# Parsing with Tree-Adjoining Grammars

Anoop Sarkar
Simon Fraser University
*anoop@cs.sfu.ca*

ACL/HCSNet Advanced Program in
Natural Language Processing
Melbourne, Australia July 13-14, 2006



# High Level Overview

- 1st session
  - Specific motivations for Tree-Adjoining Grammars (TAGs)
  - Lexicalized TAG: definition, examples and extensions
  - Lexicalized TAGs from TreeBanks
  - Synchronous TAG

# High Level Overview

- 2nd session
  - Statistical parsing: generative models for TAG
  - TAG-based shallow parsing: SuperTagging and Stapling
  - Bootstrapping between CFG and LTAG parsers



# Preliminaries

# Formal Languages and NLP

| Formal Language Theory | NLP |
|---|---|
| Language (possibly infinite) | Text Data, Corpus (finite) |
| Grammar | Grammar (usually inferred from data, produces infinite set) |
| Automata | Recognition/Generation algorithms |



# Sentences as Strings

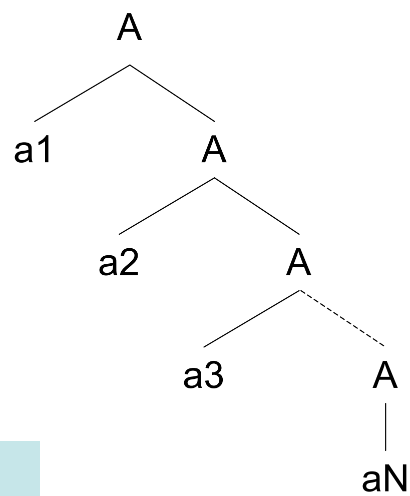
| | | | | |
|---|---|---|---|---|
| David | likes | peanuts | | |
| Noun | Verb | Noun | | |
| David | said | that | Mary | left |
| Noun | Verb | Comp | Noun | Verb |

- Linear order: all important information is contained in the precedence information, e.g. useful “feature functions” are w-2, w-1, t-2, t-1, w0, w+1, w+2, t+2, t+1, etc.
- No hierarchical structure but every part-of-speech is lexicalized, e.g. Verb is lexicalized by likes
- Language (set of strings) generated by finite-state grammars

# Finite State Grammars

A → a A
A → a

A1 ⇒ a1 A2
⇒ a1 a2 A3
⇒ a1 a2 a3 A4
⇒ a1 a2 ... aN




Terminal symbol: a
Non-terminal symbol: A



# Context-Free Grammars

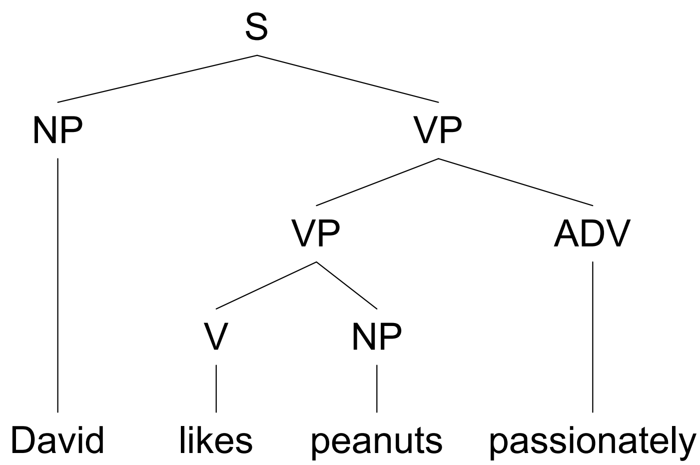
S → NP VP
VP → V NP | VP ADV
NP → David | peanuts
V → likes
ADV → passionately

• CFGs generate strings, e.g. language of G above is the set:

{ David likes peanuts,
David likes peanuts passionately,
... }

• Lexical sensitivity is lost
• CFGs also generate trees: hierarchical structure produced is non-trivial

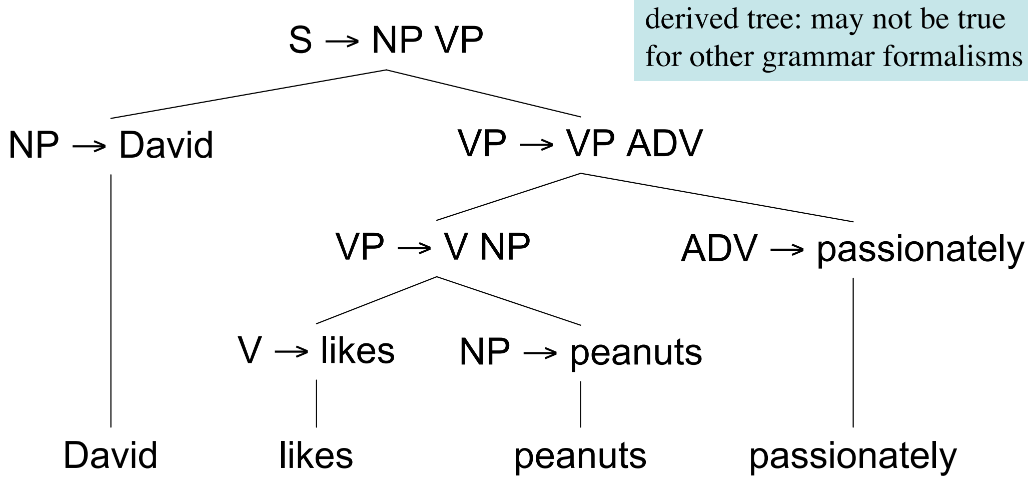
# CFG: Derived/Parse Tree






# CFG: Derivation Tree

derivation tree is identical to derived tree: may not be true for other grammar formalisms



# Preliminaries

- Rules of the kind $\alpha \rightarrow \beta$ where $\alpha$, $\beta$ are strings of terminals and non-terminals
- Chomsky hierarchy: regular, context-free, context-sensitive, recursively enumerable
- Automata: finite-state, pushdown, LBA, Turing machines (analysis of complexity of parsing)
- A rule $\alpha \rightarrow \beta$ in a grammar is lexicalized if $\beta$ contains a terminal symbol
- Lexicalization is a useful property, e.g. a rule like NP $\rightarrow$ NP creates infinite valid derivations



# Motivation #1

## Context-sensitive predicates on trees bear less fruit than you think*

* borrowed from a title of a paper by A. Joshi

# Strong vs. Weak Generative Capacity

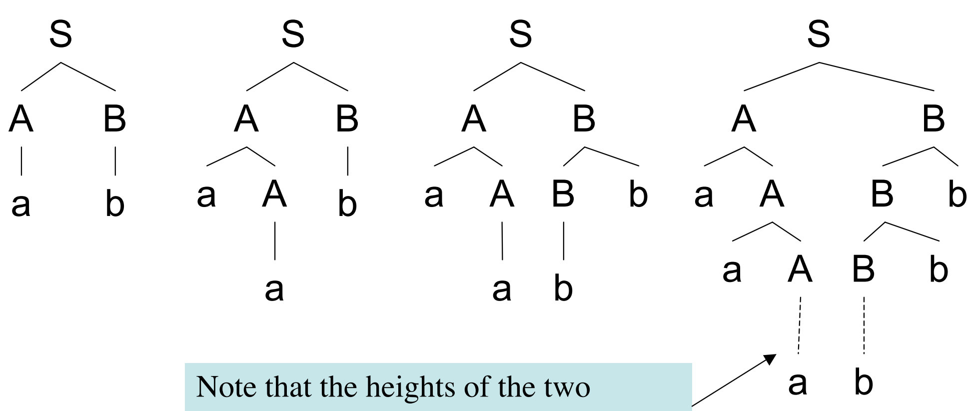
- A property of a formal grammar, e.g. of a regular grammar or a CFG
- **Weak Generative Capacity** of a grammar is the set of strings or the language
- **Strong Generative Capacity** of a grammar is the set of structures (usually the set of trees) produced by the grammar
- The tree for an utterance/sentence is the source of semantics, so the set of trees is more relevant for CL/NLP



# Tree Sets

$S \rightarrow A\ B$
$A \rightarrow a\ A \mid a$
$B \rightarrow B\ b \mid b$

This grammar generates the tree set shown

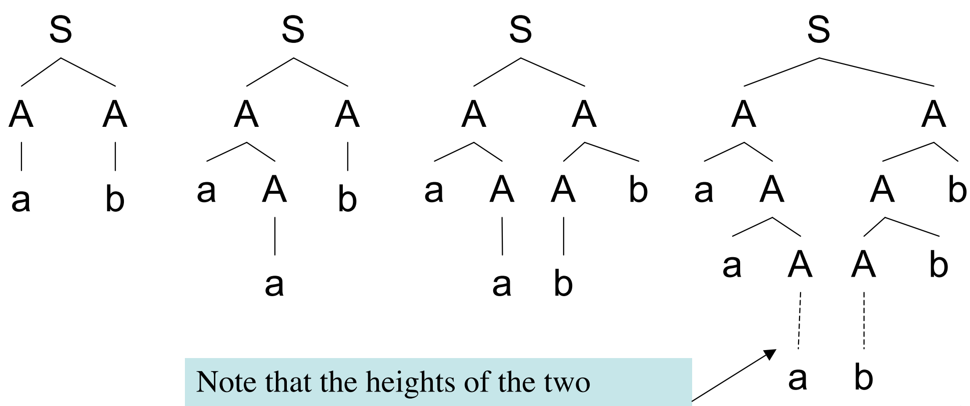



Note that the heights of the two branches do not have to be equal

# A Tree Set with no CFG

Claim: There is no CFG that can produce the tree set shown below:

~~S → A A~~
~~A → a A | a~~
~~A → A b | b~~




Note that the heights of the two branches do not have to be equal

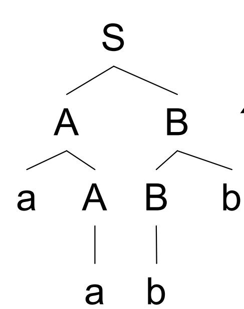


# Generating Tree Sets

- A simple trick: start with a CFG that almost works
- Then re-label the node labels, map B to A to get the desired tree set
- But how can we directly generate the tree sets?
- We need a **generative device** that generates *trees*, not *strings*
- (Thatcher, 1967) and (Rounds, 1970) provided such a generative device

S → A B
A → a A | a
B → B b | b

Map B to A

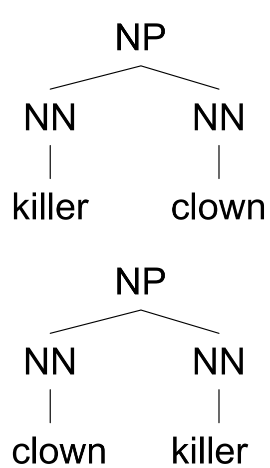


# Some definitions

- A **local set** is defined as the set of the set of trees generated by each CFG
- A **recognizable set** is the set of trees generated by each CFGs plus a re-labeling homomorphism
- The recognizable sets strictly contain the local sets
- Recognizable sets provide surprising connections between automata theory, decidability and logic
- They also provide insights into NLP tasks like parsing and syntax-based MT



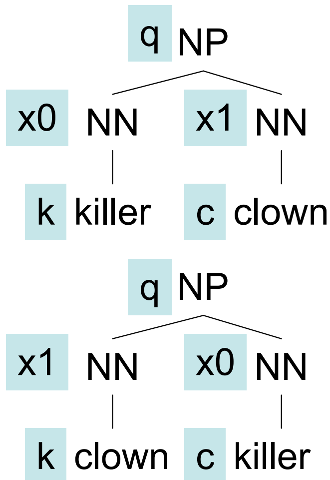
# Regular Tree Grammars




- Consider a simple tree set with two trees for the strings { *killer clown*, *clown killer* }
- No CFG can recognize this simple tree set without also recognizing trees for *clown clown* and *killer killer*
- A Regular Tree Grammar recognizes this tree set (analogy with regular grammars on strings)

Example from (May & Knight, 2006)

# Regular Tree Grammars




Example from (May & Knight, 2006)

start state: q
q → NP(x0 x1)
q → NP(x1 x0)
x0 → NN(k)
x1 → NN(c)
k → killer
c → clown

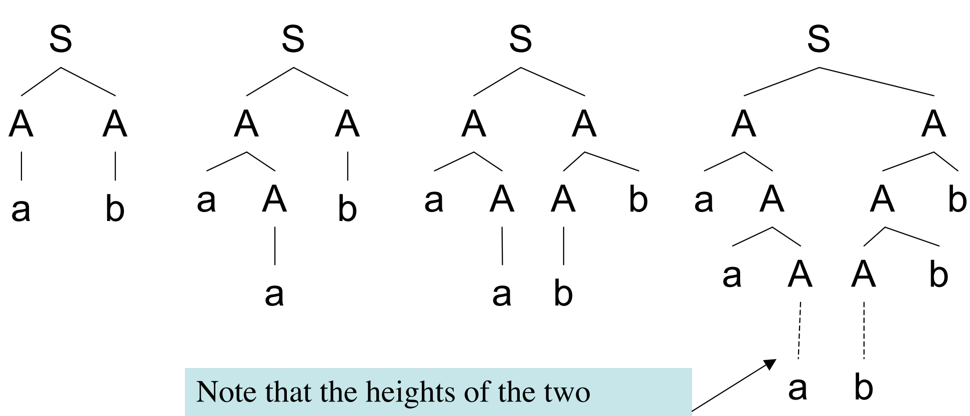
note: can be a tree of any size!

- RTGs = Top-down tree automata
- **Can generate infinite tree sets**
- **Currently used in syntax-based MT**
- for more: (Thatcher, 1967) (Rounds, 1970) (Graehl & Knight, 2004)



# Another RTG Example

q → S(x0 x1)
x0 → A(a x0) | a
x1 → A(x1 b) | b



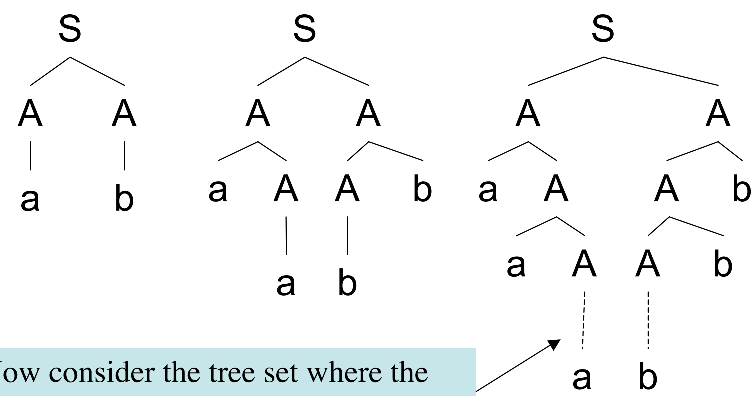

Note that the heights of the two branches do not have to be equal

# A Tree Set with no RTG

Claim: There is no RTG that can produce the tree set shown below:

~~q → S(x0 x1)~~
~~x0 → A(a x0) | a~~
~~x1 → A(x1 b) | b~~

RTG is like a regular grammar, the state cannot count how many times it was reached



Now consider the tree set where the depth of the two branches is **equal**

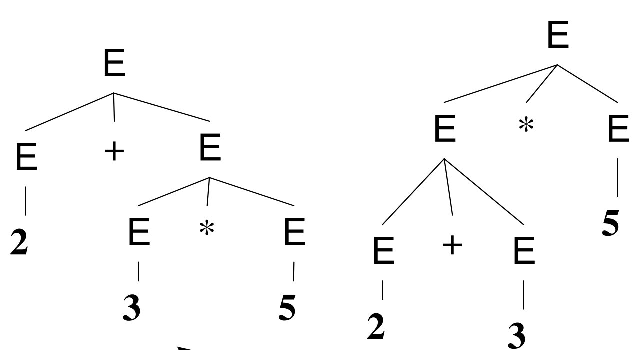


# Tree Sets: Another Example

A more practical example

E → E + E
E → E * E
E → ( E )
E → **N**

**2+3*5** is ambiguous either **17** or **25**



Ambiguity resolution: * has precedence over + cannot use RTGs!

# Tree Sets: Context-sensitivity

Eliminating ambiguity

$E \rightarrow E + E$

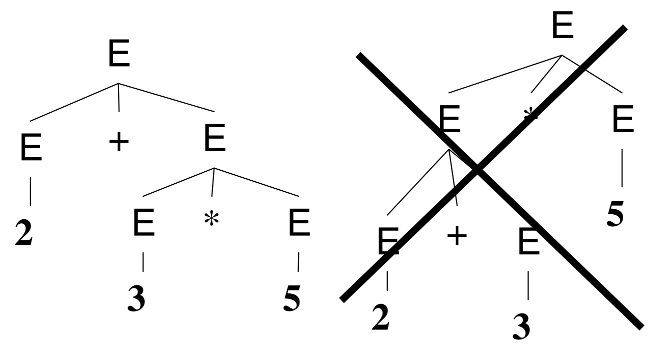
$\neg(+\_\_) \wedge \neg(*\_\_) \wedge \neg(\_\_*)$

$E \rightarrow E * E$

$\neg(*\_\_)$

$E \rightarrow ( E )$

$E \rightarrow \mathbf{N}$




**similar to context-sensitive grammars!**



# Context-sensitive Grammars

- Rules of the form $\alpha A \beta \rightarrow \alpha\gamma\beta$ where $\gamma$ cannot be the empty string, also written as $A \rightarrow \gamma$ / $\alpha\_\_\beta$
- CSGs are very powerful: they can generate languages like $\{ a^{2^\wedge i} : i > 0\}$
- This kind of computational power is unlikely to be useful to describe natural languages
- Like other grammar formalisms in the Chomsky hierarchy CSGs generate string sets
- What if they are used to **recognize** tree sets?

# Context-sensitive predicates

- Consider each CSG rule $A \rightarrow \gamma / \alpha\_\_\beta$ to be a **predicate** (i.e. either true or false)
- Apply all the rules in a CSG as predicates on an input tree
- If all predicates are true then accept the tree, else reject the tree
- Can be easily extended to a set of trees
- So a CSG can be used to accept a tree set
- Can we precisely describe this set of tree languages?



# Peters-Ritchie Theorem

- The Peters-Ritchie Theorem (Peters & Ritchie, 1967) states a surprising result about the generative power of CSG predicates
- Consider each tree set accepted by CSG predicates
- **Theorem**: The string language of this tree set is a context-free language
- Each CSG when applied as a set of predicates can be converted into a weakly equivalent CFG
- See also: (McCawley, 1967) (Joshi, Levy & Yueh, 1972) (Rogers, 1997)

# Local Transformations

- This theorem was extended by (Joshi & Levy, 1977) to handle arbitrary boolean combinations and sub-tree / domination predicates
- Proof involves conversion of all CSG predicates into top-down tree automata that accept tree sets
- (Joshi & Levy, 1977) result shows that **local transformations** used in early linguistic formalisms can be all written down as weakly equivalent CFGs
- Important caveat: we assume some source generating trees which are then validated



# Locality of CSG predicates

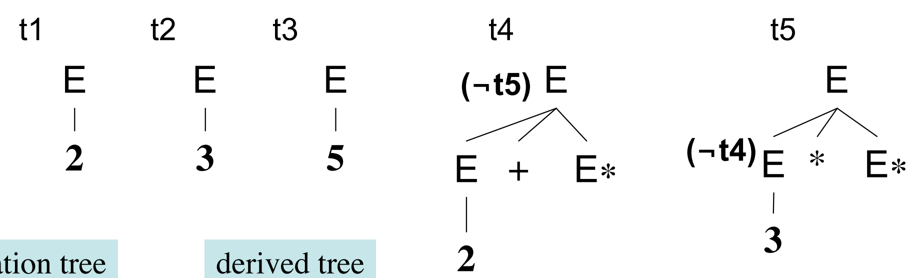
- An analysis of the kinds of CSG grammars used to define linguistic analyses in practice showed an interesting fact
- All the CSG predicates were very local
- They did not include in the context various parts of the tree that were arbitrarily far apart
- Long distance dependencies were expressed by chaining together many local CSG predicates
- This insight can be used to generate trees from an input string and validate them using CSG predicates

# Tree-Adjoining Grammars

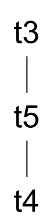
- Construct a tree set out of tree fragments
- Each fragment contains only the structure needed to express the locality of various CSG predicates
- Each tree fragment is called an elementary tree
- In general we need to expand even those non-terminals that are not leaf nodes: leads to the notion of adjunction



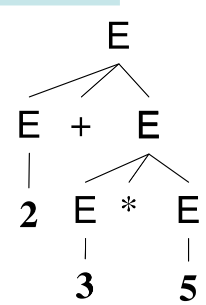
# Tree-Adjoining Grammars




derivation tree

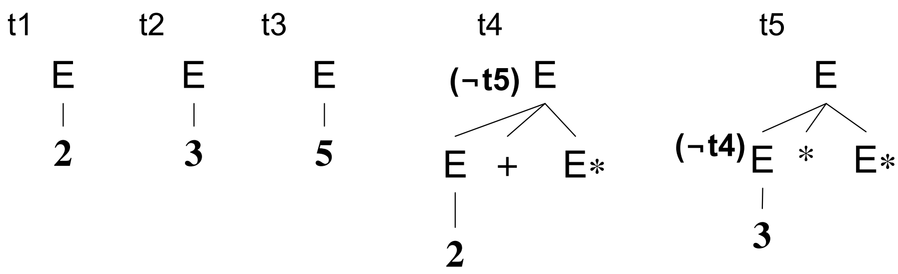


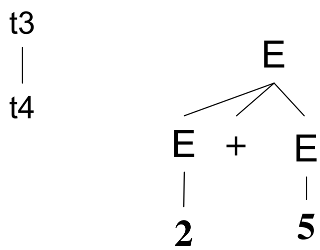

derived tree




all constraints are checked locally in the tree where adjunction occurs

# Tree-Adjoining Grammars

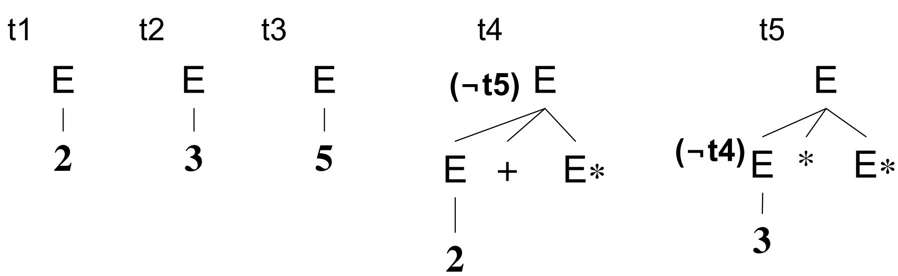


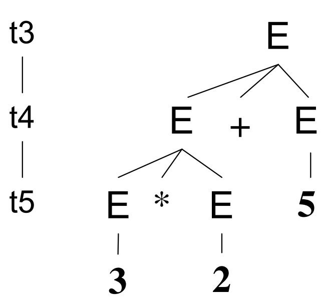




derivation for **3*****2**+**5**



# Tree-Adjoining Grammars




derivation for **3*****2**+**5**

# Motivation #2
# Lexicalization of Context-Free Grammars



# Lexicalization of Grammars

- We know that a CFG can be ambiguous: provide more than one parse tree for an input string
- A CFG can be infinitely ambiguous
- Structure can be introduced without influence from input string, e.g. the chain rule NP $\rightarrow$ NP has this effect
- Lexicalization of a grammar means that each rule or elementary object in the grammar is associated with some terminal symbol

# Lexicalization of Grammars

- Lexicalization is an interesting idea for syntax, semantics (in linguistics) and sentence processing (in psycho-linguistics)
- What if each word brings with it the syntactic and semantic context that it requires?
- Let us consider lexicalization of Context-free Grammars (CFGs)



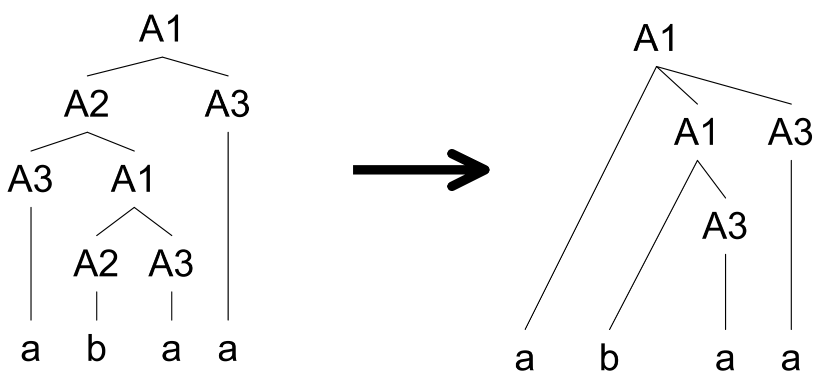
# Lexicalization of CFGs

- A **normal form** is a grammar transformation that does not change the language of the grammar
- Can we transform every CFG to a normal form where there is guaranteed to be a terminal symbol on the right hand side of each rule
- Answer: yes - using Greibach Normal Form (GNF)
- GNF: every CFG can be transformed into the form $A \rightarrow a\alpha$ where A is a non-terminal, *a* is a terminal and $\alpha$ is a string of terminals and non-terminals

# T(G) ≠ T(GNF(G))

A1 → A2 A3
A2 → A3 A1 | b
A3 → A1 A2 | a

Greibach Normal Form does not provide a strongly equivalent lexicalized grammar: the original tree set is not preserved

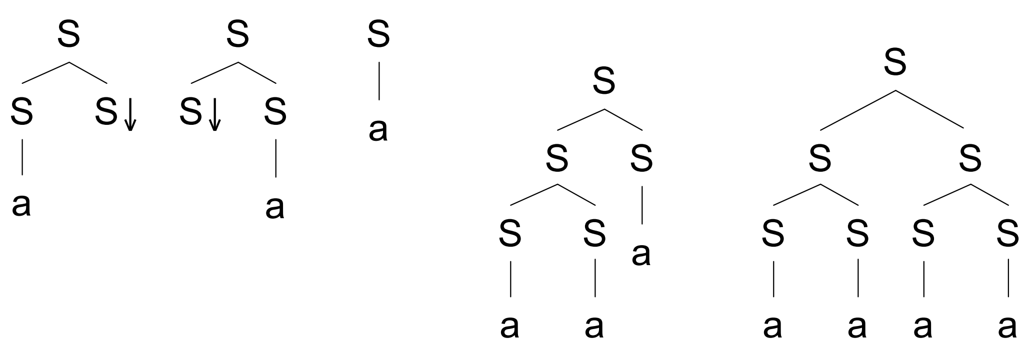





# Tree Substitution Grammar

S → S S
S → a

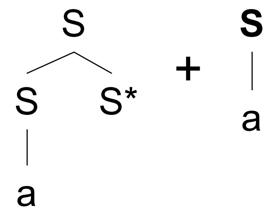
Consider a simple expansion of each context-free rule into a tree fragment where each fragment is lexicalized

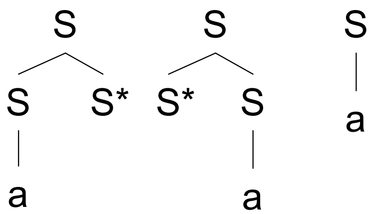


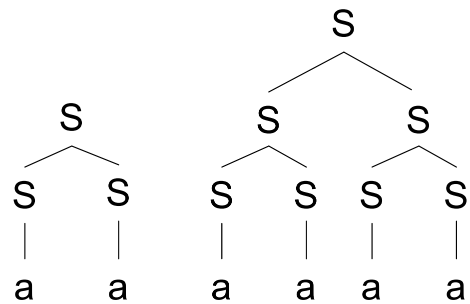

this tree cannot be derived

# Tree Adjoining Grammar

S → S S
S → a



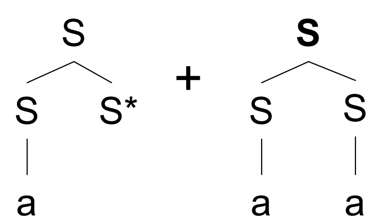


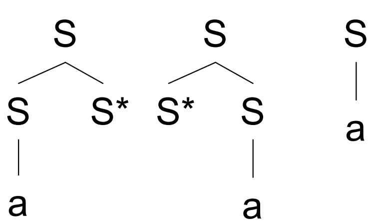


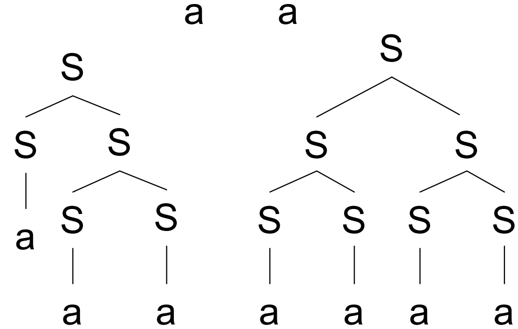


# Tree Adjoining Grammar

S → S S
S → a

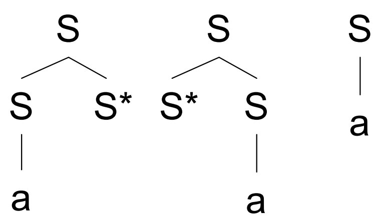


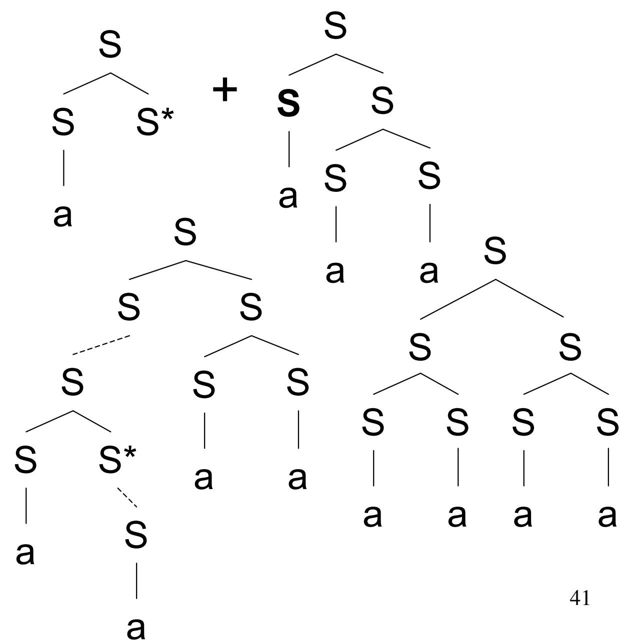




# Tree Adjoining Grammar

S → S S
S → a

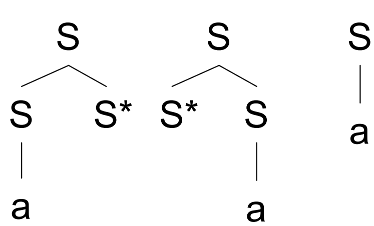


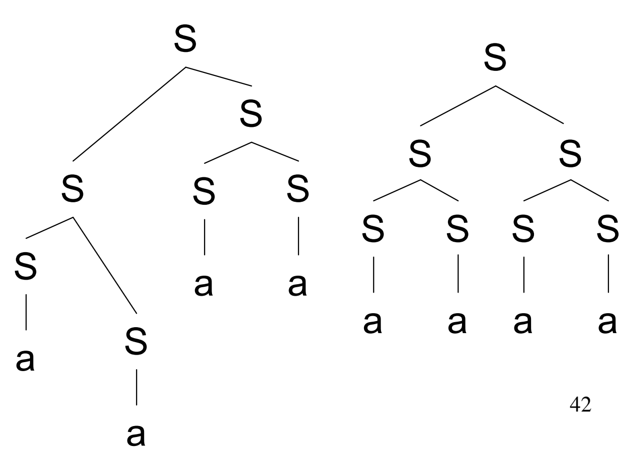




# Tree Adjoining Grammar

S → S S
S → a



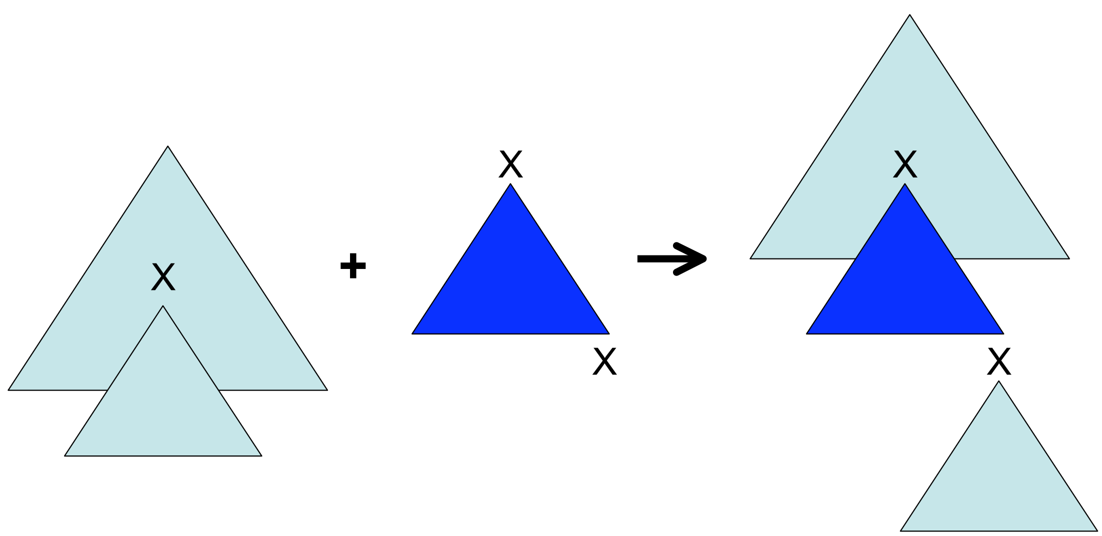


# Lexicalization Through TAG

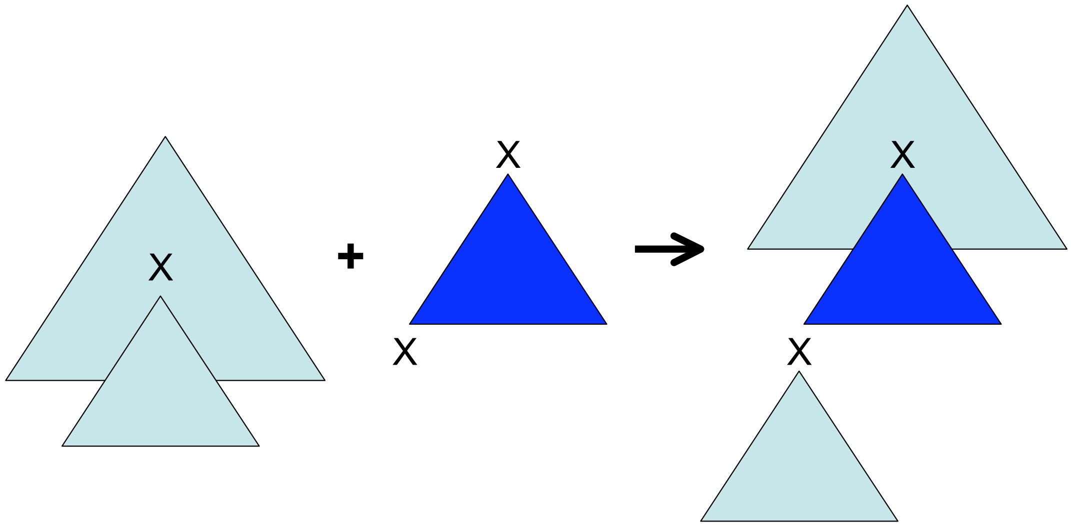
- This was an instructive example of how adjoining can be used to lexicalize CFGs while preserving the tree sets (strong generative capacity)
- (Joshi & Schabes, 1997) explain how every CFG can be strongly lexicalized by TAG
- (Joshi & Schabes, 1997) show that Tree-Adjoining Languages are closed under lexicalization: every TAL has a lexicalized TAG grammar



# Tree-Insertion with adjoining

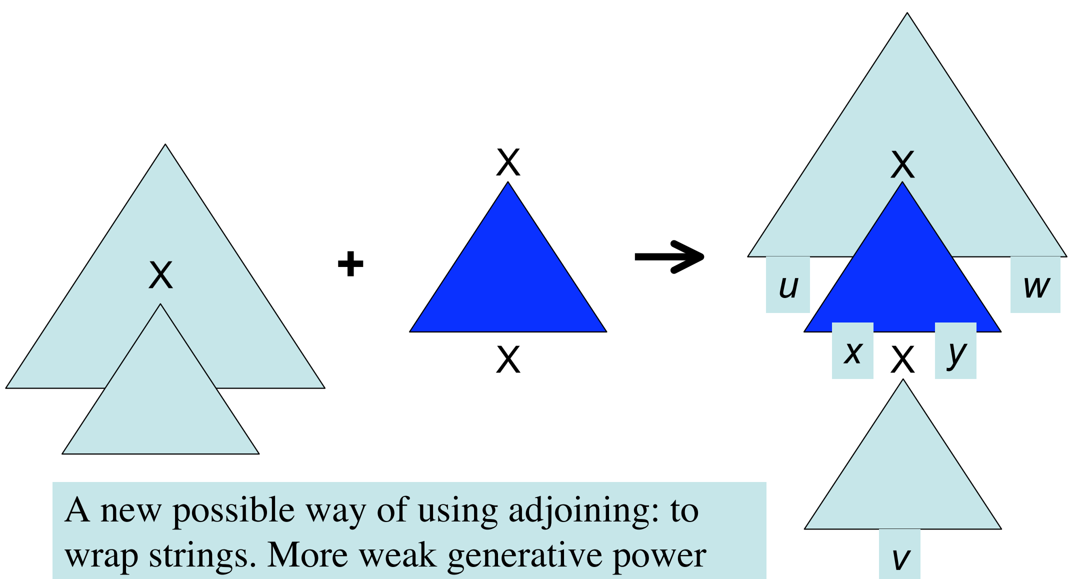


# Tree-Insertion with adjoining



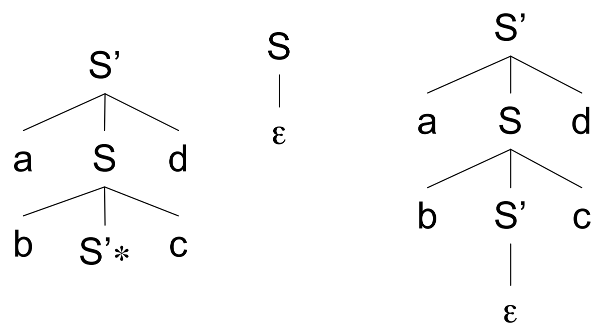


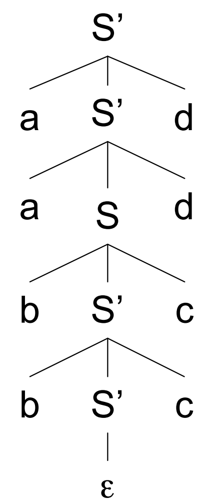

# Wrapping with adjoining




A new possible way of using adjoining: to wrap strings. More weak generative power than concatenation possible in CFGs.

# Wrapping with Adjoining




This TAG can generate the language
$L = \{ a^n b^n c^n d^n : n \geq 0 \}$
L is a context-sensitive language



# Motivation #3

# Is Human Language Regular, Context-free or Beyond?

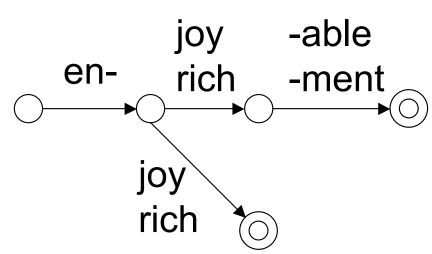
# Natural Language & Complexity

- One notion of computational complexity: the complexity of various recognition and generation algorithms
- Another notion: the complexity of the description of human languages
- What is the lowest upper bound on the description of all human languages? regular, context-free or beyond?
- Describes a class of languages, including closure properties such as union, intersection, etc.
- Automata theory provides recognition algorithms, determinization, and other algorithms



# Grammar Size

- Consider the set of strings that includes *enjoy, enrich, enjoyable, enrichment* but not **joyable, *richment*
- The CFG is clearly more compact
- Argument from learning: if you already know *enjoyment* then learning *rich* means you can generate *enrichment* as well




V → X
A → X -able | X -ment
X → en- NA
NA → joy | rich

Regular grammars can be exponentially larger than equivalent CFGs

# Sufficient Generative Capacity

- Does a formal grammar have sufficient generative capacity?
- Two cases: **weak** and **strong** generative capacity
- For strong GC: does the grammar permit the right kind of dependencies, e.g. nested dependencies
- For weak GC: usually requires some kind of homomorphism into a formal language whose weak GC can be determined (the formal language class should be closed under homomorphisms)



# Is NL regular: strong GC

- Regular grammars cannot derive nested dependencies
- Nested dependencies in English:
  - the shares that the broker recommended were bought
    N1 N2 V2 V1
  - the moment when the shares that the broker recommended were bought has passed
    N1 N2 N3 V3 V2 V1
- Can you provide an example with 4 verbs?
- Set of strings has to be infinite: competence vs. performance

# Is NL regular: strong GC

- Assume that in principle we could process infinitely nested dependencies: **competence** assumption
- The reason we cannot is because of lack of memory in pushdown automata: **performance** can be explained
- CFGs can easily obtain nested dependencies

$S \rightarrow a\ S\ b$
$S \rightarrow \varepsilon$

$S1 \Rightarrow a1\ S2\ b1$
$\Rightarrow a1\ a2\ S3\ b2\ b1$
$\Rightarrow a1\ a2\ ...\ aN\ S\ bN\ ...\ b2\ b1$
$\Rightarrow a1\ a2\ ...\ aN\ bN\ ...\ b2\ b1$



# Is NL regular: Weak GC

- Consider the following set of strings (sentences):
  - S = *if* S *then* S
  - S = *either* S *or* S
  - S = *the man who said* S *is arriving today*
- Map *if, then* to *a* and *either, or* to *b*
- Map everything else to the empty string
- This results in strings like *abba, abaaba*, or *abbaabba*

## Is NL regular: Weak GC

- The language is the set of strings

  L = { ww' : w from (a|b)* and w' is reversal of w }
- L can be shown to be non-regular using the pumping lemma for regular languages
- L is context-free



## Is NL context-free: Strong GC

- CFGs cannot handle crossing dependencies
- Dependencies like aN... a2 a1 bN... b2 b1 are not possible using CFGs
- But some widely spoken languages have clear cases of crossing dependencies
  - Dutch (Bresnan et al., 1982)
  - Swiss German (Shieber, 1984)
  - Tagalog (Rambow & MacLachlan, 2002)
- Therefore, in terms of strong GC, NL is not context-free

# Is NL context-free: Weak GC

- Weak GC of NL being greater than context-free was harder to show, cf. (Pullum, 1982)
- (Huybregts, 1984) and (Shieber, 1985) showed that weak GC of NL was beyond context-free using examples with explicity case-marking from Swiss-German

| | | | | | | |
|---|---|---|---|---|---|---|
| mer | d' chind | em Hans | es huus | lönd | hälfed | aastriiche |
| we | children-**acc** | Hans-**dat** | house-**acc** | let-**acc** | help-**dat** | **paint-acc** |
| | [ | ( | { | ] | ) | } |



this language is not context-free

# Generating Crossing Dependencies

1: S → S B C
2: S → a C
3: a B → a a
4: C B → B C
5: B a → a a
6: C → b

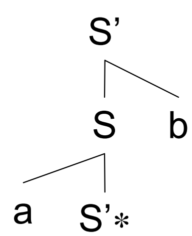
S1 ⇒ **S2 B1 C1** (1)
⇒ **S3 B2 C2** B1 C1 (1)
⇒ **a3 C3** B2 C2 B1 C1 (2)
⇒ a3 **B2 C3** C2 B1 C1 (4)
⇒ **a3 a2** C3 C2 B1 C1 (3)
⇒ a3 a2 C3 **B1 C2** C1 (4)
⇒ a3 a2 **B1 C3** C2 C1 (4)
⇒ a3 **a2 a1** C3 C2 C1 (3)
⇒ a3 a2 a1 **b3** C2 C1 (6)
⇒ a3 a2 a1 b3 **b2** C1 (6)
⇒ a3 a2 a1 b3 b2 **b1** (6)

# Simple Generation of Crossing Dependencies

- Instead of using powerful swapping operations (corresponding to more powerful automata)
- We instead build local dependencies into elementary trees
- Strong GC: Crossing dependencies arise by simple composition of elementary trees
- The context-sensitive part is built into each elementary tree: the remaining composition is "context-free"
- Weak GC: Crossing dependencies = string wrapping

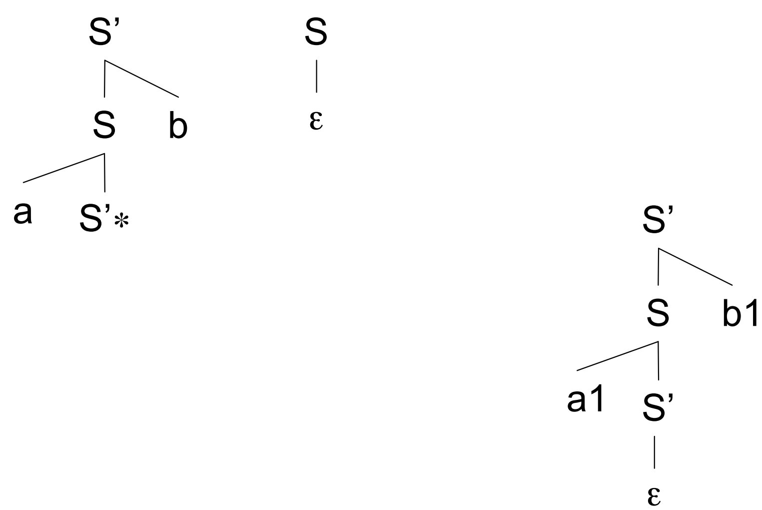


# Crossing Dependencies with Adjoining



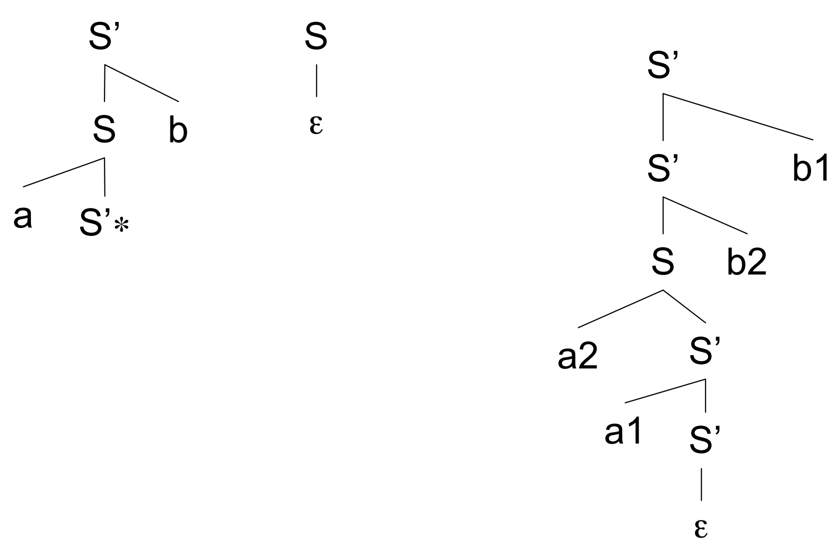


# Crossing Dependencies with Adjoining

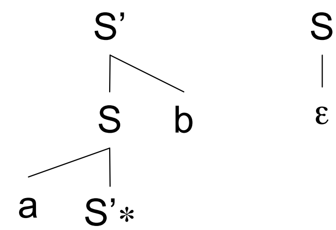


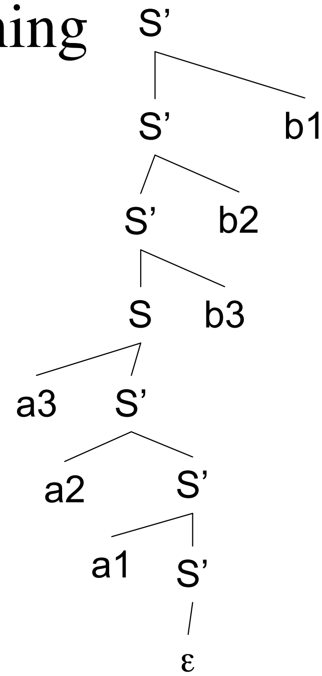



# Crossing Dependencies with Adjoining



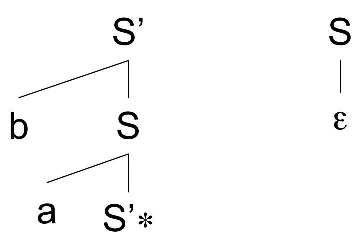
# Crossing Dependencies with Adjoining

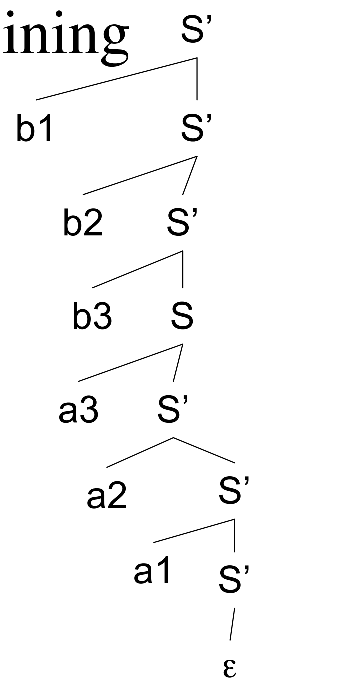





# Nested Dependencies with Adjoining



# Tractable Descriptions

- Why not use context-sensitive grammars?
- For G, given a string *x* what is the complexity of an algorithm for the question: is *x* in L(G)?
  - Unrestricted Grammars/Turing machines: undecidable
  - Context-sensitive: NSPACE[n] linear non-deterministic space
  - Indexed Grammars: NP-complete
  - Tree-Adjoining Grammars: $O(n^6)$
  - Context-free: $O(n^3)$
  - Regular: $O(n)$



# Tractable Descriptions

- Another route to lexicalization of CFGs: categorial grammars (CG is not strongly equivalent to CFGs but can lexicalize them)
- Several different mathematically precise formal grammars were proposed to deal with the motivations presented here
- Some examples: head grammars (HG does string wrapping); combinatory categorial grammars (CCG; extends CG); linear indexed grammars (LIG; less powerful than indexed grammars)
- Using formal methods introduced with TAG, (Vijay-Shanker, 1987) and (Weir, 1988) showed that HG, CCG, LIG and TAG are all *weakly equivalent*!

# Tree-Adjoining Grammars: Definition and Application to NL



# Tree-Adjoining Grammars

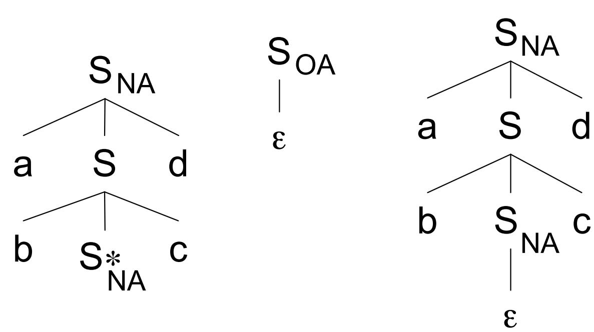
- A TAG G = (N, T, I, A, S) where
  - N is the set of non-terminal symbols
  - T is the set of terminal symbols
  - I is the set of initial or non-recursive trees built from N, T and domination predicates
  - A is the set of recursive trees: one leaf node is a non-terminal with same label as the root node
  - S is set of start trees (has to be initial)
  - I and A together are called *elementary trees*

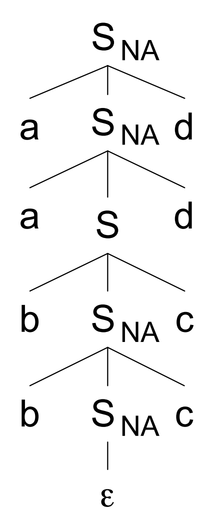
# Adjunction Constraints

- Adjunction is the rewriting of a non-terminal in a tree with an auxiliary tree
- We can think of this operation as being “context-free”
- Constraints are essential to control adjunction: both in practice for NLP and for formal closure properties
- Three types of constraints:
  - null adjunction (NA): no adjunction allowed at a node
  - obligatory adjunction (OA): adjunction must occur at a node
  - selective adjunction (SA): adjunction of a pre-specified set of trees can occur at a node



# Adjunction Constraints

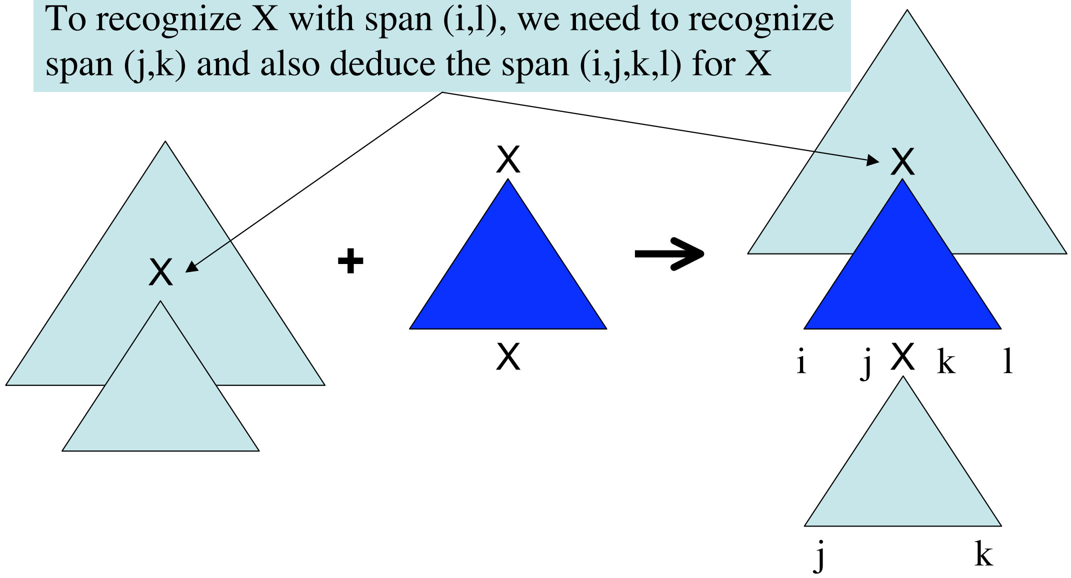



This TAG can generate the language
$L = \{ a^n b^n c^n d^n : n \geq 1 \}$
Note that the OA & NA constraints are crucial to obtain the correct language

# Parsing Complexity: CKY for TAG

To recognize X with span (i,l), we need to recognize span (j,k) and also deduce the span (i,j,k,l) for X

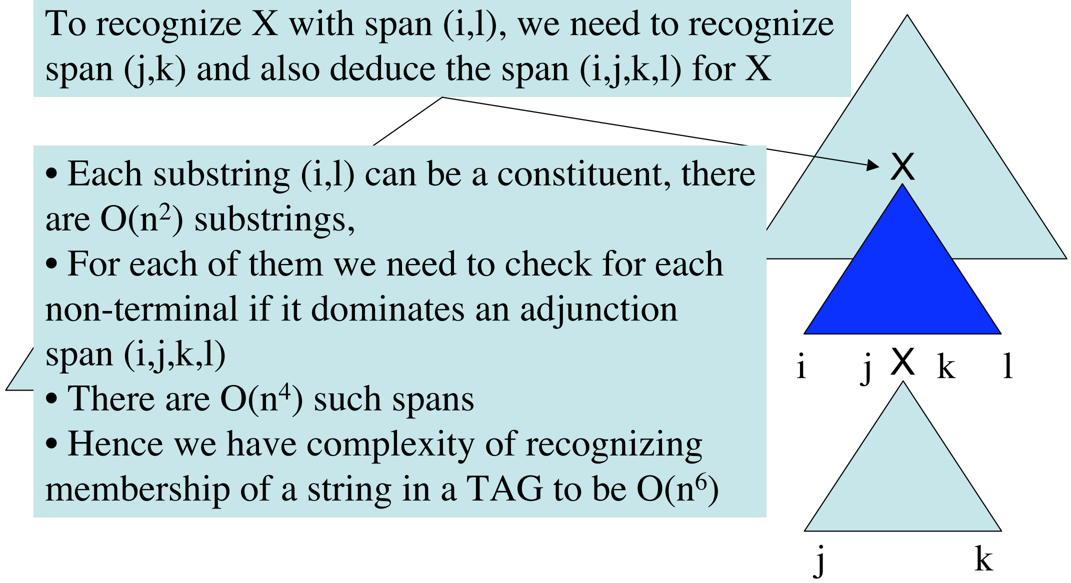





# Parsing Complexity: CKY for TAG

To recognize X with span (i,l), we need to recognize span (j,k) and also deduce the span (i,j,k,l) for X

- Each substring (i,l) can be a constituent, there are O($n^2$) substrings,
- For each of them we need to check for each non-terminal if it dominates an adjunction span (i,j,k,l)
- There are O($n^4$) such spans
- Hence we have complexity of recognizing membership of a string in a TAG to be O($n^6$)



# TAG Formal Properties

(Vijay-Shanker, 1987)

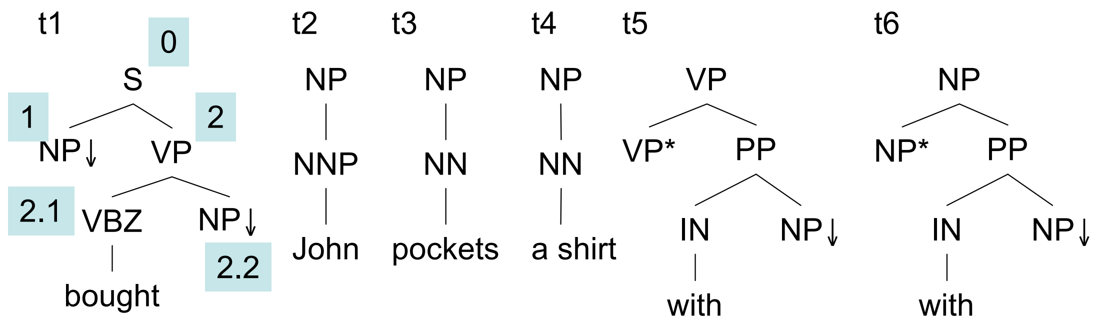
- Membership is in P: $O(n^6)$
- Tree-Adjoining Languages (TALs) are closed under *union*, *concatenation*, *Kleene closure* (*), *h*, $h^{-1}$, *intersection with regular languages*, and *regular substitution*
- There is also a pumping lemma for TALs
- TALs are a full abstract family of languages (AFL)
- TALs are not closed under intersection, intersection with CFLs, and complementation



# Lexicalized TAG

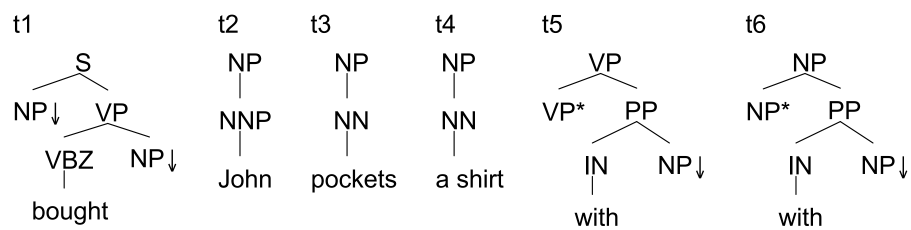
- A Lexicalized TAG (LTAG) is a TAG where each elementary tree has at least one terminal symbol as a leaf node
- A non-lexicalized TAG can always be converted to a lexicalized TAG (Joshi & Schabes, 1997)
- Lexicalization has some useful effects:
  - finite ambiguity: corresponds to our intuition about NL ambiguities,
  - statistical dependencies between words can be captured which can improve parsing accuracy

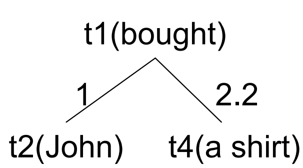
# Lexicalized TAG: example

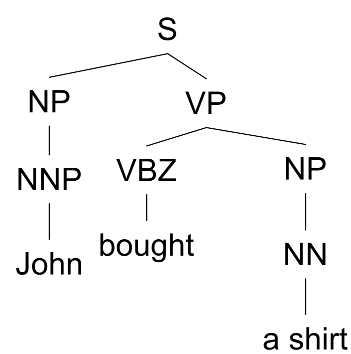



Gorn tree address: an index for each node in the tree



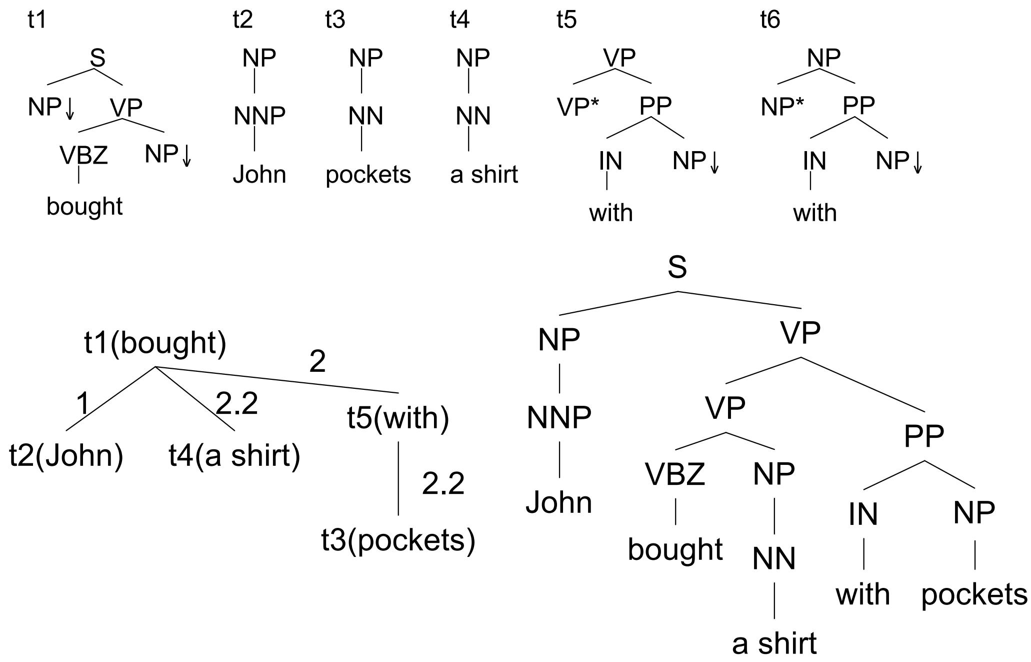



derivation tree

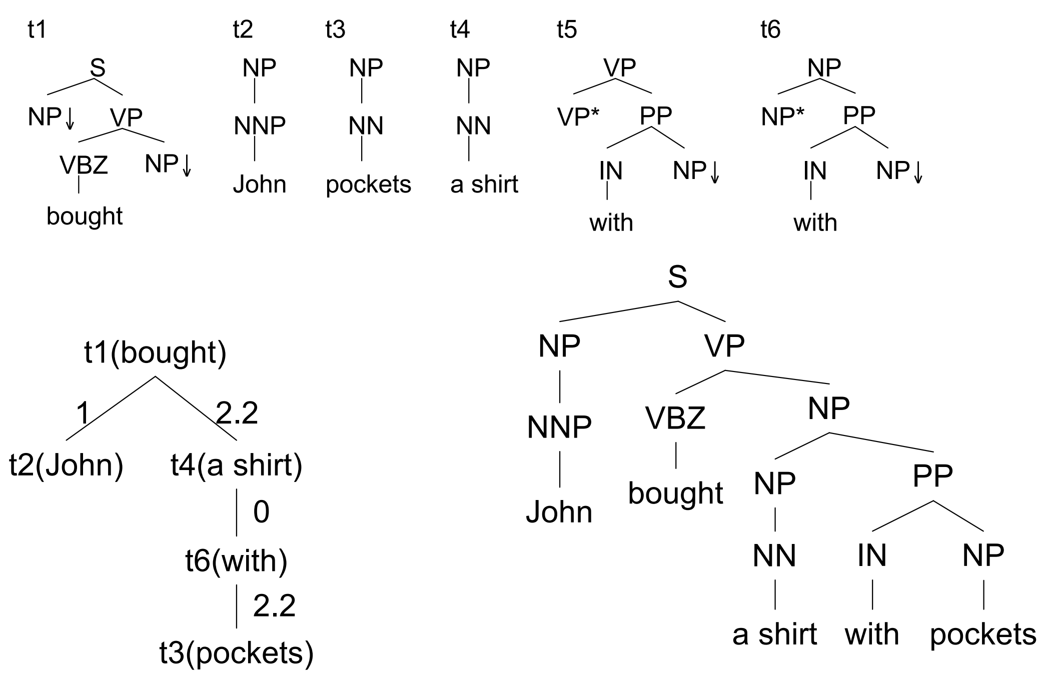
derived tree




derivation tree

derived tree






derivation tree

derived tree

# Comparison with Dependency Grammar

- Compare the derivation tree with the usual notion of a dependency tree
- Note that a TAG derivation tree is a formal representation of the derivation
- In a Lexicalized TAG, it can be interpreted as a particular kind of dependency tree
- Different dependencies can be created by changing the elementary trees
- LTAG derivations relate dependencies between words to detailed phrase types and constituency



# Localization of Dependencies

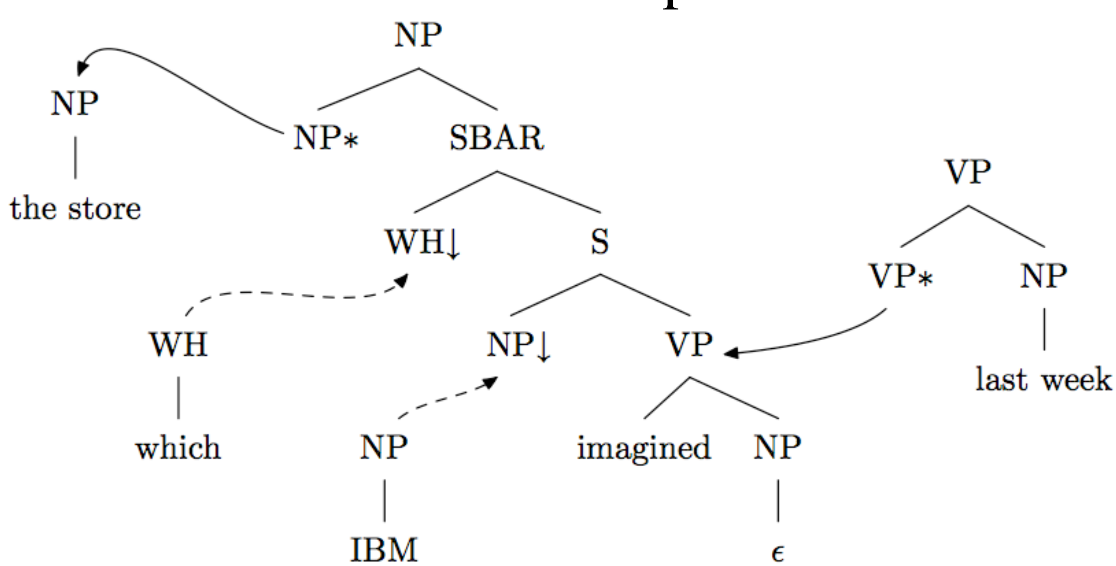
- Syntactic
  - agreement: person, number, gender
  - subcategorization: *sleeps* (null), *eats* (NP), *gives* (NP NP)
  - filler-gap: *who$_i$ did John ask Bill to invite t$_i$*
  - word order: within and across clauses as in scrambling, clitic movement, etc.

# Localization of Dependencies

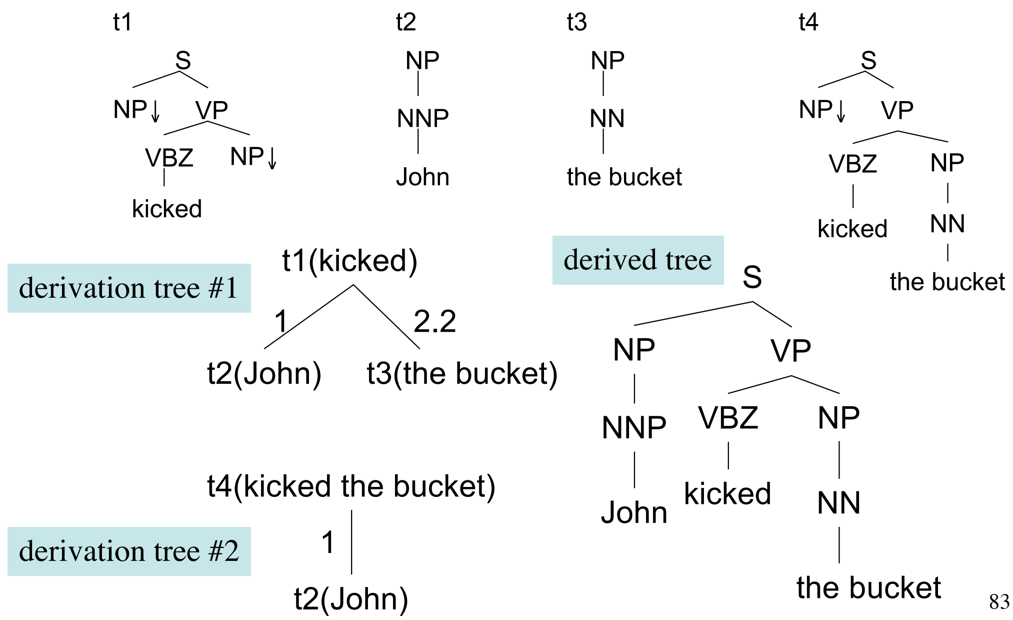
- Semantic
  - function-argument: all arguments of the word that lexicalizes the elementary tree (also called the *anchor* or *functor*) are localized
  - word clusters (word idioms): non-compositional meaning, e.g. *give a cold shoulder to, take a walk*
  - word co-occurences, lexical semantic aspects of word meaning



# Localization of Dependencies

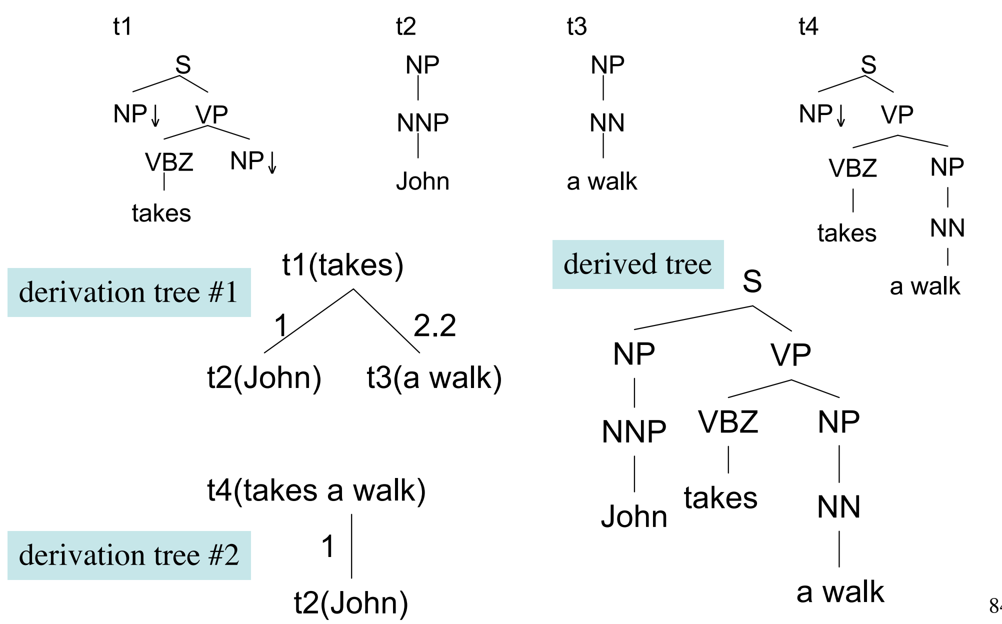


# Idioms




83

# Phrasal/Light Verbs




84

# TAG and Generation

- TAG has some useful properties with respect to the problem of NL generation
- Adjunction allows a generation planning system to add useful lexical information to existing constituents
- Makes planning for generation output more flexible
  - e.g. if the system has a constituent *the book*, it can choose to add new information to it: *the red book*, if there is a distractor for that entity for the hearer
- cf. Matthew Stone's SPUD system



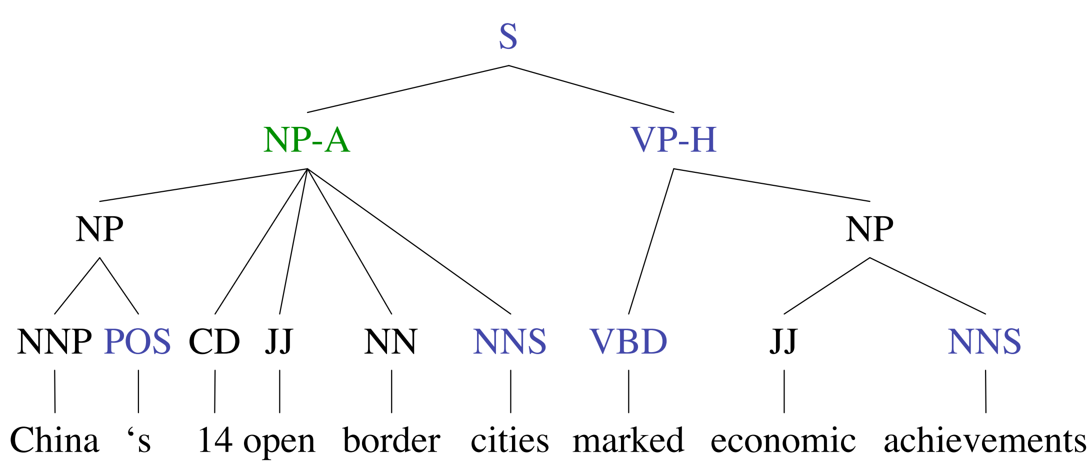
# Mapping a TreeBank into Lexicalized TAG derivations

# LTAG Derivations from TreeBanks

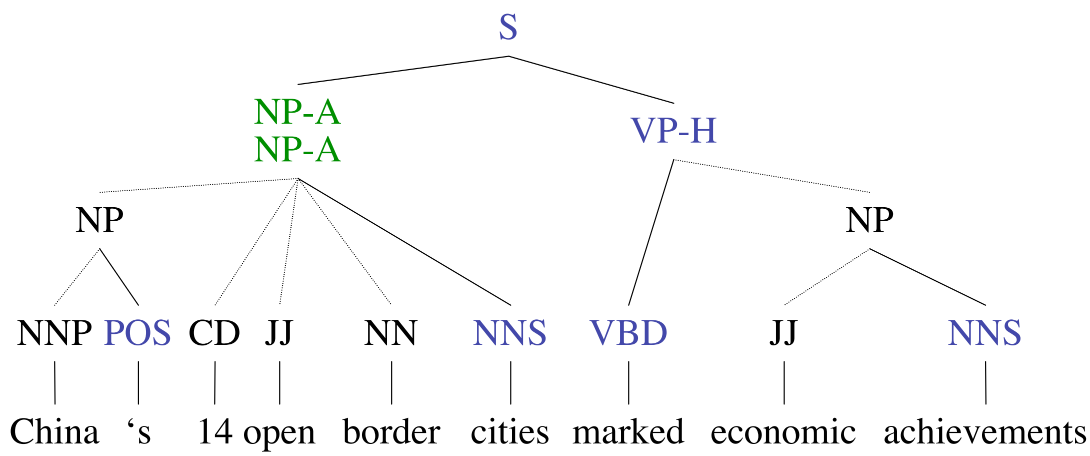
- TreeBanks contain phrase structure trees or dependency trees
- Converting dependency trees into LTAG is trivial
- For phrase structure trees: exploit head percolation rules (Magerman, 1994) and argument-adjunct heuristic rules
- First mark TreeBank tree with head and argument information
- Then use this information to convert the TreeBank tree into an LTAG derivation
- More sophisticated approaches have been tried in (Xia, 1999) (Chiang, 2000) and (Chen, 2000)



# LTAG derivations from TreeBanks

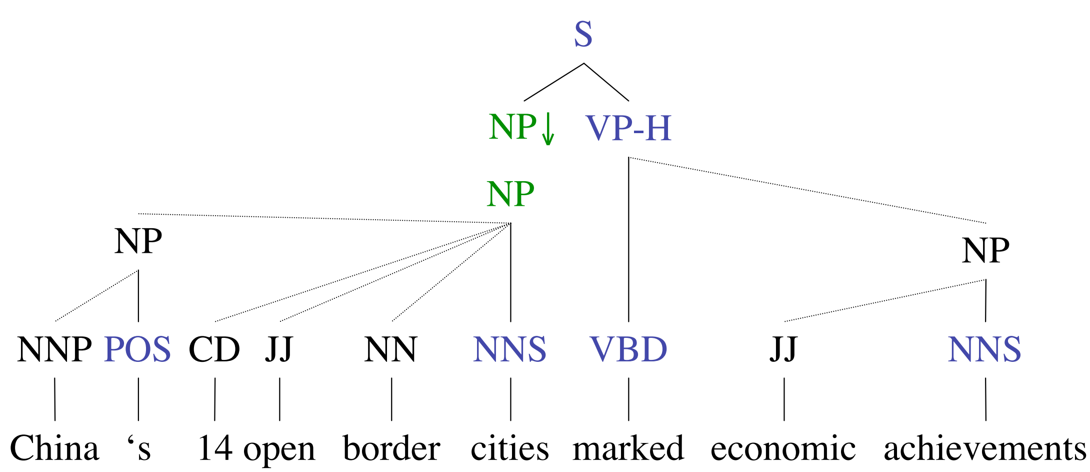


# LTAG derivations from TreeBanks

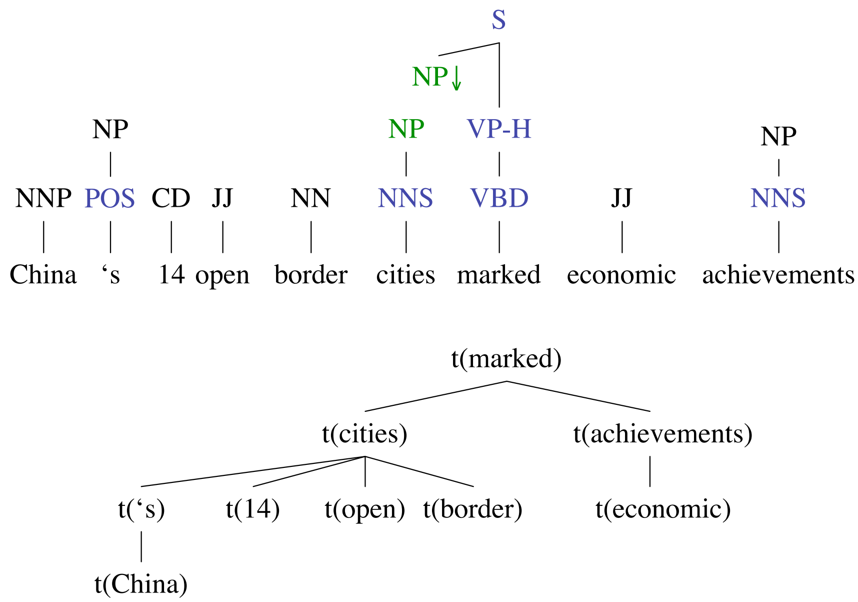





# LTAG derivations from TreeBanks



# LTAG derivations from TreeBanks





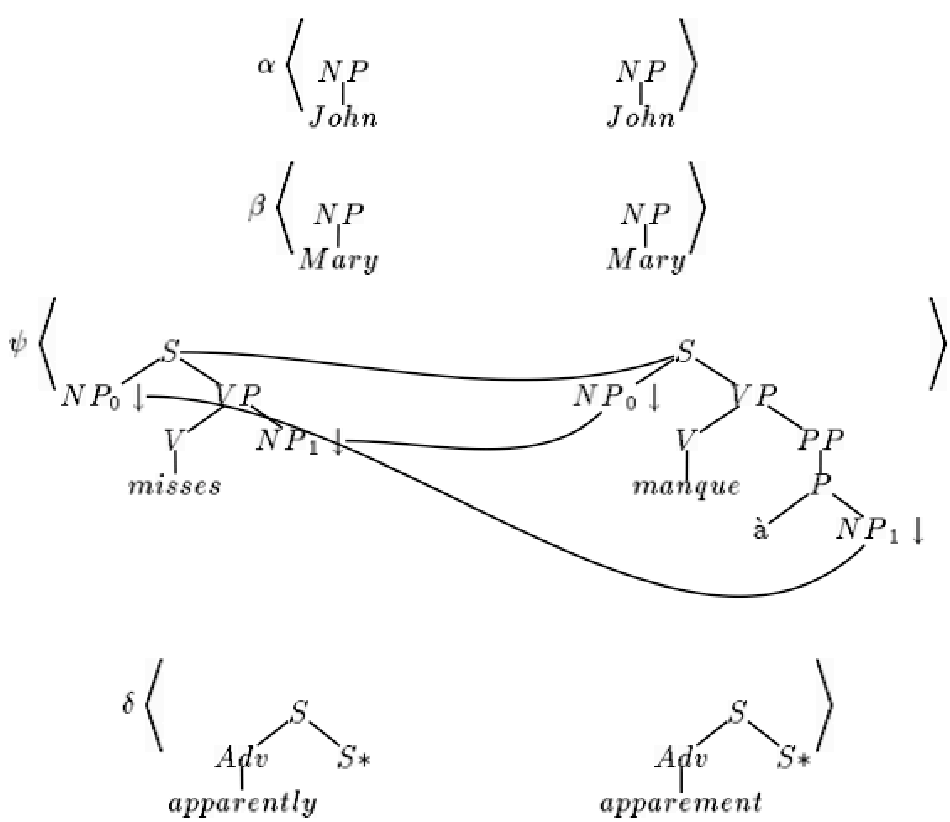

# Synchronous TAG

# Synchronous TAG

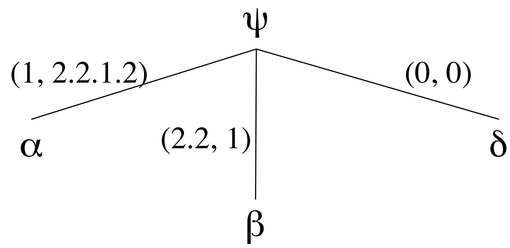
(Shieber, 1994)

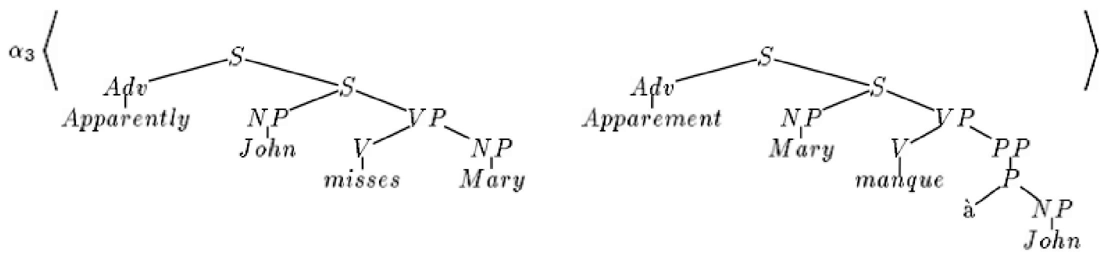
- Just like TAG we have derivation trees
- Except each node in the derivation is not a single elementary tree but rather a pair of trees
- The derivation tree now can be used to build a pair of derived trees
- Synchronous TAG can be used to generate a pair of derived trees or map a source input string to target output string
- Applications: NL semantics (scope ambiguity, etc.) and syntax-based machine translation





# Synchronous TAG






# Bibliography

(Thatcher, 1967): J. W. Thatcher, *Characterizing derivation trees of context-free grammars through a generalization of finite-automata theory*, J. Comput. Sys. Sci., 1 (1967), pp. 317-322

(Rounds, 1970): W. C. Rounds, *Mappings and grammars on trees*, Math. Sys. Theory 4 (1970), pp. 257-287

(Peters & Ritchie, 1969): P. S. Peters and R. W. Ritchie, *Context sensitive immediate constituent analysis -- context-free languages revisited*, Proc. ACM Symp. Theory of Computing, 1969.

(Joshi & Levy, 1977): A. K. Joshi and L. S. Levy, *Constraints on Structural Descriptions: Local Transformations*, SIAM J. of Comput. 6(2), June 1977.

(May & Knight, 2006): J. May and K. Knight, *Tiburon: a weighted automata toolkit*, In Proc. CIAA, Taipei, 2006

(Graehl & Knight, 2004): J. Graehl and K. Knight, *Training tree transducers*, In Proc. of HLT-NAACL, Boston, 2004

# Bibliography


(Joshi, 1994): A. K. Joshi, *From Strings to Trees to Strings to Trees ...*, Invited Talk at ACL'94, June 28, 1994.

(Joshi & Schabes, 1997): Joshi, A.K. and Schabes, Y.; Tree-Adjoining Grammars, in *Handbook of Formal Languages*, G. Rozenberg and A. Salomaa (eds.), Vol. 3, Springer, Berlin, New York, 1997, 69 - 124.

(Schabes & Shieber, 1994): Yves Schabes and Stuart M. Shieber. An Alternative Conception of Tree-adjoining Derivation. *Computational Linguistics*, 20(1), March 1994.

(Shieber, 1994): Stuart M. Shieber. Restricting the weak-generative capacity of synchronous tree-adjoining grammars. *Computational Intelligence*, 10(4):371-385, November 1994. cmp-lg/9404003.