Syntactic Pattern Recognition

In many cases, statistical pattern recognition does not offer good performance because statistical features do not (and cannot) represent sufficient information that is needed.

In SYNTPR, structure is paramount.

Classification may be based on measures of pattern structural similarity

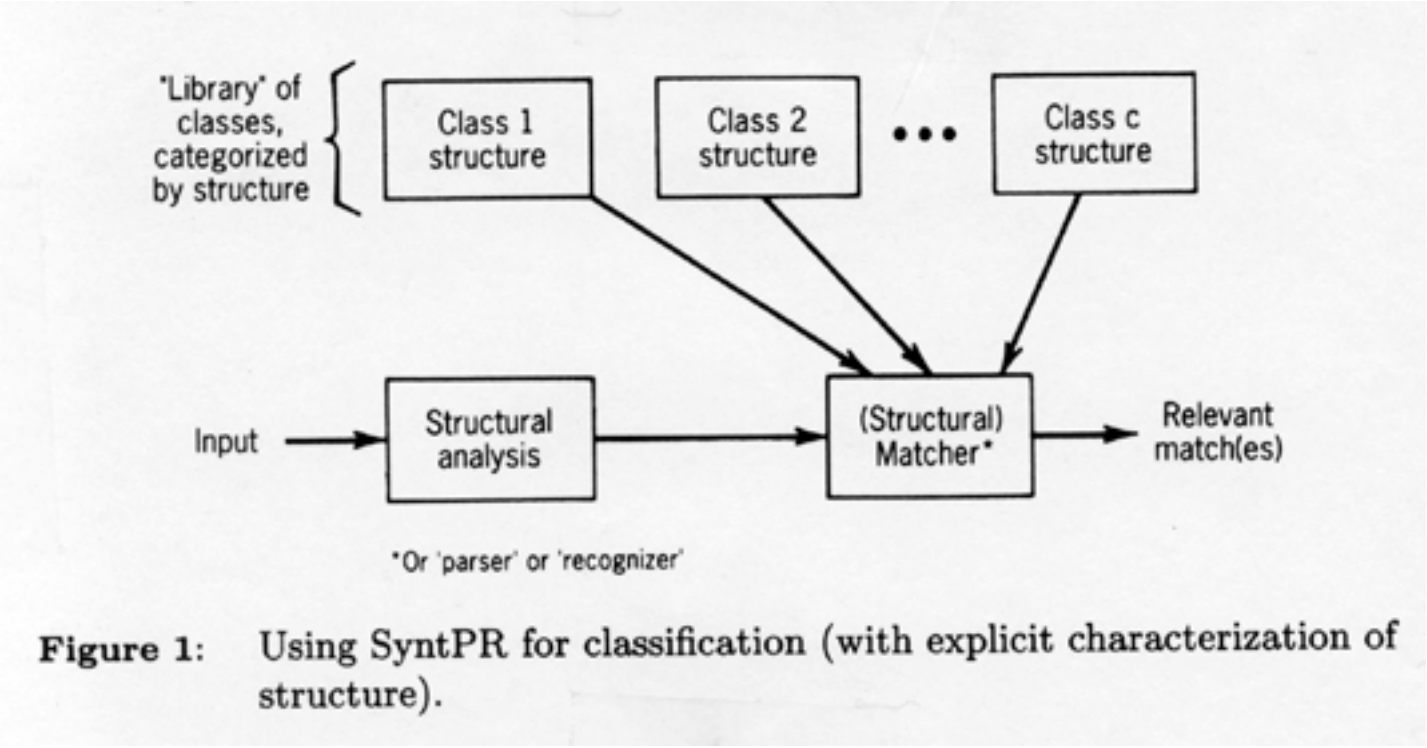


Figure 1: Using SyntPR for classification (with explicit characterization of structure).

Quantifying Structure

- Formal Grammars
- Relational Descriptions (Graphs)

Syntactic Recognition

- Parsing (formal grammars)
- Relational graph matching
  (Attributed relational graphs)

Hirerarchical Approaches

Often, SYNTPR techniques are hierarchical ... decomposition of complex patterns into simpler patterns - example: written language.

Grammar-Based Approaches (Formal grammars)

String grammars:

Linear strings of terminal symbols (terminals)

Definitions and Conventions

Alphabet V

$V = \{a,b,c,...,z\}$

Concatenation a•b produces a sequence ab

String over V

- a single symbol
- concatenation of zero or more symbols

Length of string s $|s|$

String x = aaaa ,,, $a = a^n$

Empty string e $|e| = 0$

$e \bullet x = x \bullet e = x$

Set of strings of length 2

$$V \bullet V = V^2$$
$$V \bullet V \bullet V = V^3$$

$$V^+ = V \cup V^2 \cup V^3 \cup ...$$

V^+ ... set of all nonempty strings producible using V

adding the empty string:

$$V^* = \{e\} \cup V^+$$

V^* is the closure (set) of V

V^+ is the positive closure of V

(Obviously, strings may be infinite.)

Cardinality of V^* is often infinite

$$| V^* | = \infty$$

Grammars and Languages

Grammars give meaning to a subset of strings

$$L \subseteq V^*$$

L is *language*

Union, Concatenation, Iterates, Substrings

Union:

$$L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$$

Concatenation

$$L_1 \bullet L_2 = \{s \mid s=s_1s_2 \text{ where } s_1 \in L_1, s_2 \in L_2 \}$$

Iterate of L_1

$$L_1^{\text{iterate}} = \{s \mid s_1s_2...s_n, n \geq 0, s_i \in L_1\}$$

Substring

y is substring of x, $x,y \in V^*$ if $u,v \in V^*$ and

$$x = u\ y\ v$$

Grammars

Grammar G is a four-tuple

$$G = \{V_T, V_N, P, S\}$$

where

V_T set of terminal symbols (primitives)
the choice of V_T is art not science

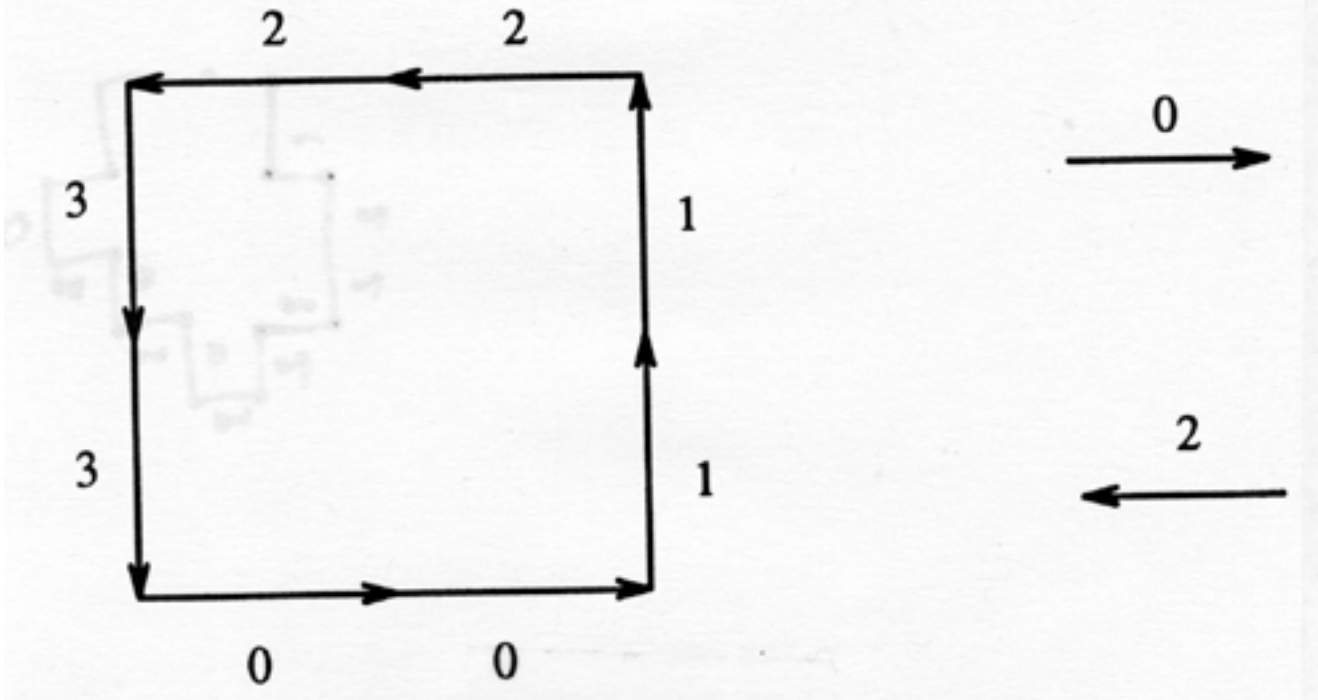
V_N set of nonterminal symbols (variables)
V_T, V_N are disjoint ... $V_T \cap V_N = \varnothing$

P set of rules (production rules, productions, rewriting rules)

S starting symbol (root), $S \in V_N$

$$
\begin{aligned}
P: \quad & (1) \quad s \rightarrow abcd \\
& (2) \quad aAbBcCdD \rightarrow a1Ab2Bc3Cd0D \\
& (3) \quad aAbBcCdD \rightarrow ABCD
\end{aligned}
$$

$$s \rightarrow^1 abcd \rightarrow^2 a1b2c3d0 \rightarrow^2 a11b22c33d00 \rightarrow^3 11223300$$



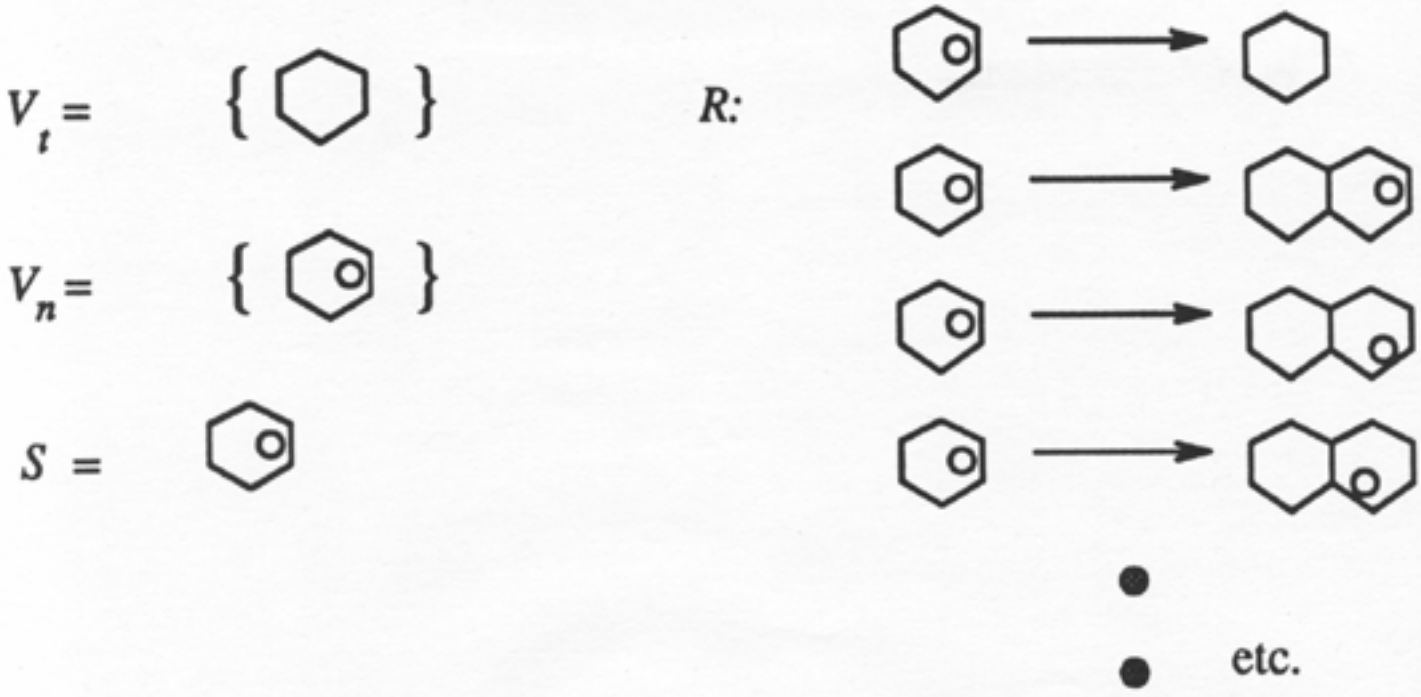


Figure 13.6 *Grammar generating hexagonal textures.*

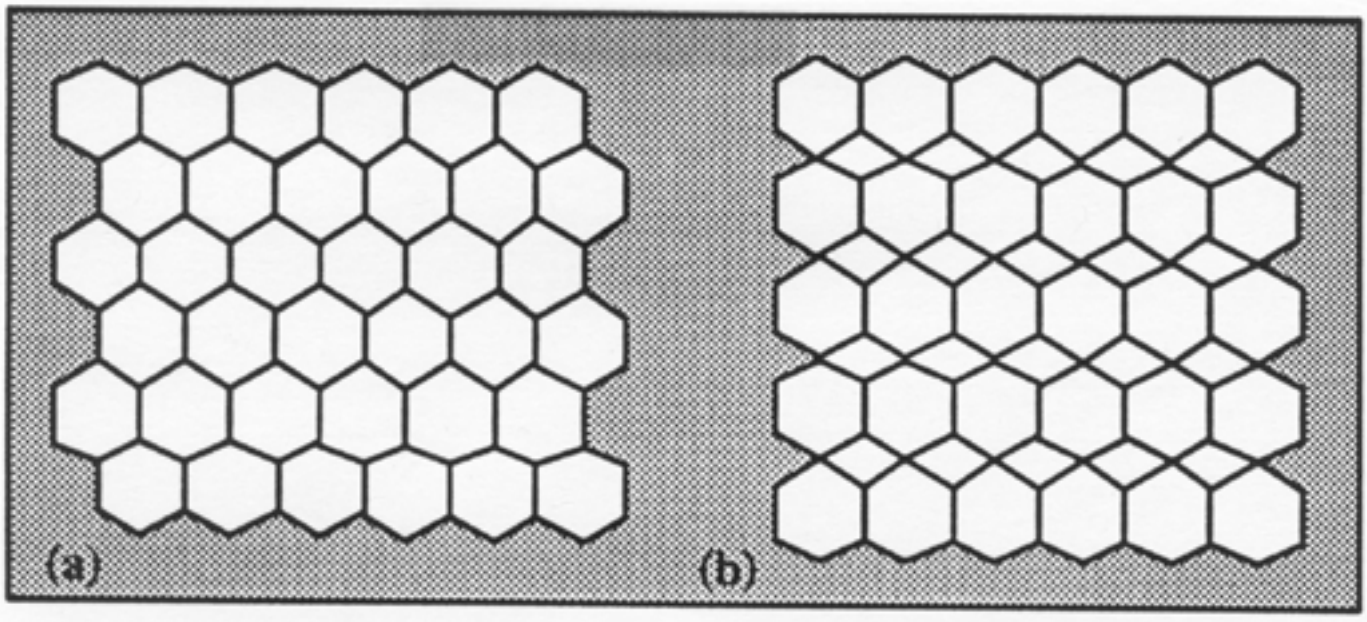


Figure 13.7 *Hexagonal textures: (a) Accepted, (b) rejected.*

Rule constrains

Rules may be constrained to the form:

$$A \rightarrow B$$

where

$$A \in (V_T \cup V_N)^+ - V_T^+$$

and

$$B \in (V_T \cup V_N)^*$$

A must consist of at least one member of V_N and B can consist of any combination of terminals and nonterminals

This is a partial definition of

Phrase Structure Grammar

Grammar Application Modes

1. Generative mode
   Grammar creates strings of terminal symbols using P

2. Analytic mode
   Given a string and specification of G, determine

whether

a) the string was generated by G

b) if yes, determine the structure of the string

Number of possible string patterns

Any subset $L \subseteq V_T^*$ is a language

If |L| is finite ... finite language

using entire V_T^*, language is infinite

To limit the number of possible strings, only the strings generated by some grammar will be considered

Language generated by grammar G ... L(G)

each string consists of terminals from V_T of G

each string was produced from S using P of G

Grammar Types and Production Rules

Notation:

nonterminals	upper case	S, T, ...
terminals	lower case	a,b, ...
length of string δ		$n= \|\delta\|$
Greek letters	strings	α, β, ...

Types of string grammars (Chomsky 1957)

General production rule

$\alpha \rightarrow \beta$ string α is replaced with string β

Type 0: T_0 (Free or Unrestricted Grammars)

no restrictions on the rewriting rules
little practical significance
erasing rules allowed ...
constraint $|\alpha| < |\beta|$ does not exist

Type 1: T_1 (Context-Sensitive Grammars), CSG

Restrictions:

$$\beta \neq e$$

and

$$|\alpha| <= |\beta|$$

Thus, the rules are restricted to the form:

$\alpha\alpha_i\beta \rightarrow \alpha\beta_i\beta$
α_i replaces β_i in the context of α, β,
α, β may equal e

Type 2: T_2 (Context-Free Grammars), CFG

Restrictions

$$\alpha = S \in V_N$$

single nonterminal

$$|S| <= |\beta|$$

Alternatively:

every rule must be of the form

$$S \rightarrow \beta$$

a nonterminal can be replaced with a string consisting of terminals and nonterminals

- CFG are most descriptively versatile grammars for which effective parsers are available

- Production rule restriction increased compared to T_1

Type 3: T_3 (Finite State or Regular Grammars) FSG

Restrictions:

same as for T_2 plus

at most one nonterminal symbol is allowed on each side of the rule

$$\alpha = S \in V_N$$

$$|S| <= |\beta|$$

and

$$A \rightarrow a \quad \text{OR} \quad A_1 \rightarrow aA_2$$

The above are **the only** allowed production rule forms,

a must be nonempty

<u>Graphical Representations of FSGs</u>

nodes ... nonterminals, node T is the terminal node

arc from A_i to A_j exists for each rule $A_i \rightarrow aA_j$

arc from A_i to T exists for each rule A_i → a

→ out-degree of T is 0, in-degree of S is 0

EXAMPLE:

FSG... $G = \{V_T, V_N, P, S\}$

$V_T = \{a, b\}$ $V_N = \{S, A_1, A_2\}$

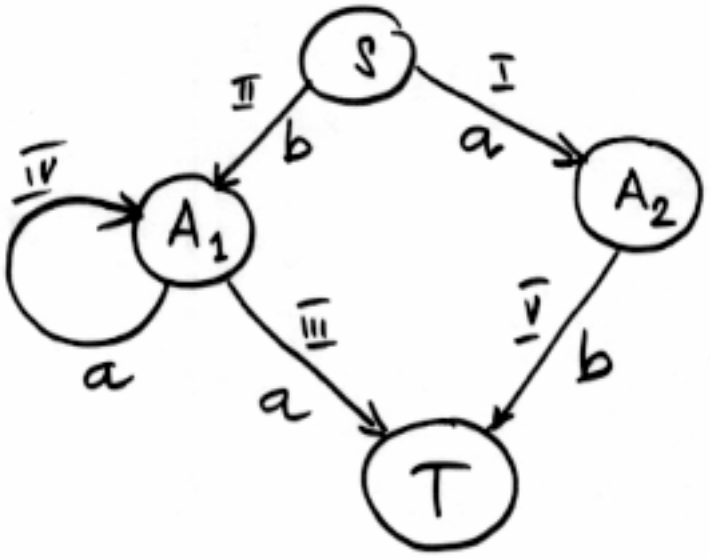
$P = \{$ $S \xrightarrow{I} a A_2$;

$S \xrightarrow{II} b A_1$;

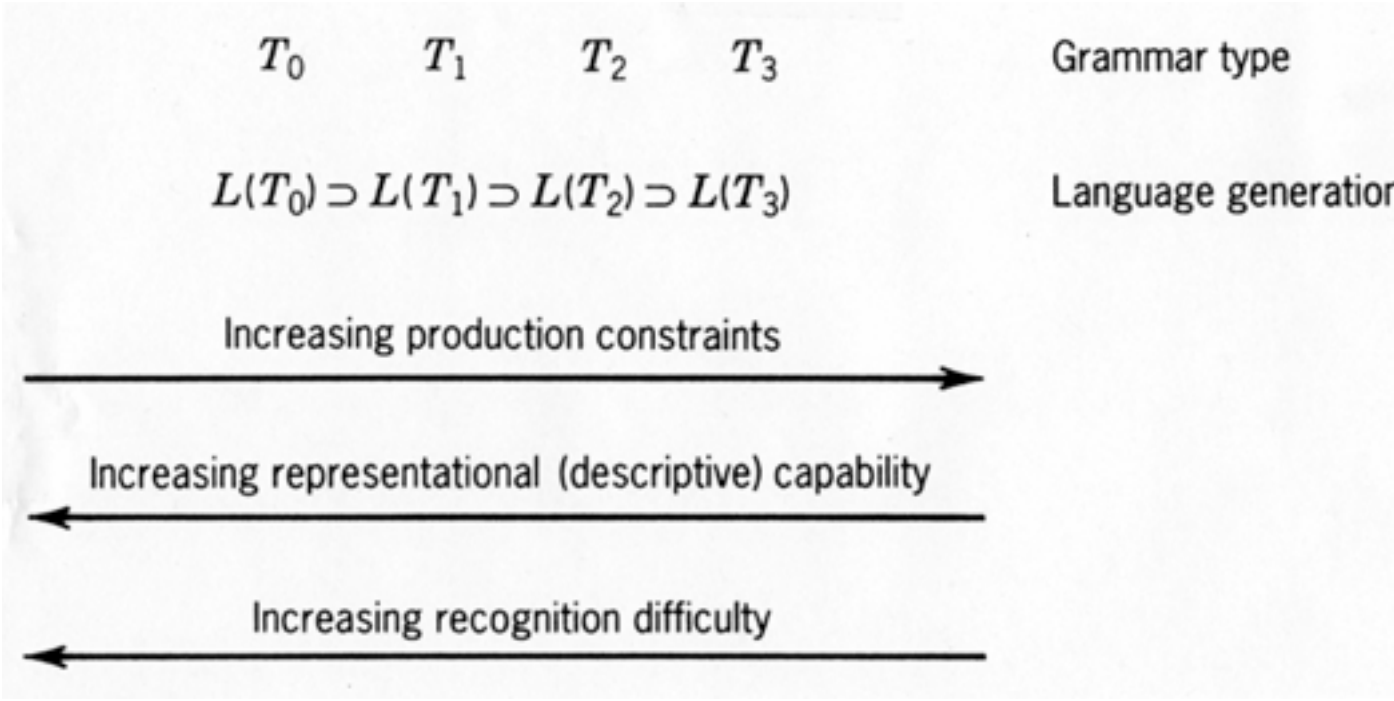
$A_1 \xrightarrow{III} a$

$A_1 \xrightarrow{IV} a A_1$

$A_2 \xrightarrow{V} b \}$



GRAPHICAL REPRESENTA OF G_{FSG}


T_0 T_1 T_2 T_3
Grammar type
$L(T_0) \supset L(T_1) \supset L(T_2) \supset L(T_3)$
Language generation
Increasing production constraints
Increasing representational (descriptive) capability
Increasing recognition difficulty

Derivations and Productions

Rewriting rules "→" allowable replacement, or production

Derivation ... conversion of one string to another "→"

Equivalence of Grammars

Two grammars G_1 and G_2 are equivalent
iff

$$L(G_1) = L(G_2)$$

Algorithms to determine equivalence exist for FSGs

A general algorithm to determine equivalence does not exist for CFGs

However,
two non-equivalent grammars may generate identical strings

Examples of RULES:

CFG:

$S \rightarrow aAa$

$A \rightarrow a$

$A \rightarrow b$

FSG:

$S \rightarrow a\ A_1$

$S \rightarrow b\ A_1$

$A_1 \rightarrow a$

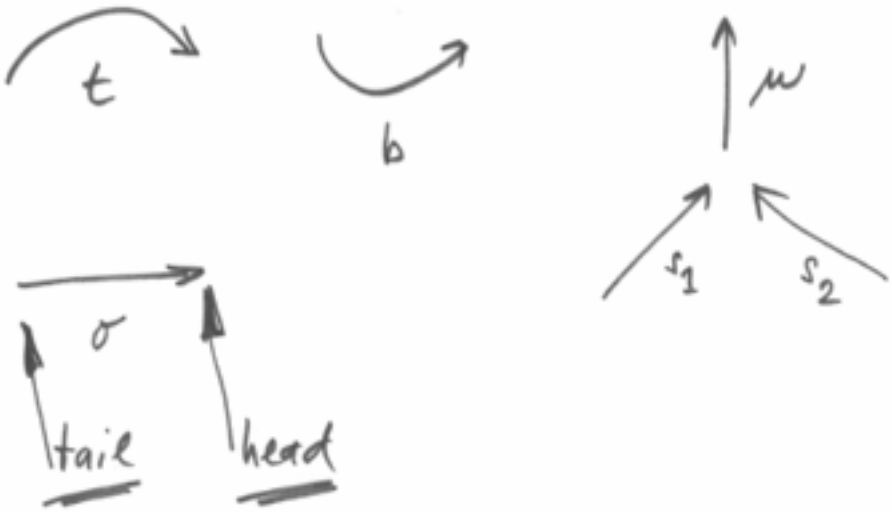
$A_1 \rightarrow b$

EXAMPLE

2D LINE DRAWING

$G_{cyl.} = \{ V_T, V_N, P, S \}$

$V_T = \{ t, b, u, o, s, *, \neg, + \}$



$V_N = \{ Top, Body, Cylinder \}$

$P = \{ Cylinder \longrightarrow Top * Body,$
$Top \longrightarrow t * b$
$Body \longrightarrow \neg u + b + u \}$

OPERATORS

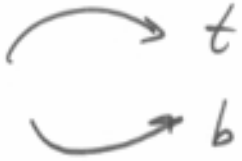
$+$ head-to-tail concatenation

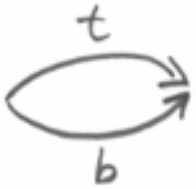
$*$ head-head AND tail-tail attachment

$\neg$ head and tail reversal

$\rightarrow$ may be replaced by ..

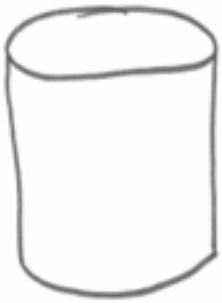
$t * b$





$s_1 + \neg s_2$

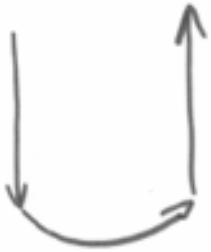
S = Cylinder



Cylinder → Top * Body

Top → t * b

Body → ¬μ + b + w



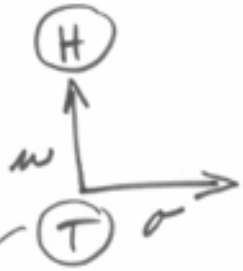
Top * Body

EXAMPLE

Character Description

PDL .. picture description language

$$A = w + \left(\left(w + \sigma + \neg w \right) * \sigma \right) + \neg w$$



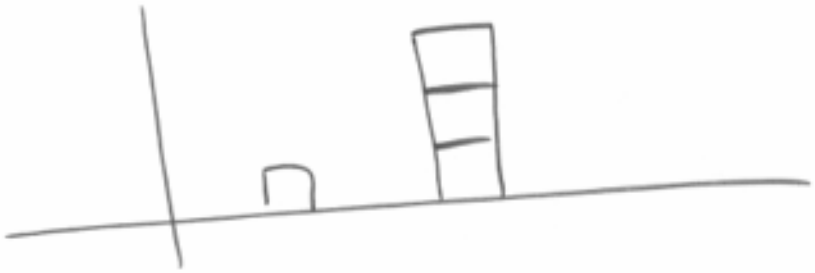
$$F = w + (\sigma * w) + \sigma$$

BLOCK WORLD DESCRIPTION

STACKS OF BLOCKS ON A TABLE

A) DISTINGUISH 2-BLOCK SITUATION FROM 3-BLOCK SITUATION

DIFFERENCE ... STRUCTURE



versus

There is no unique naming of blocks

1. There is always four blocks
2. The bottom of a stack must reside on the table

Let's develop grammars G_2 and G_3 to describe the 2-block and 3-block cases

Note the context-sensitive nature of G_3

G_2 (2-blocks). (Note: In V_T, + means 'also' and ↑ means 'on top of.')

$$V_T = \{table,\ a\_block,\ +,\ \uparrow\}$$

$$V_N = \{DESC,\ LEFT-STACK,\ RIGHT-STACK\}$$

$$S = DESC \in V_N$$

$$P = \{$$

$$DESC \rightarrow LEFT-STACK + RIGHT-STACK$$

$$DESC \rightarrow RIGHT-STACK + LEFT-STACK$$

$$LEFT-STACK \rightarrow a\_block \uparrow a\_block \uparrow table$$

$$RIGHT-STACK \rightarrow a\_block \uparrow a\_block \uparrow table\ \}$$

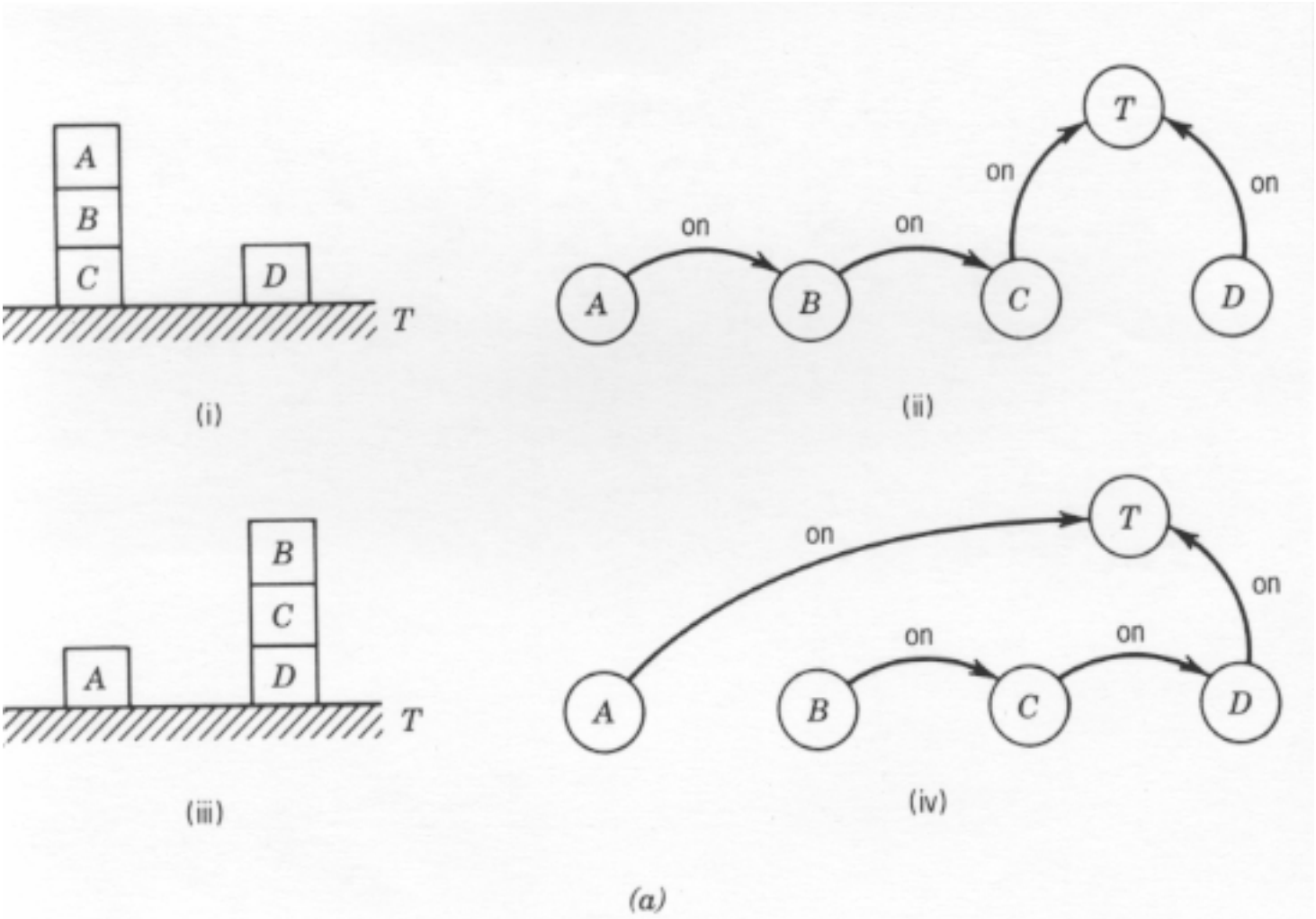
G_3 (3-blocks). V_T, V_N and S are the same as G_2.

$$P = \{$$

$$DESC \rightarrow LEFT-STACK + RIGHT-STACK$$

$$LEFT-STACK + RIGHT-STACK \rightarrow a\_block \uparrow table + a\_block \uparrow a\_block \uparrow a\_block \uparrow table$$

$$LEFT-STACK + RIGHT-STACK \rightarrow a\_block \uparrow a\_block \uparrow a\_block \uparrow table + a\_block \uparrow table\}$$



(a)

Figure 11: The 2-blocks versus 3-blocks descriptions.
(a) '3-blocks' class.
(i) Example
(ii) Graphical representation of (i)
(iii) Alternate example (same class)
(iv) Graphical representation corresponding to (iii).

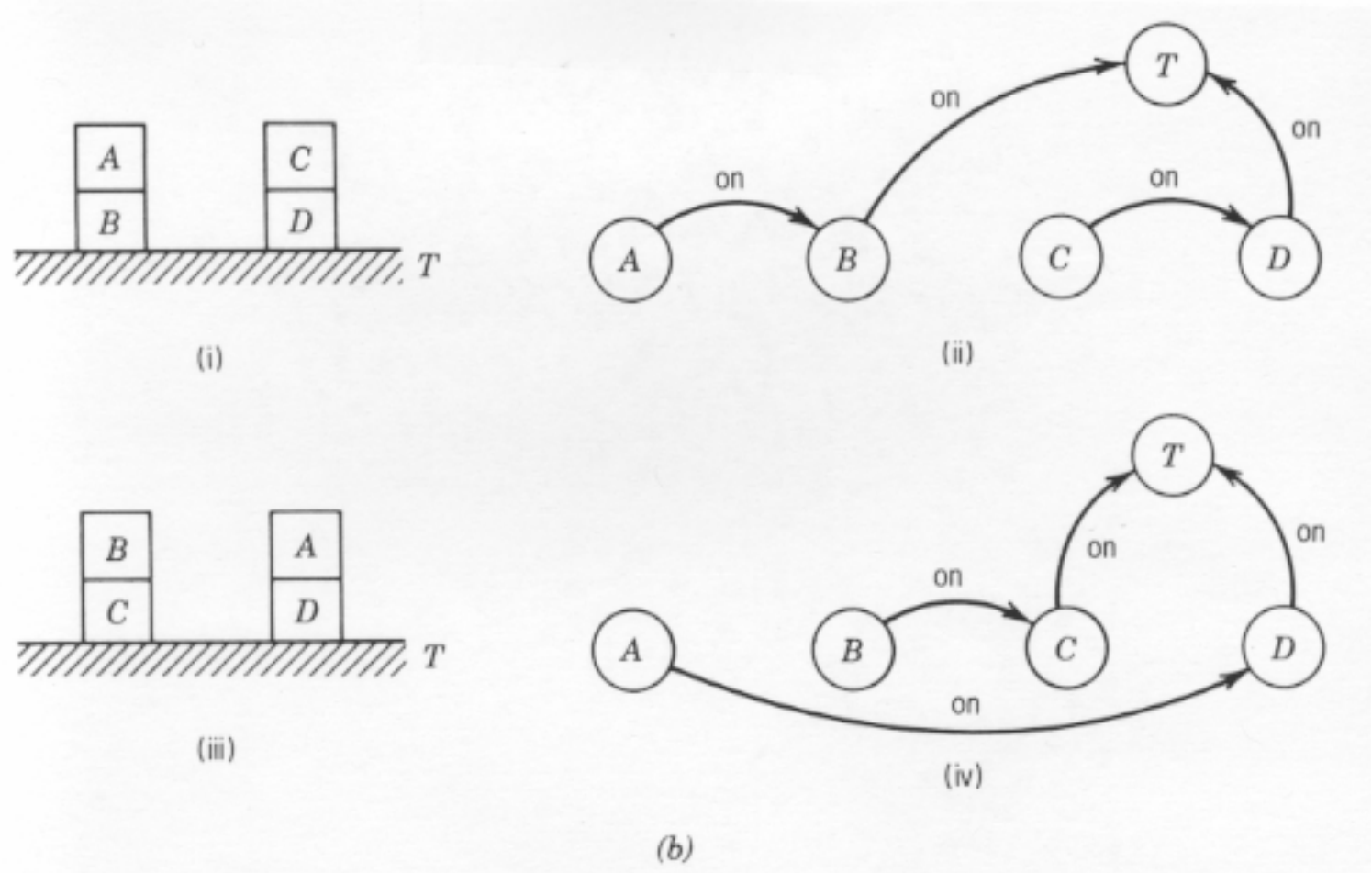


Figure 11 (cont.): **(b)** '2-blocks' class.
(i) Example
(ii) Corresponding graphical representation
(iii) Alternate example (same class)
(iv) Graphical description corresponding to (iii).

Syntactic Recognition via Parsing

We know how to generate syntactic description using formal grammars

Now, let's **inverse** the problem, assume that we have the syntactic description and the objective is to determine which $L(G_i)$ the string s belongs to ... which class it belongs to.

String Matching

For finite languages, it is possible to generate the entire language, compare individual strings with the string s.

Problems ... even if language is finite, it is usually large, the matching is inefficient

Metrics must account for similarity of primitives AND similarity in structure

Parsing

Is the pattern syntactically well formed in the context of one or more prespecified grammars.

Parser = syntactic analyzer

Parsers are usually associated with grammar types.
The more restrictive the grammar type, the simpler parser can be used.

Special case of Context-Free Grammar ...
Chomsky normal form CNF

A CFG is in CNF if each element of P is in one of the following forms

$A \rightarrow BC$ where $A,B,C \in V_N$

$A \rightarrow a$ where $A \in V_N$, $a \in V_T$

Lemma:

For any CFG, there exists an equivalent CNF.

Parsing

- Parser may have hierarchical structure to be more efficient

- Decomposition in subparts

The Derivation Tree

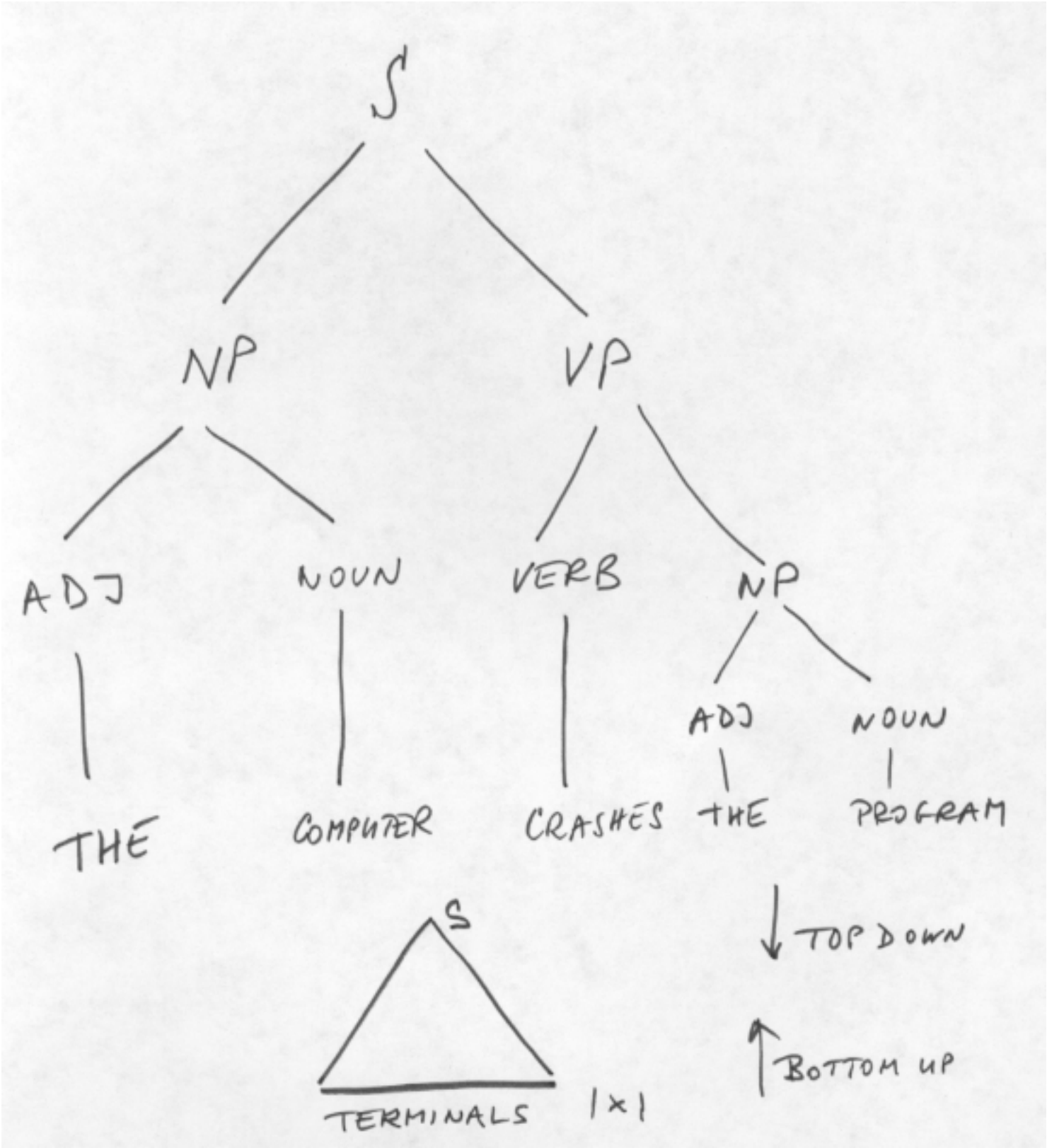
Example:
$G_1 = \{V_T, V_N, P, S\}$

V_T = {the, program, crashes, computer}
V_N = {SENTENCE,ADJ,NP,VP,NOUN,VERB}
P = {
SENTENCE → NP + VP,
NP → ADJ + NOUN
VP → VERB + NP
NOUN → computer|program
VERB → crashes
ADJ → the }
S = SENTENCE

Generation using grammars:

A. the program crashes the computer
B. the program crashes the program
C. the computer crashes the program
D. the computer crashes the computer

Case C … Derivation Tree



Parsing Problem - an abstract view:
grammar:

$G = \{V_T, V_N, P, S\}$

Filling the interior of the derivation tree triangle.

If successful, we determined that $x \in L(G)$.

Filling from the top ... Top-down approach
from the bottom ... Bottom-up approach

Bottom up parsing from the terminals toward S

Top down parsing from S toward the terminals

<u>Parsing/Generation similarities</u>
Generative mode is substantially easier

Parser - must determine the extent of nonterminals
- must find use for all elements (all must be used)

Parsing complexity
often a high-complexity problem
using a priori information helps tremendously

The Decidability Problem

Given L(G) and string x, the question is:

$$x \overset{?}{\in} L(G)$$

If this question can be answered in finite time, the parsing problem is fully decidable.

Comparing Parsing Approaches

- difficult

- for some grammars, top-down is better
- for other grammars, bottom-up is better.

- transformation or normalization of grammar may affect parsing efficiency

Brute force approaches (top down and bottom up) have often exponential complexity - exponentially grows with $|x|$.

CYK ... Cocke Younger Kasami Parsing Algorithm

parsing in $|x|^3$ steps

working with context-free grammars CFG
in Chomsky normal form CNF

CNF:
productions either $A \rightarrow BC$
or $A \rightarrow a$

Thus, derivation of any string ... series of binary decisions

Grammar:

S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b

Construction of a CYK parse table:
start from location (1,1), if a substring of x, beginning with x_i and of length j can be derived from a nonterminal, this nonterminal is placed in (i,j).

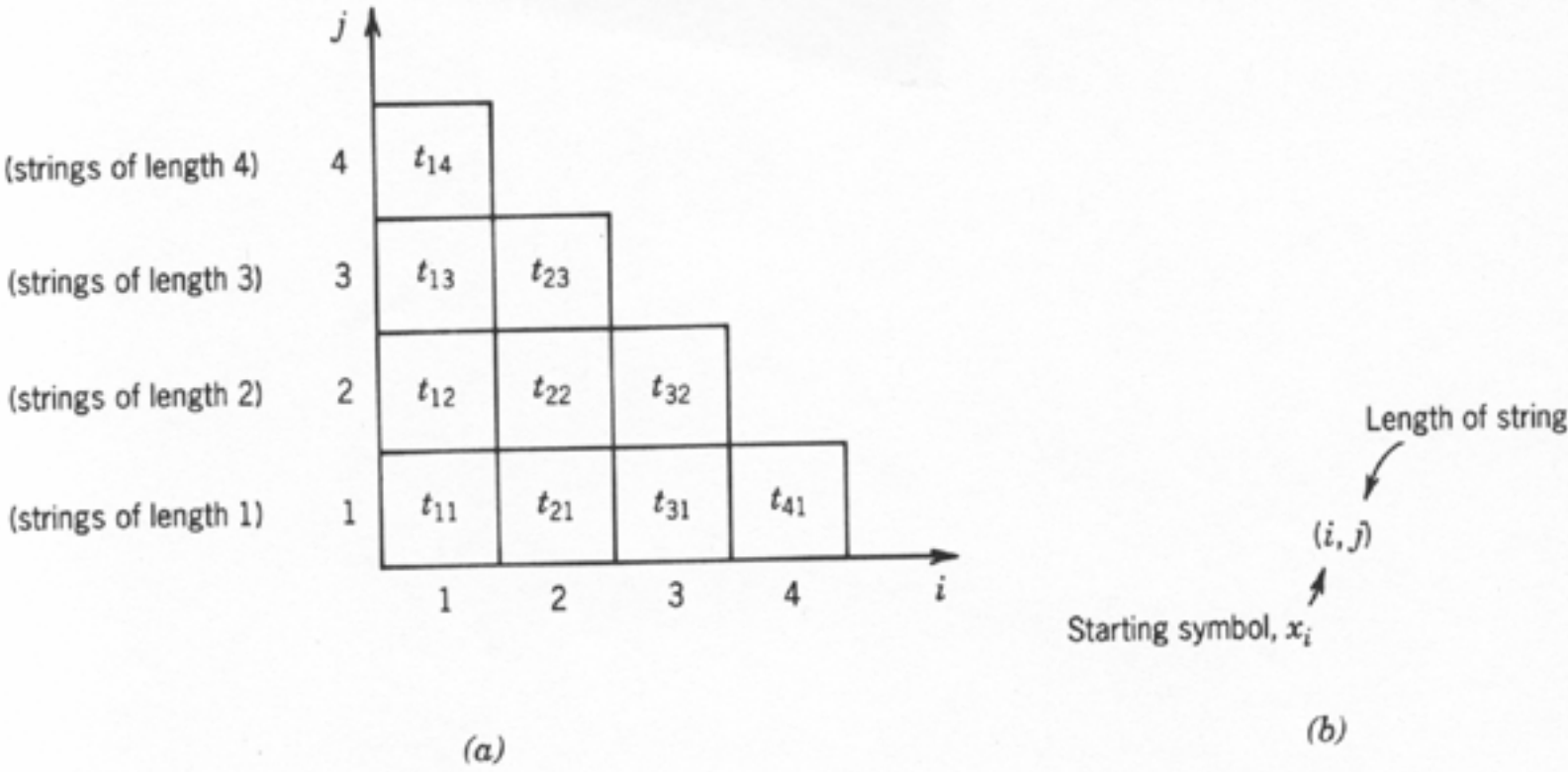
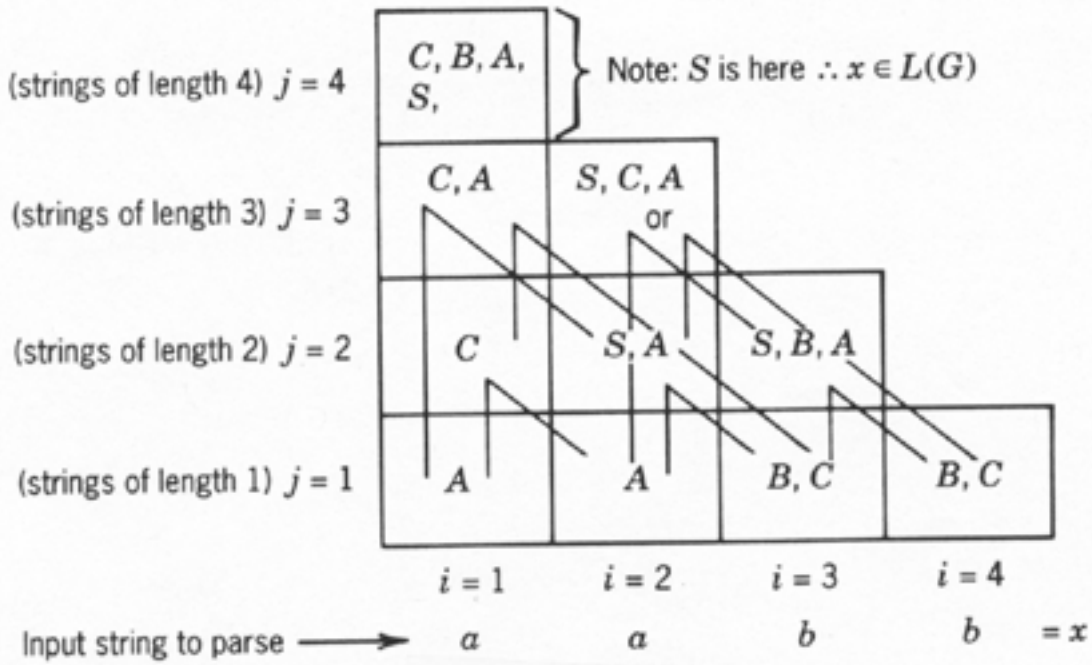


Figure 3: CYK parse table.
(a) Example of table for $|x| = n = 4$.
(b) Structure of cell entries.



Augmented Transition Networks (ATNs) in Parsing

Transition network (TN) is a digraph (directed graph) showing context free production rules of a grammar

Easily maps into finite state machines

TN: set of nodes ... states
set of labeled arcs ... nonterminals or terminals

TN parses an input string by starting with an initial state (S) and checking for allowed transitions until the goal is reached = until a successful parse is found.

Goal states are labeled END or double circled.

Parsing is done by consuming the input string

An arc may be traveled under one of the following conditions:
1) the arc is labeled with a terminal node and the next entry in the input string is the same terminal.
This terminal will be consumed

2) The arc is labeled with a nonterminal. In this case, control passes to one or more TNs related to this nonterminal.

If the parser reaches a state (node) where no outgoing arc is applicable, a failure is encountered.

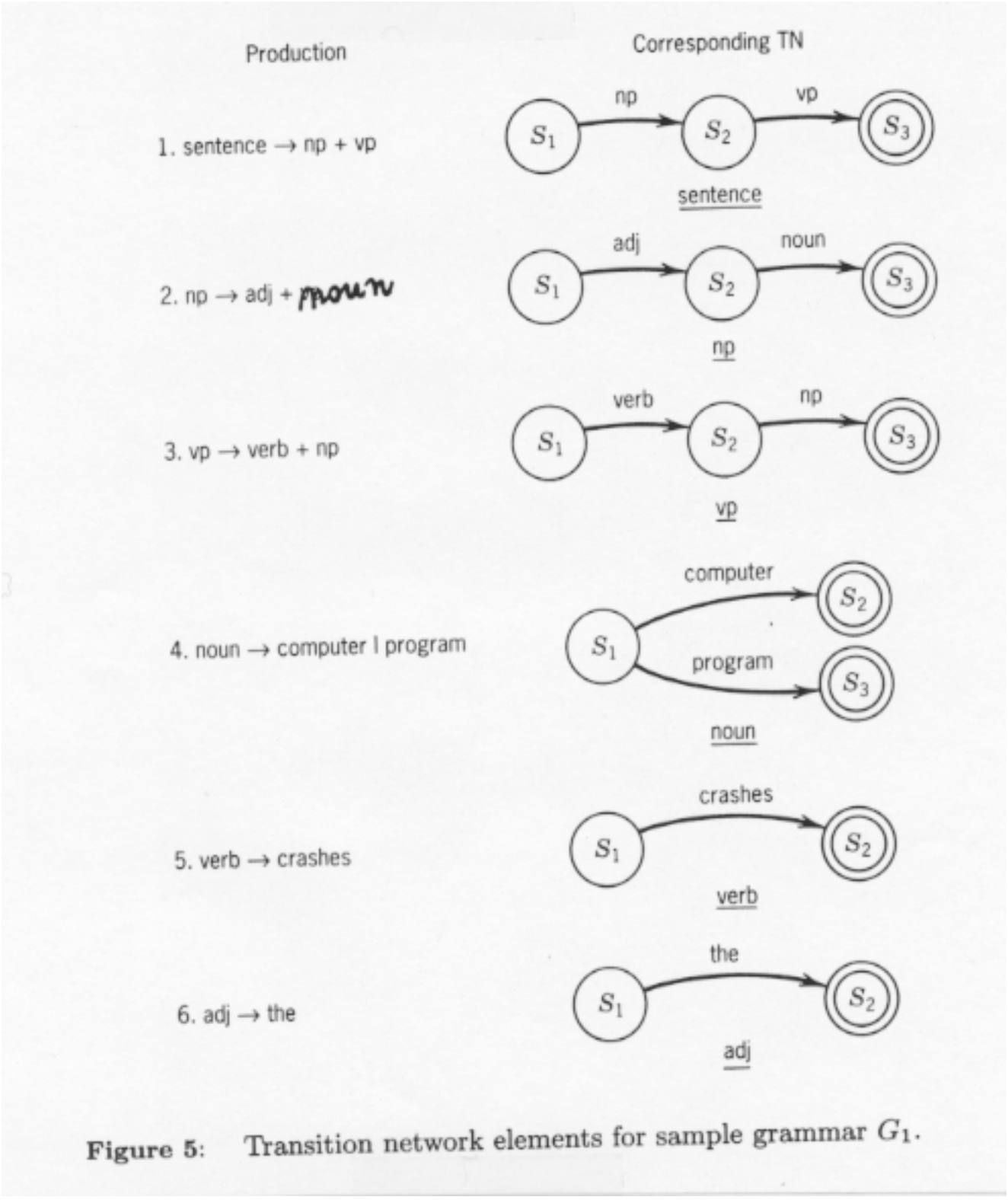


Figure 5: Transition network elements for sample grammar G_1.

Augmented TNs ... ATNs

adding several features, especially *recursion*

- conditional tests,
- corresponding actions (jump to another ATN under certain condition, etc.)

Higher Dimensional Grammars

facilitate relational descriptions

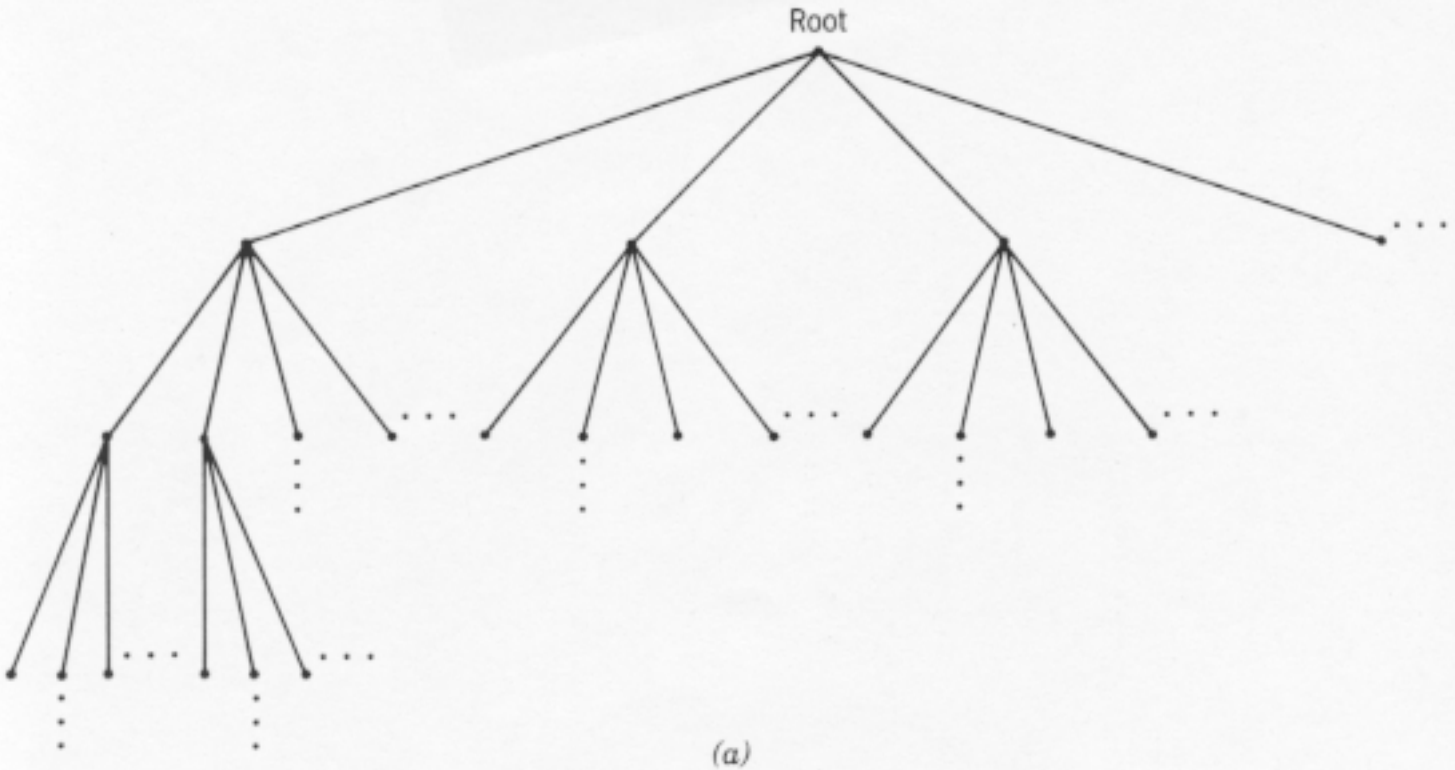
rewriting rules are more complex

Popular:
- tree grammars
- web grammars

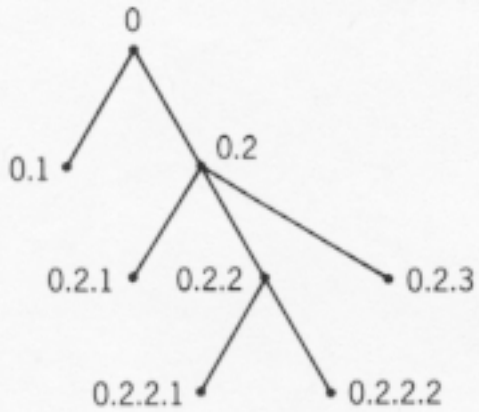
Note:
no correlation between dimensionality of the problem and dimensionality of the grammar

Tree Grammars

useful for hierarchical decompositions



(a)



(i)

T = { 0, 0.1, 0.2, 0.2.1, 0.2.2, 0.2.3, 0.2.2.1, 0.2.2.2 }

(ii)

(b)

Figure 6: Tree characterizations.
(a) General tree structure (unlabeled notes).
(b) Sample tree.
(i) Labeled tree, T.
(ii) Description using alphabet V.

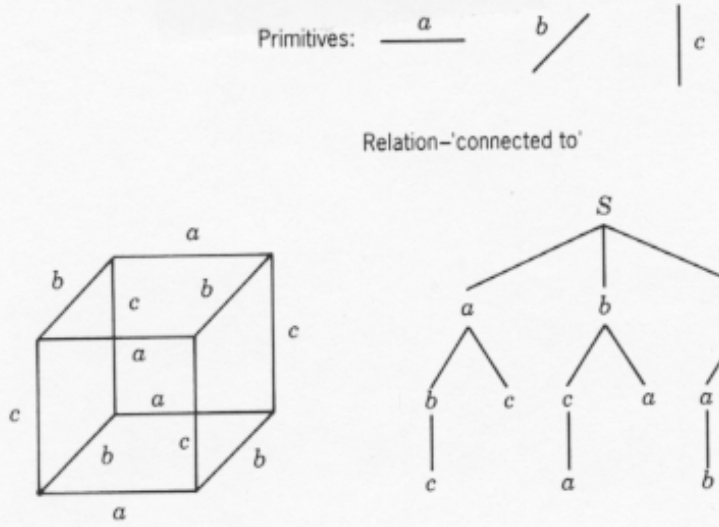


Figure 7: Tree representation of a cube.

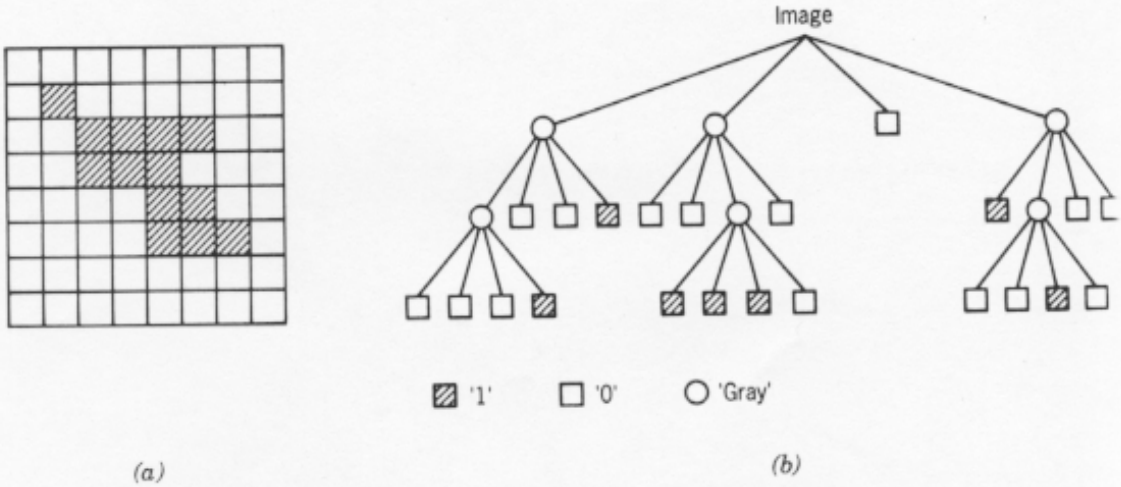


Figure 8: Quadtree representation of binary image.
(a) 8×8 image.
(b) Resulting quadtree.

Traversing a Tree

Depth-first
Breadth-first

Tree Similarity

Similarity measure ... for 2 trees T_1, T_2,
similarity is denoted $d(T_1,T_2)$ using string descriptions of the corresponding trees

Tree grammars

conceptually identical to chain grammars, more complex rules due to more freedom in replacements

Stochastic Grammars

Formal grammars assumed that languages generated by two grammars were disjoint, however it is rarely the case.

It was not presented how to incorporate a priori information about likelihood of classes

Stochastic grammar is a four-tuple

$$G_s = \{V_T, V_N, P_s, S_s\}$$

production rules are of the form:

$$a_i \xrightarrow{p_{ij}} b_j$$

where p_{ij} is a probability that a_i is replaced with b_j

Thus, several rules with the same left side can be present in the stochastic grammar

Sum of all probabilities for such rules = 1

If $\neq 1$, the grammar is called fuzzy.

Learning via Grammatical Inference

So far, we assumed that grammars were defined.

If grammars are defined by the designer of the SYNTPR system, then no training is needed.

More realistic situation ... grammars are not known.

Learning process of inferring grammars from a training set of examples ... **grammatical inference (GI).**

GI is a supervised learning approach.

Syntactic Learning

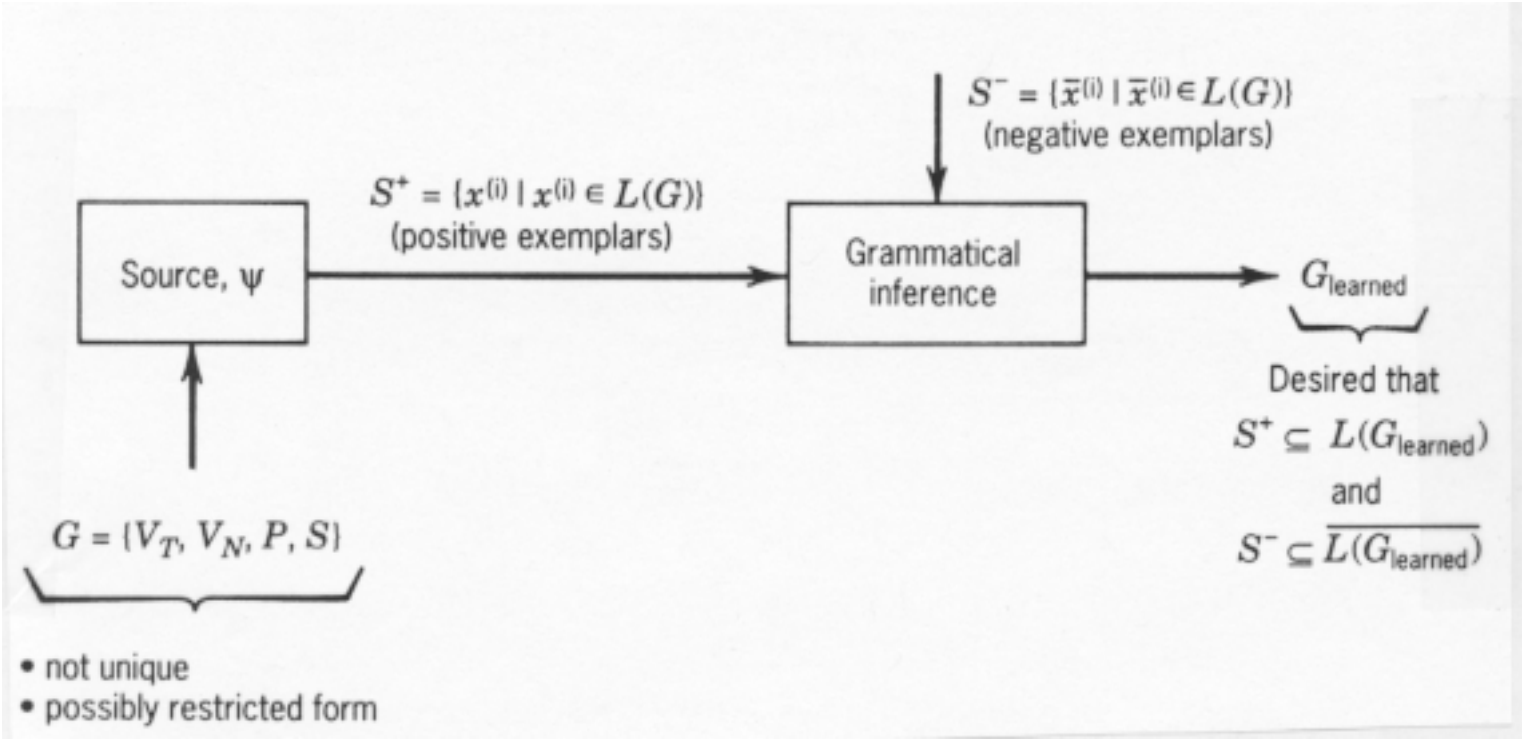
!!! No unique relationship between a given language and some grammar.

Thus, the same language may be generated by several different grammars

... which of the several grammars will be learned as a result of GI?

Additional constraints are applied to the learned grammar.

How to characterize the grammar source ψ ?



Training set

Goal: use training set H to learn the grammar G_{learn} which is "close" to the grammar G we look for.

Positive examples S^+
Negative examples S^-

Training Set $H = \{S^+, S^-\}$

$S^+ = \{x^{(i)} \mid x^{(i)} \in L(G)\}$

$S^- = \{\neg x^{(i)} \mid \neg x^{(i)} \notin L(G)\}$

In other words, goal is to

- develop a grammar G_{learn} that can generate S^+, but not S^-

Even better ...

- develop a grammar that in addition can represent properties of the training set
  - **inductive character** of learning

Example:

$S^+ = \{ab, aabb, aaabbb, aaaabbbb\}$

it would be nice to expect that aaaaabbbbb will also belong to the language

$\Rightarrow$ include production rules

$$A \rightarrow ab, \quad A \rightarrow aAb$$

Cardinality of S^+

L(G) is usually infinite but S^+ is always finite

$$S^+ \subseteq L(G)$$

$$|S^+| << |L(G)|$$

Also

$$S^- \subseteq \neg L(G)$$

$$|S^-| << |\neg L(G)|$$

Since a finite sample does not uniquely define a language, such finite sample may be associated with an infinite number of languages.

$\Rightarrow$ **Inferring a unique grammar is impossible.**

Quality of the training set

S^+ must be structurally complete ... all production rules of the grammar must be reflected in S^+.

How to guarantee this ???

It is known that using only S^+ makes grammar inference undecidable and it is true even for regular grammars.

Even if S^- is used, problem is still NP hard.

Heuristics must be used to ensure computational feasibility.

GI objectives

1)
Specify the class of grammar, or request that the inferred grammar is of minimal complexity

2)
Create G_{learn} such that it generates all strings from S^+ and no strings from S^-. Require (this is difficult) that G_{learn} also generates strings similar to those from S^+ (and not from S^-).

3) Require that the inference algorithm is of reasonable computational complexity = requirement of usage of heuristics.

Intuitive GI procedure

S^+, S^- given

$G^{(0)}$ ... the initial guess about grammar G

$$G^{(0)} = \{V_T^{(0)}, V_N^{(0)}, P^{(0)}, S^{(0)}\}$$

Procedure

1) set k = 0

2) choose one element $x^{(i)}$ from S^+, using $G^{(k)}$, parse $x^{(i)}$, if parse is successful, continue, otherwise modify $G^{(k)}$.

3) choose one element $\neg x^{(i)}$ from S^-, using $G^{(k)}$, parse $x^{(i)}$, if parse is **not** successful, continue, otherwise modify $G^{(k)}$.

4) If all elements from H were successfully parsed, stop. Otherwise, increment k and continue with step (2).

Problems:

A) How to modify $G^{(k)}$?
B) modifications in each step lead to combinatorial explosion.

Grammar inference - more realistic approach

Let's restrict the grammar to **finite state** (regular) ... single strings.

Therefore, rules are only of type:

$$A \rightarrow a \quad \textit{and} \quad A \rightarrow aB$$

Example:

$x^{(i)} \in S^{+}$, $x^{(i)}$ = caaab

We can derive the first guess of V_T and V_N

$$V_T = \{ a,b,c\}$$

and

$$V_N = \{ S, A, B\}$$

Production rules derived:

$S \rightarrow cA$	1
$A \rightarrow aB$	2
$B \rightarrow aC$	3
$C \rightarrow aD$	4
$D \rightarrow b$	5

Obviously, the fact that we have a new non-terminal for each rule will result in an excessive number of rules for large sample sizes and/or long strings.

However, rules 2,3,4 are quite similar and can be combined to reduce redundancy.

Is this set of rules identical?

$S \rightarrow cA$	1
$A \rightarrow aA$	2
$A \rightarrow b$	3

General procedure

1) for all $x^{(i)} \in S^+$, determine the set of distinct terminals, construct V_T.

2) for each $x^{(i)} \in S^+$, define the corresponding set of productions by considering the string from left to right, construct V_N and P.

This approach (same as in Example above) yields a language $L(G_c) = S^+$.

However, redundancies exist.

Also, $L(G_c)$ is finite.

3) merge production rules to produce a recursive grammar and a corresponding infinite language.

Example:

S^+ = {bbaab, caab, bbab, cab, bbb, cb}

… V_T = { a,b,c }

left to right inferring of rules:

$S \rightarrow bA$

$A \rightarrow bB$

$B \rightarrow b$

$B \rightarrow aC$

$S \rightarrow cD$

$C \rightarrow b$

$C \rightarrow aE$

$E \rightarrow b$

$D \rightarrow b$

$D \rightarrow aF$

$F \rightarrow aG$

$F \rightarrow b$

$G \rightarrow b$

rules are similar B,C,D,E,F,G same rule
and
rules are similar B,C,D,F ... same nonterminal

$\Rightarrow$ $S \rightarrow bA_1 \mid cA_2$
$A_1 \rightarrow bA_2$
$A_2 \rightarrow aA_2 \mid b$

Graphical approaches to SYNTPR

Graph matching - especially for higher-dimensional graphs - replaces parsers

Graph similarity assessment becomes important

Digraphs $\rightarrow$ Semantic Nets $\rightarrow$ Relational graphs

Graph: $G = \{N,R\}$

N ... set of nodes
R ... set of arcs $R \in N \times N$

Semantic net each node has a label

Relational graph arcs represent relations
= relations have a label

Graphs and Pattern Recognition

- each pattern is represented by a graph and graphs are compared

- as usually, reality is more difficult than this straightforward concept - match is rarely absolute in complex graphs, and even if it is, computational expense is high

Comparing Relational Graph Descriptions

Scenario 1 (conservative):
any feature or relation not present in both graphs results in a match failure

Scenario 2 (optimistic):
any single match of feature or relation yields success

Scenario 3 (realistic):
somewhere in between

Why bother with graph matching - we have developed grammatical approach, parsers, grammatical inference, etc.?

Graph matching is advantageous

- when the training set is too small to correctly infer the grammar

- when each pattern can be considered a class prototype

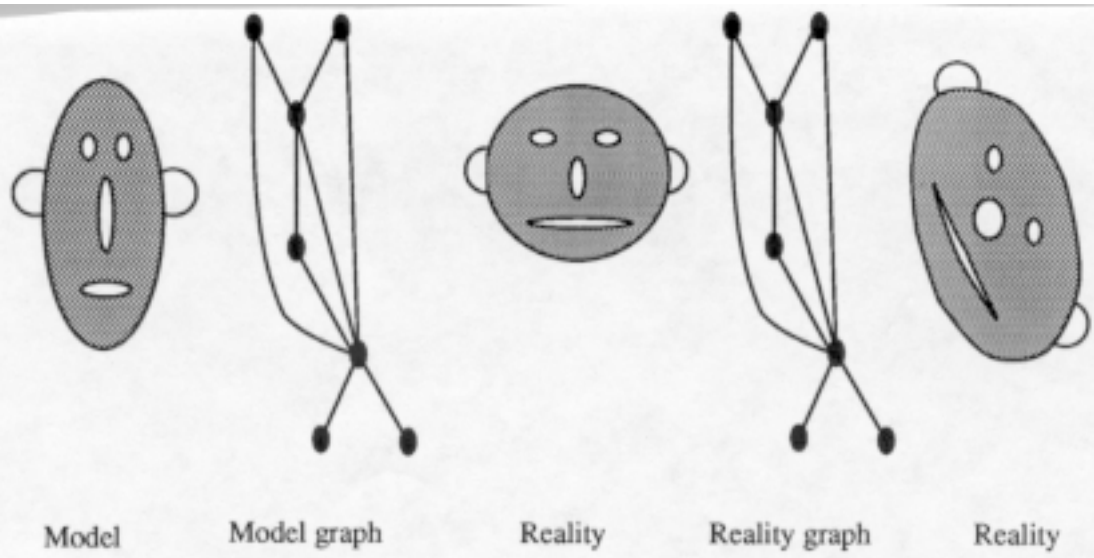


Figure 3.20: Graph matching problem.

3.5.1 Isomorphism of graphs and subgraphs

1. **Graph isomorphism**. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, find a 1:1 and *onto* mapping (an isomorphism) f between V_1 and V_2 such that for each edge of E_1 connecting any pair of nodes $v, v' \in V_1$, there is an edge of E_2 connecting $f(v)$ and $f(v')$; further, if $f(v)$ and $f(v')$ are connected by an edge in G_2, v and v' are connected in G_1.

2. **Subgraph isomorphism**. Find an isomorphism between a graph G_1 and subgraphs of another graph G_2.

   This problem is more difficult than the previous one.

3. **Double subgraph isomorphism**. Find all isomorphisms between subgraphs of a graph G_1 and subgraphs of another graph G_2.

Node properties invariant under graph isomorphism

Node partitioning:

- node attributes (evaluations)
- the number of adjacent nodes (connectivity)
- the number of edges of a node (node degree)
- types of edges of a node
- the number of edges leading from a node back to itself (node order)
- the attributes of adjacent nodes
- etc.

Algorithm 39: Graph isomorphism

1. Take two graphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$.
2. Use a node property criterion to generate subsets V_{1i}, V_{2i} of the node sets V_1 and V_2. Test whether the cardinality conditions hold for corresponding subsets. If not, the isomorphism is disproved.
3. Partition the subsets V_{1i}, V_{2i} into subsets W_{1j}, W_{2j} satisfying conditions given in equation (3.35) (no two subsets W_{1j} or W_{2j} contain the same node). Test whether the cardinality conditions hold for all the corresponding subsets W_{1j}, W_{2j}. If not, the isomorphism is disproved.
4. Repeat steps (2) and (3) using another node property criterion in all subsets W_{1j}, W_{2j} generated so far. Stop if one of the three above mentioned situations occurs.
5. Based on the situation that stopped the repetition process, the isomorphism either was proven, disproven, or some additional procedures (like backtracking) must be applied to complete the proof or disproof.

$$v_{2i} \in \bigcap_{j|v_{1i} \in V_{1j}} V_{2j} \tag{3.34}$$

$$\bigcap_n W_{1i} = \emptyset \quad \text{and} \quad \bigcap_n W_{2i} = \emptyset \tag{3.35}$$

$$v_{2i} \in \{ \bigcap_{\{j|v_{1i} \in W_{1j}\}} W_{2j} \} \cap \{ \bigcap_{\{k|v_{1i} \notin W_{1k} \wedge W_{1k} = W_{2k}\}} W_{2k}^C \} \tag{3.36}$$

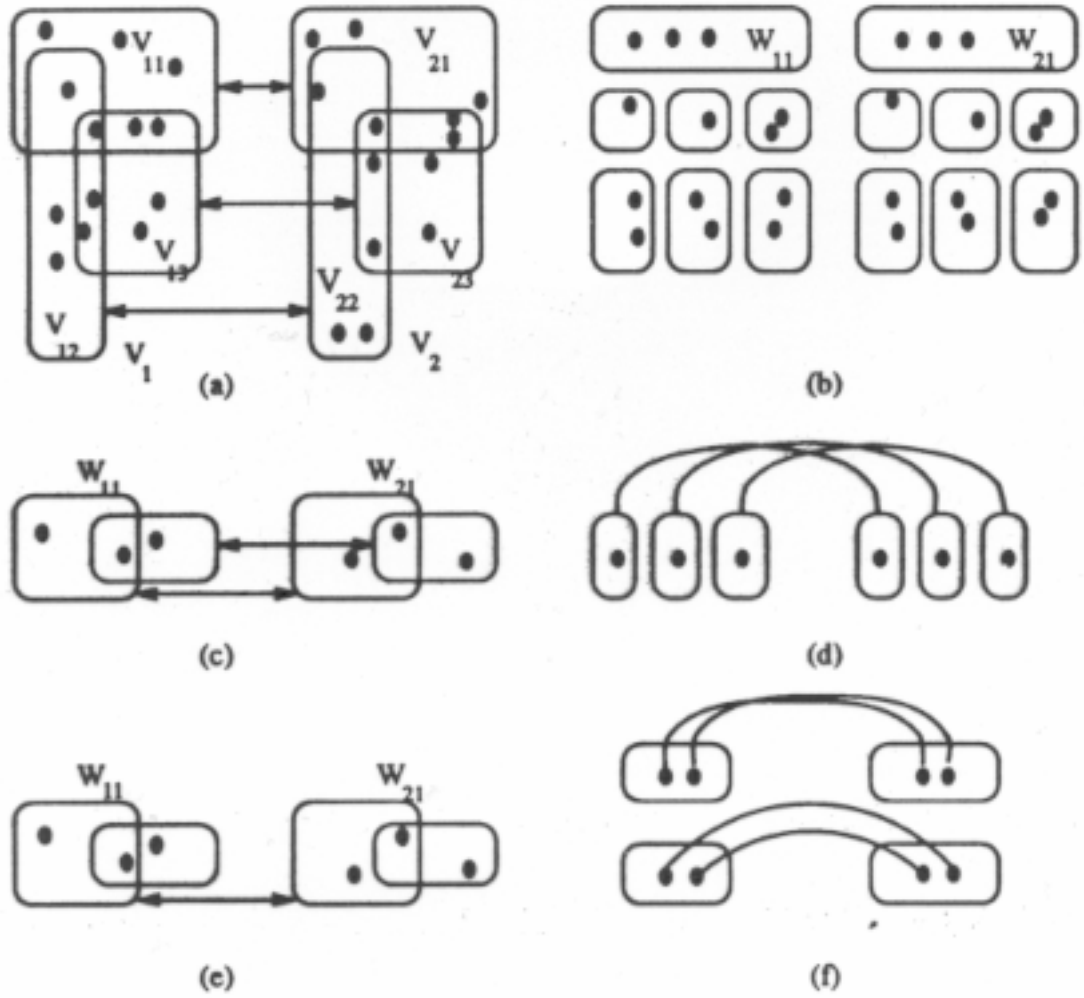


Figure 3.21: Graph isomorphism: (a) Testing cardinality in corresponding subsets, (b) partitioning node subsets, (c) generating new subsets, (e) graph isomorphism disproof, (f) situation when arbitrary search is necessary.

Matching Measures that Allow Structural Deformations

- we need a “distance” measure reflecting graph similarity

1) Extraction of features from G_1 and G_2 forming two feature vectors; followed by StatPR recognition using the feature vectors.

2) metric ... the minimum number of transformations necessary to transform G_1 to G_2

transformations:

- node insertion
- node deletion
- node splitting
- node merging
- etc.

difficulties:

- computational complexity
- difficult to design an adequate distance measure to assess different graphs from the same class as similar and from different classes as dissimilar

Double Subgraph Isomorphism

Can be converted into a subgraph isomorphism using the assignment graph

A pair v_1, v_2 is called an assignment if the nodes v_1, v_2 have the same node property descriptions, and two assignments v_1, v_2 and v'_1, v'_2 are compatible if (in addition) all relations between v_1 and v'_1, also hold for v_2 and v'_2 (graph arcs between v_1 and v'_1 and v_2 and v'_2 must have the same evaluation, including the no-edge case).

Algorithm 40: Maximal clique location

1. Take an arbitrary node $v_j \in V$; construct a subset V_{cl}
2. In the set V^C_{clique} search for a node v_k that is connect in V_{clique}. Add the node v_k to a set V_{clique}.
3. Repeat step (2) as long as new nodes v_k can be found.
4. If no new node v_k can be found, V_{clique} represents the node set of the maximal clique subgraph G_{clique} (the maximal clique that contains the node v_j).

Template and Springs Principle

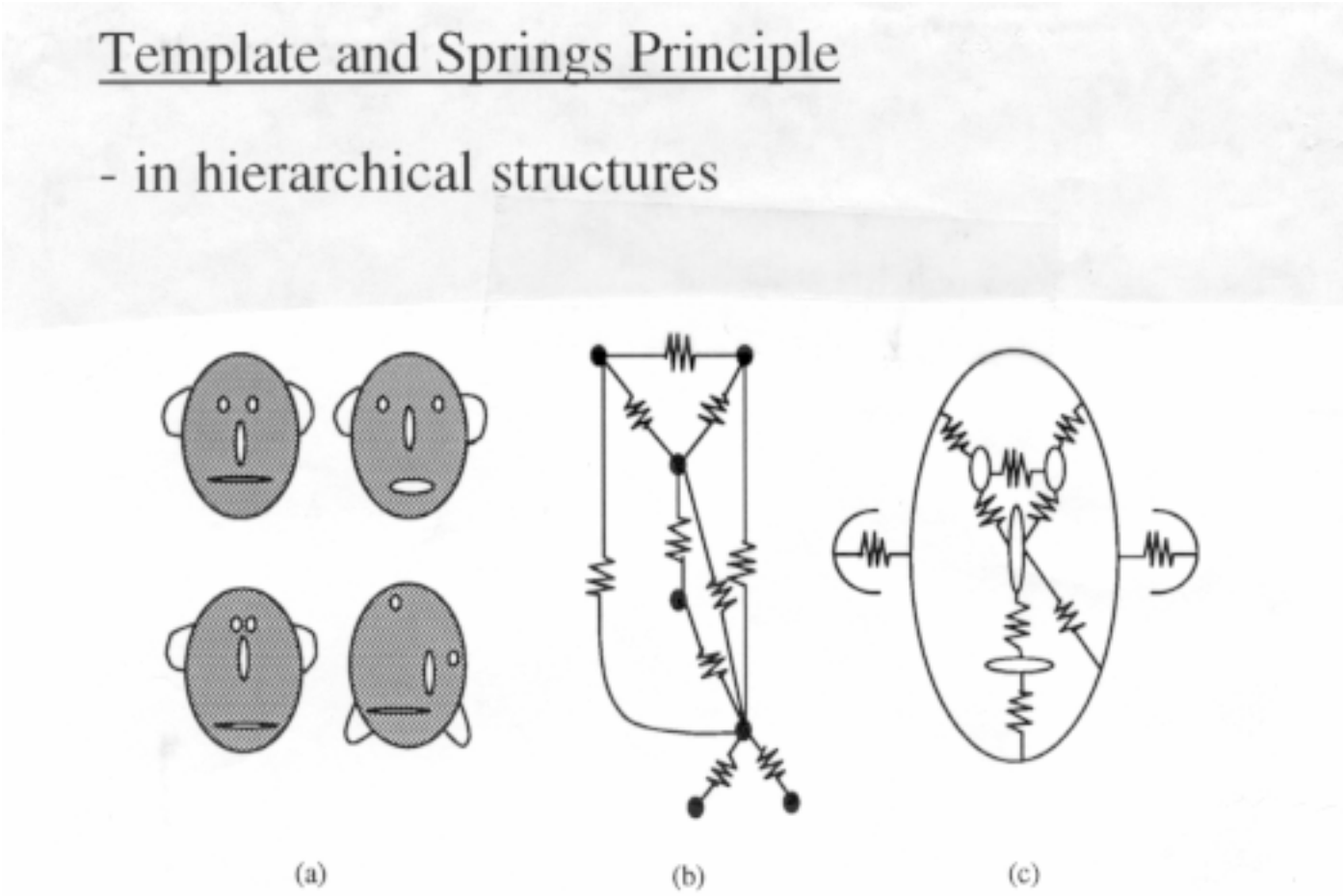


Figure 3.22: Templates and springs principle: (a) Different objects having the same description graphs (b),(c) nodes (templates) connected by springs, graph nodes may represent other graphs in finer resolution.