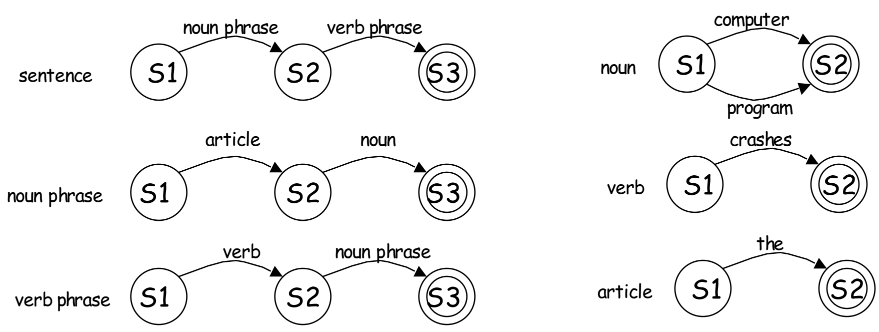
- We have seen a graphical representation of a FSG; similar ideas can be applied to CFG by transition networks (TN)
- TN is a directional graph to show productions via
  - Nodes : representing states
  - Arc : labeled to represent either nonterminals or terminals
- TN parses an input string by
  - starting from an initial state,
  - sequentially checking each symbol in the input string against label of an arc emanating from present node
  - if a match is found attention focuses on this new node and matching process is repeated
  - if a goal state is reached, parsing stops, successfully
  - if it reaches a state in TN without an outgoing arc → failure, apply backtracking to the previous node

## Transition Networks in Parsing (2/2)

- <u>Example</u>: $G=\{V_T,V_N,P,S\}$, $V_T$ ={the,computer,program,crashes}, S : <sentence>,

$V_N$ ={<sentence>, <noun phrase>, <verb phrase>, <article>, <noun>, < verb>}

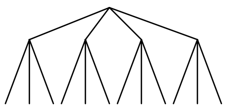
P : <sentence> →<noun phrase>+<verb phrase> , <noun phrase> → <article>+ <noun>, <verb phrase> →<verb>+<noun phrase>, <verb> →crashes, <article> →the, <noun> →computer|program



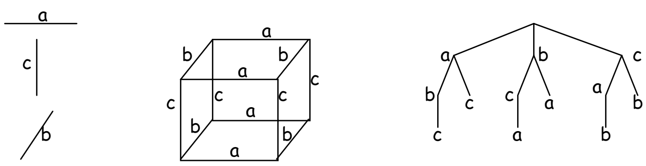
## Higher Dimensional Grammars

- There exist grammars other than "string grammars"
- They are useful in 2-D or higher-D pattern representation applications, but they have more complex production rules
- In 2D, it is possible to define "attachment points"
- The most popular approach :
  - Tree grammar

## Tree Grammars (1/4)

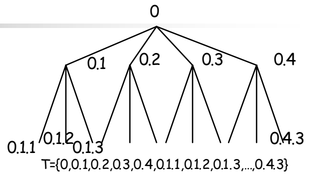


- A *tree* is a directed acyclical graph
- Trees store pattern info into two ways :
  - Nodes : store pattern primitive info
  - Arcs : reflect relational info between nodes
- Tree structures are useful for
  - Structural representation that involve hierarchical decompositions
  - Describing complex patterns using primitives with multiple connection points



# Tree Grammars (2/4)

- Enumeration of all the nodes of a general tree can be obtained by the alphabet V={0,1,2,…,}U{.}

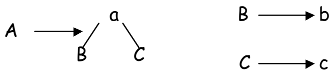



- In order to formally characterize tree grammars, tree generatio should be constrained
- Ranked alphabets is a pair (V,r) which maps alphabet symbols to nonnegative integers and they are used to relate node labels to trees :
  - e.g. V : labels nodes, r : denotes outdegree of the node
    T={(0,4),(0.1,3),(0.2,3),(0.3,3),(0.4,3),(0.1.1,0),(0.1.2,0)…(0.4.3,0)}
- A tree grammar is a four-tuple G={V,r,P,S} where V=$V_T$U$V_N$ , P is a set of productions involving trees, S is an element of "starting" (often single-node) tree

# Tree Grammars (3/4)

- For tree grammars, replacement rules to form productions mean a tree is replaced by another tree
- There are two options for tree grammar productions
  - Rewriting of a (sub)tree with nodes initially labelled as nonterminal by exactly the same tree with terminal nodes

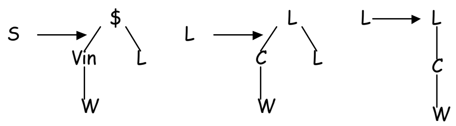

  - Expansive production form (including a special case : expansion terminated)



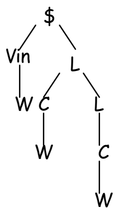

- A derivation using a tree grammar : $T_\alpha \overset{a}{\Rightarrow} T_\beta$
- There is significance of making the derivation at node a :
  - e.g. a tree production rule $T_i \rightarrow T_j$ should exist, while $T_i$,$T_j$ are subtrees of $T_\alpha$ and $T_\beta$ at node a, respectively

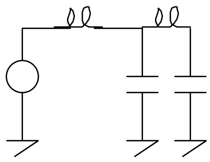
# Tree Grammars (4/4)

- Example : A tree grammar to generate LC networks :
  $G=\{V,r,P,S\}$ where $V_T=\{Vin,L,C,W,\$\}$ and $r(Vin)=1$, $r(L)=\{2,1\}$, $r(w)=0$, $r(\$)=2$; P:




- After applying 3 productions consecutively, one can obtain :



# Grammatical Inference

- Learning a grammar by examples (Grammatical Inference) is much preferable compared to a design by hand
- Grammatical inference (GI) is the supervised learning approach in SyntPR
- A general algorithm for GI
  1. Initialize with an initial grammar $G_0$
  2. Find positive and negative examples for the desired grammar
  3. For every $x$, check whether $G_0$ parses $x$
  4. If not, add a new `simple' production rule to $G_0$ that parses $x$, and does not parse the whole $x$
  5. Repeat until the resulting grammar achieves correct parsing
- "Adding a new production rule" is usually achieved by :
  - choosing from a predetermined set with simple rules first
  - using a specific initial knowledge about the underlying model
  - using a constrained grammar, such as FSG

# Grammatical Inference : Examples

- Inferring a single string, x∊aaab, into a FSG :
  - Minimum $V_T$ and $V_N$ should be $V_T$={a,b,c} ; $V_N$={S,$A_1$,$A_2$}
  - From left to right, following productions will generate x

  $S \to cA_1$ $A_1 \to aA_2$ $A_2 \to aA_3$ $A_3 \to aA_4$ $A_4 \to b$ (note $A_{3,4}$ are redundant)

- Inferring several strings into a FSG :

  Following strings are given : D={bbaab,caab,bbab,cab,bbb,cb}
  - From left to right, following productions will generate the set

    $S \to bA_1$ $S \to cA_4$ $A_1 \to bA_2$ $A_2 \to b$ $A_2 \to aA_3$ $A_3 \to b$ $A_3 \to aA_{3'}$ $A_{3'} \to b$ $A_4 \to b$ $A_4 \to aA_5$ $A_5 \to aA_{5'}$ $A_5 \to b$ $A_{5'} \to b$
  - Merge redundant generations :

    $S \to bA_1$ $S \to cA_4$ $A_1 \to bA_2$ $A_2 \to b$ $A_2 \to aA_2$ $A_4 \to b$ $A_4 \to aA_4$
  - A final refinement gives :

    $S \to bA_1$ $S \to cA_2$ $A_1 \to bA_2$ $A_2 \to b$ $A_2 \to aA_2$