Grammar formalisms
Tree Adjoining Grammar: Formal Properties, Parsing

Laura Kallmeyer, Timm Lichte, Wolfgang Maier
Universität Tübingen

16.05.2007 und 21.05.2007

Part I

Formal Properties of TAG

Outline: Formal Properties of TAG

1. Some sample languages
2. Pumping Lemma for TAL
3. Closure Properties

Some sample languages (1)

Central question: How does the class of Tree Adjoining Languages (TAL) look like?
Some languages that are in $TAL \setminus CFL$:

- The copy language

$$\{ww \mid w \in \{a, b\}^*\}$$

- The counting languages for 3 and 4:

$$\{{a_1}^k {a_2}^k {a_3}^k \mid k \geq 0\}$$

$$\{{a_1}^k {a_2}^k {a_3}^k {a_4}^k \mid k \geq 0\}$$

Some sample languages (2)

Some languages that are not in *TAL*:

- The double copy language

$$\{www \mid w \in \{a, b\}^*\}$$

  In general, any copy language with more than one copy following the first w is not in *TAL*.
- The counting languages for $n > 4$:

$$\{a_1{}^k a_2{}^k \ldots a_n{}^k \mid k \geq 0\}$$

- Languages of exponential growth:

$$\{a^{2^k} \mid k \geq 0\}$$

Some sample languages (3)

$\Rightarrow$ TAG extend CFG but only in a very limited way.

In order to situate a class of languages with respect to other classes, one needs to know something about the properties of this class. Particularly useful:

- Pumping Lemmas
- Closure Properties

Pumping Lemma for TAL (1)

For CFL, the following pumping lemma holds:
Let L be a context-free language. Then there is a constant c such that for all $w \in L$ with $|w| \geq c$: $w = xv_1yv_2z$ with

- $|v_1v_2| \geq 1$,
- $|v_1yv_2| \leq c$, and
- for all $i \geq 0$: $xv_1{}^iyv_2{}^iz \in L$.

Pumping Lemma for TAL (2)

The reason why this is so is the following:

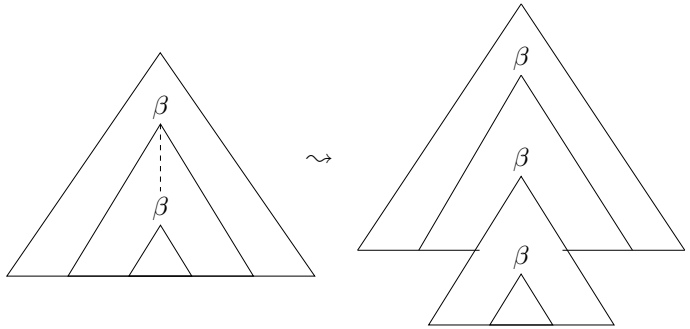
- In the context-free tree, from a certain tree height on, there is always a path with two occurences of the same non-terminal.
- Then the part between the two occurrences can be iterated. This means that the strings to left and the right of this part are pumped.

How about TAL?

- The TAG derivation trees are context-free.
- Therefore, the same iteration is possible here.

Pumping Lemma for TAL (3)

Iteration in TAG derivation trees:



Pumping Lemma for TAL (4)

Looking at what this means for the strings, one can show the following:
Pumping Lemma for TAL:
If L is a TAL, then there is a constant c such that if $w \in L$ and $|w| \geq c$, then there are $x, y, z, v_1, v_2, w_1, w_2, w_3, w_4 \in T^*$ such that

- $|v_1 v_2 w_1 w_2 w_3 w_4| \leq c$,
- $|w_1 w_2 w_3 w_4| \geq 1$,
- $x = x v_1 y v_2 z$, and
- $x {w_1}^n v_1 w_2{}^n y w_3{}^n v_2 w_4{}^n z \in L(G)$ for all $n \geq 0$.

Vijayashanker (1987) even claims that a stronger version of this lemma holds, but in his proof, one step is not clear. Therefore we use this weak form.

Pumping Lemma for TAL (5)

Pumping lemmas can be used to show that certain languages are not in a certain class.
Example:

- To show: $L = \{a^n b^m a^n b^m a^n b^m \mid n, m \geq 0\}$ is not a TAL.
- Assume that L is a TAL and therefore satisfies the pumping lemma with a constant c. Consider the word $w = a^{c+1} b^{c+1} a^{c+1} b^{c+1} a^{c+1} b^{c+1}$.
- None of the w_i, $1 \leq i \leq 4$ from the pumping lemma can contain both a's and b's. Furthermore, at least three of them must contain the same letters and be inserted into the three different a^{c+1} respectively or into the three different b^{c+1}.
- Contradiction since then either $|v_1| \geq c + 1$ or $|v_2| \geq c + 1$.

Pumping Lemma for TAL (6)

As a corollary of the pumping lemma, one obtains that *TAL* are of constant growth (the word length grows in a linear way):
A language L has the *constant growth property* iff there is a constant $c_0 > 0$ and a finite set of constants $C \subset \mathbb{N} \setminus \{0\}$ such that for all $w \in L$ with $|w| > c_0$, there is a $w' \in L$ with $|w| = |w'| + c$ for some $c \in C$.

Closure Properties (1)

It is often useful to reduce a language L to a simpler language before showing that it is not in a certain class C. This can be done with closure properties.
TAL are closed under

- union, concatenation, Kleene closure and substitution.
- homomorphisms, intersection with regular languages, and inverse homomorphisms.

$\Rightarrow$ TALs form a substitution closed *Full Abstract Family of Languages (AFL)*. (Full AFL = closed under intersection with regular languages, homomorphisms, inverse homomorphisms, union, concatenation and Kleene star.)

Closure Properties (2)

The argumentation to show that L is not in a class C goes then as follows:
Assume that L is in C. Then (supposing C is closed under operation f), $L' = f(L)$ is also in C. If we know that L' is not in C, this is a contradiction.
Consequently, L is not in C.

Closure Properties (3)

Example:
To show: the double copy language $L = \{www \mid w \in \{a, b\}^*\}$ is not in *TAL*.
Assume that L is in *TAL*. Then (since *TAL* is closed under intersection with regular languages), the language $L' := L \cap a^*b^*a^*b^*a^*b^* = \{a^n b^m a^n b^m a^n b^m \mid n, m \geq 0\}$ is in *TAL* as well. Contradiction since L' does not satisfy the pumping lemma for *TAL*.
Consequently, L is not in *TAL*.

Part II

TAG Parsing

Outline: TAG Parsing

4 Parsing Basics
- Parsing and Recognition
- Items and Deduction-Based Parsing
- CYK and Earley

5 Earley-Style Parsing for TAG
- Preliminaries
- Items and Inference Rules
- From Recognition to Parsing

6 Summary
- Mild Context-Sensitivity
- Parsing

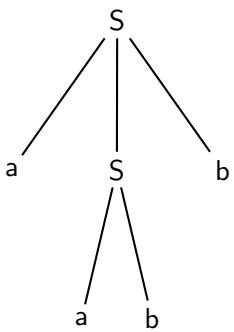
Recognition and parsing

What do we want to use a grammar for? We are interested in knowing

- if a certain word/sentence is licenced by the grammar (**recognition**)
- the structure(s) that a grammar assigns to a grammatical word/sentence (**parsing**)

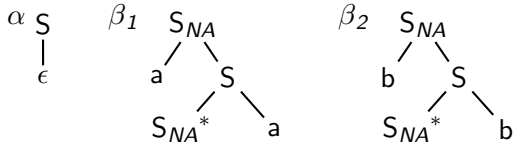
Context-free grammar

- CFG G with two rules: $S \rightarrow aSb$, $S \rightarrow ab$.
- Recognize input *aabb*: yes!
- Parse input *aabb*:

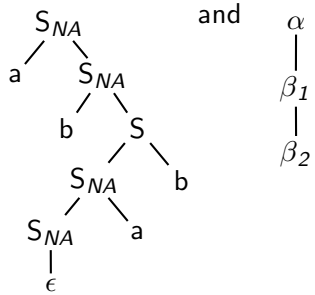


Tree-adjoining grammar

- TAG with three trees:



- Recognize input *abab*: yes!
- Parse input *abab*:



Characterization of Parsing (CFG case)

Given a grammar G, we want to check the grammaticality of a certain input w and find the corresponding structure.
Very informal description:

- **Initialize**: Start with trees related to terminal symbols (*bottom-up*) or related to the root symbol (*top-down*)
- **Parse**: Successively combine trees to bigger trees according to rewriting rules
- **Goal**: Stop when we have a tree with root node labeled with goal label ("S") and yield exactly w

How do we do this?

We fill a chart with all possible constituents and check if it contains the goal tree. For this, the CYK-algorithm for CFG (Chomsky Normal Form, CNF) can be used:

for each `position` p_0
  $C[p_0, p_0+1] := \{A \in N | A \rightarrow w_{p_0+1} \in P\}$
for each `position` p_0
  for each `position` p_1
    for each `position` p_2
      $C[p_0, p_2] := C[p_0, p_2] \cup \{A \in N | A \rightarrow BC \in P$
      $\wedge B \in C[p_0, p_1] \wedge C \in C[p_1, p_2]\}$
return `true` **if** $S \in C[0, n]$.

The algorithm proceeds *bottom-up*.

CKY example

	0	1	2	3	4	5	6
7	S	VP	NP		PP	NP	N
6						Det	
5					P		
4	S	VP	NP	N			
3			Det				
2		V					
1	NP						
	I	saw	a	man	with	a	telescope

Towards parsing schemata (1)

- Problem: The parsing strategy (i.e. the strategy of getting the final parse tree) is hidden in a bunch of control structures (loops, chart)
- These are implementation details the parsing strategy does not depend on.

Better: **Parsing schemata**!

Towards parsing schemata (2)

Parsing schemata allow for abstract specification of

- parsing initialization
- parsing process (producing partial results)
- the parsing goal

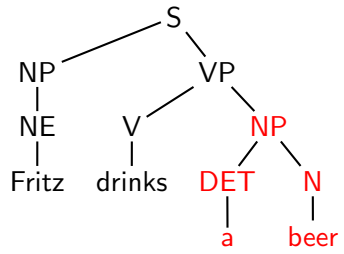
omitting implementation details.

We describe now the CYK algorithm for CFG G (CNF) as a parsing schema. w is the input word, w_i a position i on w, $n = |w|, 1 \leq i \leq n$

We need **Items** and **deduction rules**.

CYK: Parse trees/Items

- We operate with **parse trees** (our partial results!). A parse tree can be characterized by a its root A and the span on w that corresponds to its yield.



- As **items**:
  - NP spanning $w_3 \ldots w_4$: **[NP,2,4]**
  - Root S spanning $w_1 \ldots w_4$: **[S,0,4]**

CYK: Items, general form

In the context of the CYK algorithm, the general form of an item is

$$[A, i, j]$$

while $A \in N$, i, j positions on w.

CYK: Deduction rules, general form

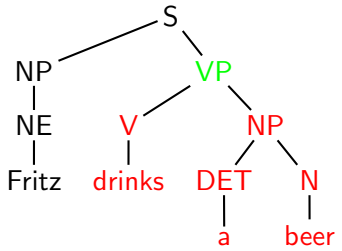
As their name suggests, deduction rules are used to deduce new items. Their general form is:

$$\frac{Antecedent}{Consequent} Sideconditions$$

meaning (in the context of parsing) that given the items in the antecedent and with the side conditions fullfilled, we can introduce the items in the consequent.

CYK: Deduction rules

- Creation of new parse trees by combination of items. Grammar rules determine legal combinations.



- Combine V and NP to VP if both are adjacent and G contains some rule $VP \rightarrow V\ NP$
- Deduction rule for this operation:

$$\frac{[B, i, j][C, j, k]}{[A, i, k]} A \rightarrow BC$$

CYK: Initialization and Goal

- Initialize parsing with special rule:

$$\frac{}{[A, i, i+1]} A \rightarrow w_i + 1$$

- Stop deduction when goal item has been deduced:

$$[S, 0, n]$$

Parsing schemata summary

Parsing schemata offer a concise way of specifying parsing algorithm without having to care about implementation details. We can choose:

- Parse direction: right-to-left, left-to-right, bidirectional
- Parse strategy: bottom-up, top-down, ...
- Item processing order (queue scheduling)
- ...

Idea of Earley Parsing

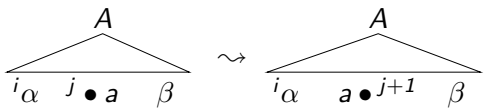
- CYK algorithm creates lots of items which are not part of the final solution: Inefficient!
- Earley idea: Guide parsing by marking the current position in a production and letting subsequent operations depend on it
- Introduce *dotted productions* of the form $A \rightarrow \alpha \bullet \beta$
  - $\alpha, \beta \in (N \cup T)^*$
  - β empty: *completed* item, otherwise *active* item
  - $\bullet$ is a position marker
  - α and everything below has already been recognized
  - β and the part below will be investigated next
- New item form: $[A \rightarrow \alpha \bullet \beta, i, j]$ where α spans $w_{i+1} \ldots w_j$

Earley Parsing: Scan

- First operation: **Scan**
- Move dot if the symbol following the dot is a terminal which matches the terminal in w following α
- Deduction rule:

$$\frac{[A \rightarrow \alpha \bullet a\beta, i, j]}{[A \rightarrow \alpha a \bullet \beta, i, j+1]} w_{j+1} = a$$

- Visualization:



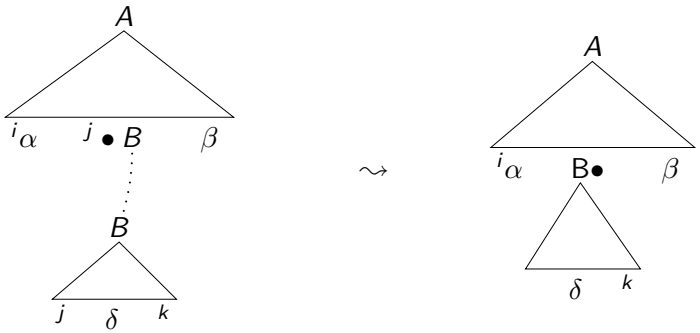
given that $w_{j+1} = a$

Earley Parsing: Complete

- Second operation: **Complete**
- Corresponds roughly to the CYK deduction rule.
- Move dot if
  - the symbol right of the dot is some nonterminal B
  - some completed item exists describing a tree rooted with B and covering a string adjacent to α.
- Deduction rule:

$$\frac{[A \rightarrow \alpha \bullet B\beta, i, j][B \rightarrow \delta \bullet, j, k]}{[A \rightarrow \alpha B \bullet \beta, i, k]}$$

Earley Parsing: Complete (Visualization)

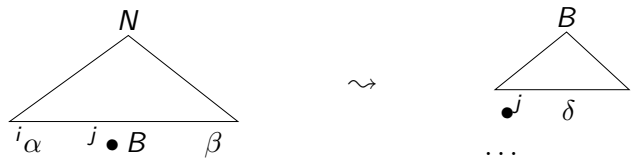


Earley Parsing: Predict

- Third operation: **Predict**
- If the dot is left of some nonterminal B, introduce new items describing trees rooted with B
- This guides parsing: We only introduce items which *could* be part of the solution
- Deduction rule:

$$\frac{[A \rightarrow \alpha \bullet B\beta, i, j]}{[B \rightarrow \bullet\delta, j, j]} B \rightarrow \delta$$

Earley Parsing: Predict (Visualization)



TAG adjunction

- Transfer Earley approach to TAG parsing
- Earley Parsing: Left-to-right scanning of the string, using predictions to restrict hypothesis space
- TAG auxilary trees can contribute **two** unconnected substrings in the final string: Adjunction splits the string it contributes at the footnode
- Problem: How do we recognize adjunction?

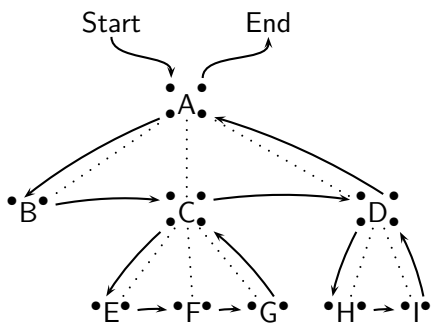
Tree traversal

To enable for left-to-right scanning of the input string while allowing for recognition of adjunction, we introduce a **tree traversal**.

- The current position is marked with a dot
- A *dotted tree* has exactly one dotted node
- The dot can have exactly four positions with respect to the node: left above, left below, right above, right below

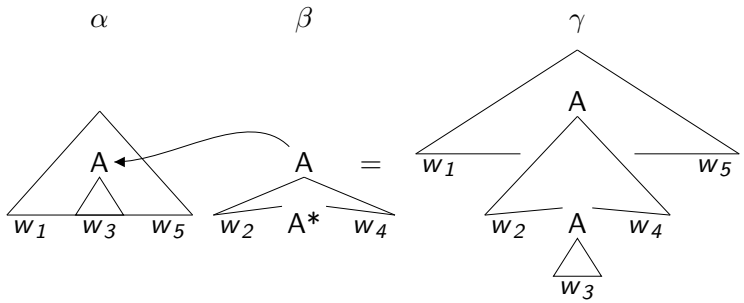
Tree traversal: example

This is how we traverse a tree (...how we move the dot):



An adjunction operation

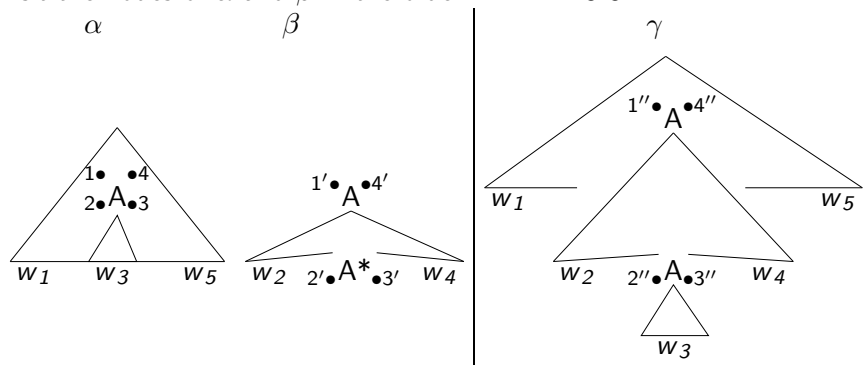
We adjoin β into α with γ as the result.



Our tree traversal can be used to see $w_1 w_2 w_3 w_4 w_5$ in exactly this order, i.e. it can be used to recognize the adjunction:

Tree traversal and TAG adjunction

We should visit the nodes of γ in the following order: 1" 2" 3" 4". We don't build γ, but the traversal can be used to ensure that we visit the nodes of α and β in the order 1 1' 2' 2 3 3' 4' 4.



Items for parsing

What kind of information do we need in an item s?

$$s = [\alpha, dot, pos, i, j, k, l, sat?]$$

where

- $\alpha \in I \cup A$ is a (dotted) tree, *dot* and *pos* the address and location of the dot
- i, j, k, l are indices on the input string, where $i, l \in \{0, \ldots, n\}$. $j, k \in \{0, \ldots, n\}$ or both unbound
- *sat*? is a flag. It controls (prevents) multiple adjunctions at a single node ($sat? = 1$ means node blocked for adjunction)

What do the items mean?

- $[\alpha, dot, la, i, j, k, l, nil]$: In α part left of dotted node ranges from i to l. If α is an auxiliary tree, part below foot node ranges from j to k
- $[\alpha, dot, lb, i, -, -, i, nil]$: In α part below dotted node starts at position i
- $[\alpha, dot, rb, i, j, k, l, nil]$: In α part below dotted node ranges from i to l. If α is an auxiliary tree, part below foot node ranges from j to k
- $[\alpha, dot, ra, i, j, k, l, sat?]$: In α part left and below dotted node ranges from i to l. If α is an auxiliary tree, part below foot node ranges from j to k. If $sat = nil$, nothing was adjoined to dotted node, $sat = 1$ means that adjunction took place

Inference rules

- The TAG Earley parser consists of four different operations
- The first three are the usual ones: *Scan*, *Predict* and *Complete*
- The new fourth operation (*Adjoin*) handles adjunction

Some more preliminaries:

- Function $C(\alpha, \eta)$ with $\alpha \in I$ and η node in α returns the trees which can be adjoined at η.
- Boolean function $O(\alpha, \eta)$ returns if adjunction at η is obligatory.

Inference rules: Scan

The **Scan** operation scans terminals (advances on the input string). It consists of two steps:

- *ScanTerm*: If dot is left above w_{l+1}, move right and increase recognized span:

$$\frac{[\alpha, dot, la, i, j, k, l, nil]}{[\alpha, dot, ra, i, j, k, l+1, nil]} \; \alpha(dot) = w_{l+1}$$

- *Scan-ϵ*: If dot is left above an empty symbol ϵ, move right:

$$\frac{[\alpha, dot, la, i, j, k, l, nil]}{[\alpha, dot, ra, i, j, k, l, nil]} \; \alpha(dot) = \epsilon$$

Inference rules: Predict (1)

The **Predict** operation proposes new items according to the already seen left context. It consists of three steps:

- *PredictAdjoinable*: If dot in α is left above some nonterminal, predict all adjoinable auxiliary trees:

$$\frac{[\alpha, dot, la, i, j, k, l, nil]}{[\beta, 0, la, l, -, -, l, nil]} \; \beta \in C(\alpha, dot)$$

  $\Rightarrow$ **predict adjunction**
- *PredictNoAdj*: If dot in α is left above some nonterminal (and no OA constraint is present), then move down without adjunction:

$$\frac{[\alpha, dot, la, i, j, k, l, nil]}{[\alpha, dot, lb, l, -, -, l, nil]} \; O(\alpha, dot) = 0$$

  $\Rightarrow$ **predict no adjunction**

Inference rules: Predict (2)

- *PredictAdjoined*: Auxiliary tree β, dot left below the foot node. Predict all trees where β could have been adjoined.

$$\frac{[\beta, dot, lb, l, -, -, l, nil]}{[\delta, dot', lb, l, -, -, l, nil]} \; dot = Foot(\beta), \beta \in C(\delta, dot')$$

$\Rightarrow$ **predict adjunction site**

Inference rules: Complete (1)

The **Complete** operation combines present items into new ones that together span a bigger portion of the input string. The algorithm uses two (resp. three) operations:

- *Complete*: Identify tree below dot in α as the tree below the footnode of β

$$\frac{[\alpha, dot, rb, i, j, k, l, nil], \quad [\beta, dot', lb, i, -, -, i, nil]}{[\beta, dot', rb, i, i, l, l, nil]} \quad dot' = Foot(\beta), \beta \in C(\alpha, dot)$$

Inference rules: Complete (2)

- The second *Complete* operation combines two partially recognized trees to build a longer string. There are two cases:

$$\frac{[\beta, dot, rb, i, j, k, l, sat?], \quad [\beta, dot, la, h, -, -, i, nil]}{[\beta, dot, ra, h, j, k, l, nil]} \quad \beta(dot) \in N$$

$$\frac{[\beta, dot, rb, i, -, -, l, sat?], \quad [\beta, dot, la, h, j, k, i, nil]}{[\beta, dot, ra, h, j, k, l, nil]} \quad \beta(dot) \in N$$

Inference rules: Adjoin

Adjoin is a single-step operation that handles adjunction

- *Adjoin*: Adjoin an auxiliary tree β at the dotted node in tree α, set flag indicating that (α, *dot*) is now blocked for adjunction

$$\frac{[\beta, 0, ra, i, j, k, l, nil], \quad [\alpha, dot, rb, j, p, q, k, nil]}{[\alpha, dot, rb, i, p, q, l, 1]} \quad \beta \in C(\alpha, dot)$$

Inference rules: Moving the dot

MoveDot: When dot in position *ra*, move dot to right sister node at position *la*, or to parent node at position *rb* if last daugther

- *MoveRight*

$$\frac{[\beta, p, ra, i, j, k, l, sat?]}{[\beta, p+1, la, i, j, k, l, sat?]} \quad \beta(p+1) \text{ is defined}$$

- *MoveUp*

$$\frac{[\beta, p \cdot m, ra, i, j, k, l, sat?]}{[\beta, p, rb, i, j, k, l, sat?]} \quad \beta(p \cdot m + 1) \text{ is not defined}$$

- *MoveDown*: Move from parent to leftmost daughter

$$\frac{[\beta, p, lb, i, j, k, l, sat?]}{[\beta, p \cdot 1, la, i, j, k, l, sat?]}$$

Initializing and Goal item

- Initialize:

$$\frac{}{[\alpha, 0, la, 0, -, -, 0, nil]}, \alpha \in I$$

- Goal item: $[\alpha, 0, ra, 0, -, -, n, nil]$

Parsing?

- We can turn our recognizer into a parser
- To achieve that, we store each item with a set of pairs of other items from which it can be inferred
- If a certain item has been inferred from several different pairs of items, we have a case of ambiguity
- A parse forest can by constructed by tracing back the antecedents starting with the goal item

Summary: Mild Context-Sensitivity

We have seen that

1. TAG generate limited cross-serial dependencies: There is a $n \geq 2$ such that the formalism can generate all string languages $\{w^k \mid w \in T^*\}$ up to $k = n$. (For TAG, $n = 2$.)
2. TAG is polynomially parsable.
3. The class *TAL* has the *constant growth property*.

Formalisms satisfying these properties are called *mildly context-sensitive*.

Summary: Parsing

- We have seen an Earley-type parsing algorithm for Tree Adjoining Grammar
- The parser has *Complete*, *Scan* and *Predict* operations plus an *Adjunction* operation
- The algorithm has an upper time bound of $O(n^6)$