Lecture notes by Edward Loper

Course: CIS 630 (NLP Seminar: Structural Representations)
Professor: Joshi
Institution: University of Pennsylvania

Monday, January 15, 2001

0.1 Representationally Oriented Grammars

(or Grammars for Analysis, Grammars as Constraint Satisfaction)

Look at grammar as a set of constraints that the sentence must satisfy. Sentence is associated with a description:

S → D

Grammar's job is to decide whether a description D is consistant.

Just tells you what representations are licensed, doesn't say how to get them.

0.2 Derivationally Oriented Grammars

(or grammars for generation) Say how to derive a grammar. How to construct a derivation D. Generative system.

Monday, January 22, 2001

Define finite automata on tree: type_node x (state_child1, state_child2, ...) → state_node
e.g. The x () → q1 table x () → q2 NP x (q1, q2) → q3
At the top, check if you're in the set of accepting states.
"recognizable set" of trees is exactly those trees that are accepted by FSA on trees.

1 to do

- send email re what i want to work on

nested dependencies vs. cross dependencies: CFGs can only give nested dependancies.

see joshi re lexical semantic info after class.. take initiative? ;)

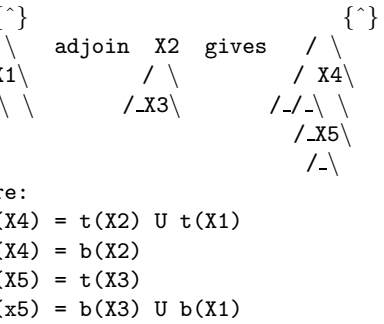
2 Unification

Implementing constraints on substitution and adjoining (and in particular adjoining)

3 types of constraints:

- selective adjoining – Feature structures implicitly specify constraints.
- null adjoining
- obligatory adjoining – at that node, at least one adjunction must take place

Adjoining changes already-built structure. It's a higher order operation than substitution, and a higher order abstraction..



Where "U" is unification..

Substitution is a special case, where X is a leaf, and it has no bottom features.

2.1 LTAG

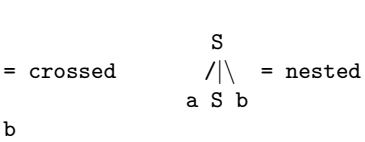
```
G = (I, A)
I = initial trees
A = adjunction treess
```

We don't give rewrite rules because adjunction and substitution are language independant.

```
T(G) = trees
L(G) = strings
TAL = ∪_G L(G)
```

Theorems:

1. TAL is more powerful than CFGs (CFL $\subset$ TAL, proper subset) Note: You can get crossed dependancies (2 nested dependancies sharing elements gives crossed dependancies)
2. even when TAG's surface form is CFG, SD's of CFGs $\subset$ SD's of TAGs. (SD=structural description)



proper analysis?

Representation should make it clear what the constituants are. What about discontinuous constituants, etc.?

we have to define what we mean by constituants..

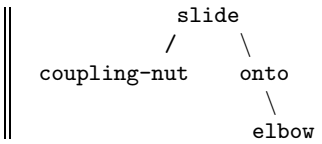
Representations should also make the dependancies clear. what are dependencies? what are dependencies between constituants?

3 Generation talk

3.1 Formal structure for generation

- Syntax
- Semantics
- Links to context (& goals of conversation)

Consider the LTAG derivation tree:



define:

- new
- assertion: move(e, h, n, p) is next(e)
- shared
- presup
- pragmatics

Thursday, March 1, 2001

4 Dependancy and Locality in TAG

What is dependancy?

- thematic dependancy:
  - relationship between predicates and arguments
  - locality: syntacitically realized within some constrained structural domain.
  - always local, but sometimes operating on traces, etc.
  - structural dependancy
  - relationship between 2 elements in a structure
  - e.g., moved element and trace (coindexing)
  - subject to locality constraints

Why?

- accounts for syntax data..

What *is* the local structural domain?

Primary argument of Frank: a privledged structural domain to express locality dependancy relations can be defined.

Terms:

- Basis: atomic units out of which stuctures are built
- Structural Composition: closure of basis over composition rules
- Transformation: modify existing rep. transformations create structural dependancies!

Kernel structures from chomsky 55: basically simple active sentences. can they be the domain of locality? but what about transformations?? if we interlevel transofmations and compositions, things get arbitrarily far apart..

In TAG, all transformations take place prior to formation of elementary trees:

```
basis --(move + merge)⟶ elementary tree
elementary tree --(adjoin + subst)⟶ sentences
```

5 Raising, Superraising, There-Insertion

Raising in GB is defined by a transformational account. Raise an element to a site higher in the tree. Raising attempts to preserve some sense of locality – trace..

In TAG, there's no transformation. Raising is defined by adjoining. Eg., define "seems" as an adjoining node that inserts between "John" and "to like broccoli." Locality is preserved because they come from the same fundamental tree.

Recursion in GB: successive cyclic movement

Recursion in TAG: multiple adjoins.

5.1 Super-Raising

- John_i seems [IP t_i to be likely [IP t_i to eat broccoli]]
- *John_i seems [IP it is likely [IP t_i to eat broccoli]]

Why is the second one bad? In GB:

- representational constraint
- derivational constraint

Either way, we must give an explicit constraint.

But in TAG, it comes for "free."

How to deal with "it is likely"?

No super-raising: you can't combine I'..IP and IP..I' to get I'..I'

6 Bob Frank

- What makes an elementary tree valid for a language?
- What makes a derived tree an acceptable sentences?
- Not all elementary trees are acceptable sentences.

Consider:

- There [seems] to be a VP in the hospital
- There [seems] a VP to be in the hospital

Second one is invalid because we don't have the elementary tree:

- There a VP to be in the hospital

But what if we interpret it as:

- [There seems] a VP to be in the hospital

So why can't "there seems" be an elementary tree? maybe because seems wants to take predicates, and "there" isn't an argument..

But what about [it seems]? If our elementary tree for it is:

```
[TP it [T' [T ...] [VP [V seems] [CP ...]
```

And what about "it is raining"?

And what about sentences (in other languages) like:

- it was danced by John.

Does EPP hold on elementary trees? Yes. Otherwise, we'd never generate the elementary trees for "it is raining." But on the other hand we have elementary trees that don't satisfy the EPP.

So why are "there" and "it" different? I.e., why do we get:

```
[TP it [T' [T ...] [VP [V seems] [CP ...]
```

but not:

```
[TP there [T' [T ...] [VP [V seems] [CP ...]
```

If we assume that subjects always begin in VP, we have to explain why/how they get to spec/TP.

Define a lexical array = a list of lexical items with selectional features. Selectional features require that an item merge with certain types of object. Then use Merge & Move to construct your elementary trees from small LAs. LAs must have at most one semantically contentful element..

When creating elementary trees, keep going until you've dealt with as many uninterpretable features as you can.. E.g., assume T has [+EPP] feature. So:

```
[VP DP [V' [V expected] TP]] →
  [TP DP [VP t [V' [V expected] TP]]]
```

But:

```
[VP [V seems] TP] →
  [TP [VP [V seems] TP]]
```

ϕ features are agreement features. But they're not selectional.

"It" agrees with ϕ, but "there" doesn't. Which means that we can get:

```
[TP it [T [VP [V seems] TP]]]
```

because "it" satisfies both the EPP and the ϕ features of T. But we can't get the same thing with "there" because there doesn't satisfy the ϕ features:

```
* [TP there [T [VP [V seems] TP]]]
```

c.f.:

```
it seems that A and B (note: seems is singular)
there are A and B (note: are is plural)
```

Claim: after we've generated our elementary trees, the unchecked features play the role of placing restrictions on what adjunctions are allowed.

A can only adjoin to B if doing so satisfies some of B features.

comments to: rfrank@jhu.edu

7 Reduced Constructions

7.1 Scrambling and Tag

```
that no one dared [ the bike to repair]
that the bike no one dared [ to repair]
```

This movement is unbounded. Scrambling not allowed by all verbs.

7.2 Clitic Climbing

Clitic can "climb" to higher clauses, for some verbs.

7.3 The problem

moved constituant ends up in the *middle* of the upper clause..

For:

```
John seems to like the pizza
```

We can just adjoin in "seems" to the E.T.:

```
NP to like NP.
```

But where does "does" belong in:

```
Does John seem to like the pizza?
```

? X-tag makes "does" its own tree. But it seems like "does" is on C, so we might want:

```
[Cbar [C does] [IP [Ibar [I seem] ...]]]
```

But then what adjoins into what? One option is to use multicomponent tree-local tag, and do:

```
[Cbar [C does] ...]
```

and

```
[Ibar [I seem] ...]
```

But you can get unbounded raising.. So the components can get arbitrarily far apart..

8 More fun with C/G

8.1 Reconstructing C/G in the form of LTAG..

CG derivation trees parallel LTAG elementary trees.

4 rules:

```
Function application:
   (S/NP) NP → S
   NP (NP\ S) NP → S
Function composition:
   X/Y Y/Z → X/Z
   X\ Y Y\ Z → X\ Z
```

You can assume [A], and derive [B]. This lets us prove that A→ B. This is "withdrawing the assumption" or "discharging the assumption"

```
 [A]         ; make assumption
  ⋮
  B
-----        ; withdraw assumption
 A→ B
```

```
CFG    ⟶  LTAG
 ↓          ↓
CG(AB) →  CG(PPT)

Where:
   → is strong-lex EDL, FRD
   ↓ is weak equivalance
   CG(AB) = standard, vanilla CG
   CG(PPT) = C/G with partial proof tree
```

This is similar to what Bob Frank was doing with LTAG and minimalism, but for CG.

Instead of assinging likes the type "(NP\ S)/S", assign it a partial proof tree:

```
           likes
             |
[NP]     (NP\ S)/S)    [NP]
         ------------------
             NP\ S
--------------------------
           S
```

Thus, these PPT types have assumptions.

Normally, the only way to satisfy an assumption is to withdraw it. But we will introduce new ways of satisfying an assumption?

Construct a finite collection of partial proof trees, where each PPT is a syntactic type associated with a lexical item. These are analagous to elementary trees. Then introduce a composition method. These must be inference rules..

8.2 Inference rule: linking

B(PPT) = the set of basic partial proof trees. How do we construct them?

- Unfold arguments of types by introducing assumptions.
- No unfolding past an argument that's not an argument of the lexical item.. e.g., adverb is VP→ VP, i.e., its type is (NP\ S)\(NP\ S). But how do we keep from unfolding the VP? Mark a node that stops unfolding with an asterisk: (NP\ S)\(NP*\ S).
- If a trace assumption is introduced while unfolding then it must be locally discharged, i.e., within the basic PPT which is being introduced.
- While unfolding we can interpolate, say, from X to Y where X is a conclusion node and Y is an assumption node.

8.3 Stretching

```
Y = u v w, with X = the single conclusion of v
Then we can say Y = u [X] w; X → v
```

e.g., stretched likes:

```
            likes
              |
[NP]     (NP\ S)/S)    [NP]
         ------------------
             NP\ S
               :
               :
            [NP\ S]
---------------------------
            S
```

Then we can splice in "passionately"

```
          passionately
               |
 [NP\ S] (NP\ S)\(NP*\ S)
------------------------
       NP\*S
```

We are still just using linking here when we combine..

Stretching is used during composition.

8.4 Traces

Introduce a special assumption, which we'll call a trace assumption.. and then discharge it on the other side. You must discharge within one elementary tree.

```
                  likes    e
                    |      |
 [NP]  [NP]   (NP\ S)/NP  [NP]
              ---------------
                  NP\ S
       ----------------------
                S
            -------- # Discharge assumption
              NP\ S
 -------------------
         S
```

We must disallow the possibility of doing discharges outside of elementary trees, because then we lose locality.. we want to keep dependancies in elementary trees.

Use a permutation operation to allow assumption and discharge to occur on different sides?

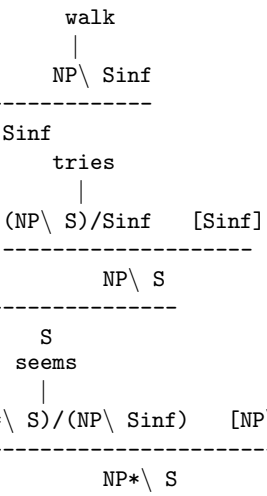
In normal CG, you have to introduce assumptions at the periphery. So introducing e in a sentence like "who John saw e yesterday" would be difficult. But since we can stretch, we can stretch "who John saw e" and

splice in "yesterday"..

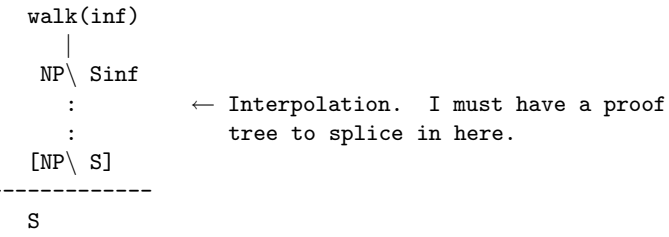
8.5 Interpolate

Interpolation is basically introducing a gap in a PPT that must be satisfied at the time PPTs are put together (i.e., while constructing full proof trees for sentences).

How to do something like "John seems to be happy"? We want a "John to be happy" tree with a spliced in "seems to".. but how do we rule out "John to be happy"? (In LTAG, we used features) Consider "John tries to walk"



So we need a new tree for walk.



You can only interpolate while constructing PPTs, not while using them to construct sentences. (c.f., trace assumptions). Then at run-time, we can splice things in (e.g., seems).. This is equivalant to forced adjunction in LTAG..

How does interpolation relate to multicomponent LTAG?

Question: discharging and introducing on different sides??

9 Synchronous Grammars

- Grammars generate languages (sets of strings)
- Synchronous grammars generate string relations (sets of pairs of strings).

Useful for:

- transalation
- interpretation (to internal language)

9.1 Synchronous CFG

(aka Syntax-directed translation schemata)

Set of pairs of rules. E.g.:

```
<S → X Y, S → A B>
<X → x Z, B → b C>
<Y → y, A → a>
```

In additon, a system of coindexation..

```
<S → X1 Y2, S → A2 B1>
<X → x Z1, B → b C1>
<Y → y, A → a>
```

Indexes say that generated nonterminals are "linked."

Start with a pair of "top" nonterminals, and rewrite them in pairs, using paired rules:

```
<S, S> ⇒ <X1 Y2, A2 B1>
```

If you rewrite a nonterminal with a given index on the left, you must rewrite the nonterminal with the same index on the right. In this case, if we rewrite X1, we must also rewrite B1.

```
<X1 Y2, A2 B1> ⇒ <x Z3 Y2, A2 b C3>
<x Z3 Y2, A2 b C3> ⇒ <x z Y2, A2 b c>
<x z Y2, A2 b c> ⇒ <x z y, a b c>
```

In this case, this is the only string generated by the grammar:

```
<x y z, a b c>
```

Derivation trees:

```
< (S (X x (Z z)) (Y y)), (S (A a) (B b (C c)))>
```

Property of result: Structure of the tree doesn't change, except that sisters may be reversed and nodes may be renamed. This is (almost?) always a property of synchronous grammars.

Indeces are what give us this isomorphism.

<an bn c*, a*b*c*, a* bn cn>

Synchronous Grammar α:

```
<AⁿBⁿC*, A*B*C*> → <AⁿBⁿ C*, A*B*, C*>
<AⁿBⁿ, A*B*> → <a AⁿBⁿ b, A*B*>
<AⁿBⁿ, A*B*> → <ϵ, ϵ>
<C*, C*> → <c C*, c C*>
<C*, C*> → <ϵ, ϵ>
```

Synchronous Grammar β:

$<$A*B*C*, A*B^{n}C^n$>$ $\rightarrow$ $<$A* B*C*, A* B^{n}C^n$>$
$<$B*C*, B^{n}C^n$>$ $\rightarrow$ $<$b B*C* c, b B*C* c$>$
$<$B*C*, B^{n}C^n$>$ $\rightarrow$ $<\epsilon, \epsilon>$
$<$A*, A*$>$ $\rightarrow$ $<$a A*, a A*$>$
$<$A*, A*$>$ $\rightarrow$ $<\epsilon, \epsilon>$

$\alpha \circ \beta$ = a^{nn}c^n