Where we are

Yesterday:
- introducing XTAG analyses of extraction
- different templates for different constructions and valency frames
- rich feature sets in the nodes (for constraining substitution and adjunction)

Today:
- How to factorize the set of templates?
  - express lexical generalizations, e.g. active-passive diathesis
  - define tree families
- How to turn this into an electronic resource?
- How to plug it into a lexicon and use it?

Outline

1. What is grammar implementation?
2. Two ways of tree template implementation
   - Metarules
   - Metagrammars
3. eXtensible Metagrammar (XMG)
4. LTAG as Construction Grammar
5. Lexicon and parser
6. Summary

As is frequently pointed out but cannot be overemphasized, an important goal of formalization in linguistics is to enable subsequent researchers to see the defects of an analysis as clearly as its merits; only then can progress be made efficiently. (Dowty 1979: 322)
Two kinds of grammar implementation

Grammar/linguistic theory → Specifications in accordance with a grammar formalism → Evaluation of the theory → Grammar resource → Computational application

What kind of grammar resource?

Tree template: S → NP, VP → V → NP

Lexical insertion: Anchor → Repairs

The implementation task for LTAG

General task
Implement a large-coverage LTAG, i.e. based on the XTAG grammar!

Subtasks:
1. Generate unlexicalized trees (= tree templates)!
2. Generate a database of lexical anchors (= the lexicon)!
3. Connect the tree templates with the lexicon (= lexical insertion)!

Two ways of grammar implementation with TAG

Two existing toolkits:

**XTAG tools**
1. Implementation tools
   ⇒ metarule approach
2. Editor/viewer for MorphDB and SynDB
3. Parser

**XMG + lexConverter + TuLiPA**
1. XMG: eXtensible MetaGrammar
   ⇒ metagrammar approach
2. lexConverter (LEX2ALL)
3. TuLiPA: Tübingen Linguistic Parsing Architecture
Outline

1. What is grammar implementation?
2. Two ways of tree template implementation
   - Metarules
   - Metagrammars
3. Extensible Metagrammar (XMG)
4. LTAG as Construction Grammar
5. Lexicon and parser
6. Summary

The situation

<table>
<thead>
<tr>
<th>12 templates for intransitive verbs</th>
<th>39 tree templates for transitive verbs</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>NP</td>
<td>NP</td>
</tr>
<tr>
<td>VP</td>
<td>VP</td>
</tr>
<tr>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>NP</td>
<td>NP</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>NP</td>
<td>NP</td>
</tr>
<tr>
<td>VP</td>
<td>VP</td>
</tr>
<tr>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
</tr>
</tbody>
</table>

Basically, XTAG defines a set of 1008 unrelated tree templates.

Metarules for LTAG

Idea from GPSG\(^{[12]}\), later applied to XTAG\(^{[2,3,21]}\)

```
```

<table>
<thead>
<tr>
<th>extraction:</th>
<th>passivization:</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>VP</td>
</tr>
<tr>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>S</td>
<td>NP</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>S</td>
<td>VP</td>
</tr>
<tr>
<td>NP</td>
<td>VP</td>
</tr>
<tr>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

Balogh & Lichte (Düsseldorf)
Metarules for LTAG: Problems

Metarules are very powerful:
- deletion, copying, recursive application, metavariables over trees
- order sensitive
- in the unrestricted case: undecidable\(^2\)

Restrictions (GPSG)\(^{2,2}\):
- finite closure: apply every metarule at most once!
  - still NP-complete
- biclosure: apply at most two metarules in a row!
  - insufficient for LTAG metarules\(^2\)
- explicit rule ordering (by means of finite state automata)\(^{21}\)

Metagrammars for LTAG: Tree descriptions

\( \mathcal{L}_D \): Description language for trees

Let \( n_2 \) and \( n_2 \) be node variables:

\[
\text{Description} := \left( \begin{array}{c}
  n_1 \rightarrow n_2 \\
  n_1 < n_2 \\
  n_1 = n_2
\end{array} \right) \quad \text{and} \quad \text{Description} \wedge \text{Description}
\]

Example:

\[ S \quad \text{corresponds to} \quad n_S \rightarrow n_{NP} \wedge \]
\[ NP \quad \text{corresponds to} \quad n_{NP} < n_{VP} \]
\[ VP \]
\[ S \quad \text{corresponds to} \quad n_S \rightarrow n_{NP1} \wedge \]
\[ NP \quad \text{corresponds to} \quad n_{NP1} < n_{NP2} \wedge \]
\[ + \]
Metagrammars for LTAG: Example

Minimal model of tree descriptions
You may add edges but not nodes!

S
NP
S

S
NP
VP

S
NP
VP

NP
VP
VP

NP
VP

S
NP
VP

S
NP
V\circ
NP

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

Metagrammars for LTAG: Example

Minimal model of tree descriptions
You may add edges but not nodes!

S
NP
S

S
NP
VP

S
NP
VP

NP
VP
VP

NP
VP

S
NP
VP

S
NP
V\circ
NP

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ

S
NP
VP

S
NP
VP

S
NP
V\circ
Metagrammars for LTAG: Properties

- no deletion, no copying, no recursion
- declarative, order insensitive
- The number of minimal models is finite.
- BUT: the number of minimal models can grow exponentially ($O(n!)$) in terms of the number of described nodes.

Does it suffice? How to express passivization?

Metagrammars for LTAG: Passivization

Metagrammars for LTAG: Classes

Tree descriptions are bundled into so-called classes:

- **$L_C$: Description language for the combination of tree descriptions**
  
  $$
  \text{Class} := \text{Name} : \text{Content} \\
  \text{Content} := (\text{Description} \mid \text{Name} \mid \text{Content} \lor \text{Content} \mid \text{Content} \land \text{Content}) 
  $$

Upon instantiating/using a class:

- Node variables are replaced by fresh ones.
- Node variables are known to the instantiating class.
- The class name is replaced by the content in the instantiating class.

$\Rightarrow$ Classes can be reused!
There are very many possible class hierarchies ...

Subject WhNP+EmptyWord Object
BaseSubject WhSubject WhObject BaseObject
VerbProjection

Balogh & Lichte (Düsseldorf)
Metagrammar for LTAG: Class hierarchies

… but not everything is possible:

alphaW0nx0Vnx1 alphanx0Vnx1

alphaW0nx0Vnx1 alphanx1Vbynx0

alphaW1nx1Vbynx0 Tnx0nx1

Outline

1. What is grammar implementation?
2. Two ways of tree template implementation
   - Metarules
   - Metagrammars
3. eXtensible Metagrammar (XMG)
4. LTAG as Construction Grammar
5. Lexicon and parser
6. Summary

eXtensible Metagrammar (XMG): Background

- developed at LORIA, Nancy, and LiFO, Orléans.
- written in Oz/Mozart, YAP and Python (as of XMG2)
- available at http://sourcesup.cru.fr/xmg

Why "eXtensible"?
- highly modularized
- dimensions with dedicated description languages and compilers
  (<syn>, <sem>, <frame>, <morph>, …)
- interface through shared variables

Some existing implementations using XMG:
- French: FrenchTAG
- English: XTAG with XMG
- German: GerTT

eXtensible Metagrammar (XMG): Example

```
class Subject export ?S declare ??S ?NP ?VP
{ <syn>{
    node ??S [cat=s]
    node ?NP (mark=subst) [cat=np]
    node ?VP [cat=vp]
}
```

```
eXtensible Metagrammar (XMG): Example

```
class Subject
export ?S
declare ?S ?NP ?VP
{
  <syn>{
    node ?S [cat=s];
    node ?NP (mark=subst) [cat=np];
    node ?VP [cat=vp];
    ?S -> ?NP;
    ?S -> ?NP;
    ?NP >> ?VP
  }
}
```

eXtensible Metagrammar (XMG): Example with `<frame>`

```
class Subj
...
<syn>{
  node ?S [cat=s];
  node ?SUBJ [cat=np, top=[i=?1]];
  node ?VP [cat=vp,bot=[e=?0]];
  node ?V (mark=anchor)
    [cat=v,top=[e=?0]];
  ?SUBJ >> ?VP
}
<frame>{
  ?0[actor,actor:?1]
  ...
```

Outline

1. What is grammar implementation?
2. Two ways of tree template implementation
   - Metarules
   - Metagrammars
3. eXtensible Metagrammar (XMG)
4. LTAG as Construction Grammar
5. Lexicon and parser
6. Summary
Claim (ongoing work)

LTAG shares central ideas with (some versions of) Construction Grammar (CxG):[13]

1. only surface structure: no transformational or derivational component
2. grammatical constructions: "form-function pairings at varying levels of complexity and abstraction" (e.g. words, phrases)
3. a network of constructions "which nodes are related by inheritance links"

By virtue of adjunction, cases of long-distance dependencies can be immediately captured:

(1) Who does Mary say sometimes walks into the house.

(2) John gives/sends the book to Mary

grammatical constructions

inheritance network of constructions

(From Kallmeyer & Osswald (2013))
LTAG as Construction Grammar: Summary

LTAG incorporates central ideas of CxG:
- only surface structure ✓
- grammatical constructions ✓
- inheritance network of constructions ✓

LTAG differs substantially from other implementations of CxG (e.g. Sign-based CxG\cite{23}, Fluid CxG\cite{24}).
⇒ different empirical predictions or theoretical ramifications?

Outline

1. What is grammar implementation?
2. Two ways of tree template implementation
   - Metarules
   - Metagrammars
3. eXtensible Metagrammar (XMG)
4. LTAG as Construction Grammar
5. Lexicon and parser
6. Summary

Lexicon and parser

Lexicon and parser: A 2-layered lexicon

loves → love → Tnx0Vnx1
“morphological lexicon” “lemma lexicon”

Morphological lexicon
maps an (inflected) token to some base form (= lemma), while preserving morphological information in a feature structure.

<table>
<thead>
<tr>
<th>loves</th>
<th>love</th>
<th>[pos=v; num=sing; pers=3;]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>Peter</td>
<td>[pos=n; num=sing; pers=3; case=nom</td>
</tr>
</tbody>
</table>

Interface with tree templates: Feature unification during lexical insertion
Lexicon and parser: A 2-layered lexicon

"morphological lexicon"  "lemma lexicon"

Lemma lexicon

maps a lemma onto tree tuple families, while also containing selectional restrictions (e.g., case assignment).

ENTRY: love
CAT: v
SEM:
ACC: 1
FAM: Tnx0Vnx1
FILTERS: []
EX:

Interface with tree templates:

EQUATIONS → nodes of tree templates
FILTERS → selection of tree templates

Lexicon and parser: The TuLiPA parser

TuLiPA

- Tübingen Linguistic Parsing Architecture (TuLiPA)
- uses Range Concatenation Grammar (RCG) as a pivot formalism.

Components:

1. TAG-to-RCG converter (on-line)
2. RCG parser → RCG derivation forest → TAG derivation forest
3. Parse viewer (derived tree, derivation tree, dependency view, semantic representation)

Availability of TuLiPA:
written in Java and released under the GNU GPL
(http://sourcesup.cru.fr/tulipa/)

Outline

1. What is grammar implementation?
2. Two ways of tree template implementation
   - Metarules
   - Metagrammars
3. eXtensible Metagrammar (XMG)
4. LTAG as Construction Grammar
5. Lexicon and parser
6. Summary

Summary

- A metagrammar contains descriptions of unanchored elementary trees.
- Metagrammar descriptions are declarative and multidimensional.
- Metagrammar descriptions make up an inheritance hierarchy.
- The metagrammar allows one to express and implement lexical generalizations, e.g., active-passive diathesis.
- TAG + metagrammars implements central ideas of CxG.

Hot topics:

- parsing with metagrammars[7]
- use metagrammars for morphological descriptions[11,20]

Adjacent topics:

- grammar induction from treebanks[5,6,14,26]


