Higher Order Structures in Minimalist Derivations

Greg Kobele
TAG+13
Universität Leipzig
Intro
Grammar formalisms, like programming languages, are useful because

they allow us to factor our explanation of linguistic behaviour into a statement of abstract regularities (the grammar), and a description of how these are computed online (the parser/parser-generator)
Intro

Current MG parsing algorithms

- needlessly explode state space (making beam search implausible)
- are based on (exponentially less succinct) MCFGs
- have only extrema on GLC lattice (inherited from MCFG)
We exploit the structure of MGs to define a MG-specific TD parsing strategy

- structures search space by 'sharing' infinite classes of items
- bringing us closer to LC

This gives a formal (very literal) reconstruction of popular psycholinguistic ideas about the human sentence processing mechanism
MGs
Overview

a formalization of Chomsky’s “minimalist program”

- I think they are an exact formalization
- I am interested in them because they are a bridge between linguistics and computer science
Properties

MGs belong to family of MCS grammar formalisms

- TAG is Monadic CFTG, and MG is (contained in) MRTG
  - Share the regularity of derivation trees
- TALs are all well-nested MCFLs, but MLs are the non-well-nested MCFLs

separation: (Kanazawa & Salvati, 2010)

\[ \{ w \# w \mid w \in L, \ L \text{ is in } CFL - EDT0L \} \]

well-nested MCFLs can have crossing dependencies, but not between syntactically complicated objects
To specify a grammar, we need to specify two things:

1. The **features**
   (which features we will use in our grammar)
2. The **lexicon**
   (which syntactic feature sequences are assigned to which words)
Features

Features come in pairs

- \( =x \) and \( x \)
- \( +y \) and \( -y \)

Like in CG, categories are *structured*

- list of features

tradition calls categories: *feature bundles*
Data structure

Binary branching trees

- internal node labels: > and <
- leaf labels: \((w, \delta)\) and \(t\)

Headed trees

\[
\begin{align*}
\text{head}(>(u,v)) &= \text{head}(v) \\
\text{head}(<(u,v)) &= \text{head}(u) \\
\text{head}(l) &= l
\end{align*}
\]
Merge

= \textcolor{red}{x}.\gamma

x.\delta

\Rightarrow

\gamma \quad \delta

<
Move

\[
\begin{align*}
&\quad +y. \gamma \\
&\quad -y
\end{align*}
\]

\[\Rightarrow\]

\[
\begin{align*}
&\quad \gamma
\end{align*}
\]

SMC
No other possible mover
A working example

<table>
<thead>
<tr>
<th>word</th>
<th>tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>boy</td>
<td>n</td>
</tr>
<tr>
<td>every</td>
<td>=n.d.-k</td>
</tr>
<tr>
<td>laugh</td>
<td>=d.v</td>
</tr>
<tr>
<td>will</td>
<td>=v.+k.s</td>
</tr>
</tbody>
</table>
Representing derivations

1. select every
2. select boy
3. merge 1 and 2 [DP every [NP boy]]
4. select laugh
5. merge 4 and 3 [VP laugh [DP every boy]]
6. select will
7. merge 6 and 5 [IP will [VP laugh [DP every boy]]]
8. move every boy [IP [DP every boy] [I′ will [VP laugh t]]]
Representing derivations

1. select every

2.

3. merge 1 and 2

4.

5. merge 4 and 3

6.

7. merge 6 and 5

8.

9. move every boy
Representing derivations

1. select every
2. select boy
Representing derivations

1. select every
2. select boy
3. merge 1 and 2
   \[ DP \ every \ [ NP \ boy ] \]
Representing derivations

1. select every
2. select boy
3. merge 1 and 2
   \[DP \text{ every } [NP \text{ boy }]]\]
4. select laugh
Representing derivations

1. select every
2. select boy
3. merge 1 and 2
   
   $[\text{DP } \text{every } [\text{NP } \text{boy } ] ]$
4. select laugh
5. merge 4 and 3
   
   $[\text{VP } \text{laugh } [\text{DP } \text{every } \text{boy } ] ]$
Representing derivations

1. select every
2. select boy
3. merge 1 and 2
   \[ \text{[DP every [NP boy]]} \]
4. select laugh
5. merge 4 and 3
   \[ \text{[VP laugh [DP every boy]]} \]
6. select will
Representing derivations

1. select *every*
2. select *boy*
3. merge 1 and 2
   
   \[\text{[DP every [NP boy]]}\]
4. select *laugh*
5. merge 4 and 3
   
   \[\text{[VP laugh [DP every boy]]}\]
6. select *will*
7. merge 6 and 5
   
   \[\text{[IP will [VP laugh [DP every boy]]]}\]
Representing derivations

1. select *every*
2. select *boy*
3. merge 1 and 2
   \[ DP \text{ every } NP \text{ boy } ]
4. select *laugh*
5. merge 4 and 3
   \[ VP \text{ laugh } DP \text{ every boy } ]
6. select *will*
7. merge 6 and 5
   \[ IP \text{ will } VP \text{ laugh } DP \text{ every boy } ]
8. move *every boy*
   \[ IP[DP \text{ every boy } ][I' \text{ will } VP \text{ laugh } t]\]
The determinacy of movement

Attract Closest

Minimal Link

Shortest Move

SMC
can only be 1 thing moving for
a particular reason at any time
The determinacy of movement

Attract Closest

Minimal Link

Shortest Move

SMC
can only be 1 thing moving for
a particular reason at any time
The determinacy of movement

The proof objects of minimalism

- are first order (i.e. trees)
The determinacy of movement

move
  \merge
    \will
    \laugh
      \merge
        \every
        \boy

The proof objects of minimalism

- are first order (i.e. trees)
- the proofs of any proposition (e.g. S) form a regular tree language
Towards MCFGs (I.)

• a categorized string is a pair \( \phi = (u, \delta) \), where
  
  \( u \) is a string

  \( \delta \) is a feature bundle

• an expression is a finite sequence of categorized strings

  \( \phi_0, \ldots, \phi_n \)

• each \( \phi_i, 1 \leq i \leq n \) represents a moving subtree

• \( \phi_0 \) represents the rest of the tree
Towards MCFGs (II.)

\[
\begin{align*}
(u, \gamma), \phi_1, \ldots, \phi_m & \quad (v, x.\delta), \psi_1, \ldots, \psi_n \\
\end{align*}
\]

\[
\begin{align*}
(u, \gamma), \phi_1, \ldots, \phi_m, (v, \delta), \psi_1, \ldots, \psi_n
\end{align*}
\]
Towards MCFGs (II.)

\[(u, =x. \gamma), \phi_1, \ldots, \phi_m \quad (v, x), \psi_1, \ldots, \psi_n\]

\[(uv, \gamma), \phi_1, \ldots, \phi_m, \psi_1, \ldots, \psi_n\]
Towards MCFGs (III.)

\[
\begin{align*}
(u, +y \cdot \gamma), \phi_1, \ldots, \phi_{j-1}, (v, -y), \phi_{j+1}, \ldots, \phi_m \\
\Rightarrow (vu, \gamma), \phi_1, \ldots, \phi_{j-1}, \phi_{j+1}, \ldots, \phi_m
\end{align*}
\]
Automata

An rule like:

\[(u, +y \cdot \gamma), \phi_1, \ldots, \phi_{j-1}, (v, -y), \phi_{j+1}, \ldots, \phi_m\]
\[(vu, \gamma), \phi_1, \ldots, \phi_{j-1}, \phi_{j+1}, \ldots, \phi_m\]

gives us an ldmbutts (tree-to-string) production:

\[\text{move}(q(u, v_1, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_m)) \rightarrow q'(vu, v_1, \ldots, v_{j-1}, v_{j+1}, \ldots, v_m)\]

where

\[q = \langle +y \cdot \gamma, \delta_1, \ldots, \delta_{j-1}, -y, \delta_j, \ldots, \delta_m \rangle\]
\[q' = \langle \gamma, \delta_1, \ldots, \delta_{j-1}, \delta_j, \ldots, \delta_m \rangle\]
An example
An example

every

(every, =n.d.−k)
An example

every boy

(every, =n.d.−k) (boy, n)
An example

merge

<table>
<thead>
<tr>
<th>every</th>
<th>boy</th>
</tr>
</thead>
</table>

\[
\begin{array}{c}
\text{(every, } =\text{n.d.} - \text{k)} \quad \text{(boy, n)} \\
\hline
\text{(every boy, d.} - \text{k)}
\end{array}
\]
An example

\[
\begin{align*}
&\text{laugh} \\
&\text{merge}
\end{align*}
\]

\[
\begin{align*}
\text{every} & \quad \text{boy} \\
\text{every boy} & \quad (\text{every, } = n.d.-k) \\
& \quad (\text{boy, } n)
\end{align*}
\]

\[
\begin{align*}
\text{(laugh, } = d.v) & \quad \text{(every boy, } d.-k)
\end{align*}
\]
An example

\[
\begin{align*}
    \text{merge} & \quad \text{laugh} \quad \text{merge} \\
    \text{every} & \quad \text{boy} \\
\end{align*}
\]

\[
\begin{align*}
    \text{(laugh, =d.v)} & \quad \text{(every, =n.d.-k)} \quad \text{(boy, n)} \\
    \text{(every, boy, d.-k)} & \quad \text{(laugh, v), (every, boy, -k)}
\end{align*}
\]
An example

\[
\begin{align*}
\text{will} & \quad \text{merge} \\
\text{laugh} & \quad \text{merge} \\
\text{every} & \quad \text{boy}
\end{align*}
\]

\[
\begin{align*}
\text{(will, } & =v.+k.s) \\
\text{laugh, } & =d.v) \\
\text{(every boy, } & =n.d.-k) \\
\text{(every boy, } & d.-k) \\
\text{(every boy, } & =-k) 
\end{align*}
\]
An example

merge
  \(_{\text{will}}\)
  \(_{\text{merge}}\)
  \(_{\text{laugh}}\)
  \(_{\text{merge}}\)
  \(_{\text{every boy}}\)

\((\text{will, } = v. + k.s) \quad (\text{every boy, } - k)\)
\((\text{laugh, } = d.v) \quad (\text{every boy, } d. - k)\)
\((\text{every, } = n.d. - k) \quad (\text{boy, } n)\)

\((\text{will laugh, } + k.s), (\text{every boy, } - k)\)
An example

\[
\begin{align*}
\text{(will,} &= v. + k.s) \\
\text{(laugh,} &= d.v) \\
\text{(every,} &= n.d. - k) \\
\text{(every boy,} &= d. - k) \\
\text{(boy,} &= n) \\
\text{(every boy will laugh,} &= s) \\
\end{align*}
\]
An example

\[
\begin{align*}
&\text{(will, } v + k.s\text{)} \quad \frac{\text{(laugh, } d.v\text{)}}{\text{every boy, } d.-k}\quad \frac{\text{(every, } = n.d.-k\text{)}}{\text{every, } n}\quad \frac{\text{(boy, } n\text{)}}{\text{(every boy, } d.-k\text{)}} \\
&\text{(will, } v + k.s\text{), (every boy, } -k\text{)} \\
&\text{(every boy will laugh, } s\text{)}
\end{align*}
\]
A slightly larger example

boy     n
every   =n.d.-k
laugh   =d.v
will    =v.+k.s

to      =v.i
seem    =i.v
More derivations
Yoda

\begin{align*}
\text{boy} & \quad \text{n} \\
\text{every} & \quad =\text{n.d.-k} \\
\text{laugh} & \quad =\text{d.v} \\
\text{will} & \quad =\text{v.+k.s} \\
\text{to} & \quad =\text{v.i} \\
\text{seem} & \quad =\text{i.v} \\
\epsilon & \quad =\text{v.v.-top} \\
\epsilon & \quad =\text{s.+top.c}
\end{align*}
Remnant movement
Parsing
Top-down parsing

Items represent cuts of derivation tree
Top-down parsing

Items represent cuts of derivation tree

move

|  □  |
Top-down parsing

Items represent cuts of derivation tree

move
  |
merge
  /
  /
□   □
Top-down parsing

Items represent cuts of derivation tree

```
move
  /
merge
  /
  /
merge
  /
  /
  /
  /
  /
  /
  /
  /
  /
```
Top-down parsing

Items represent cuts of derivation tree

```
move
  |  merge
    |  merge
      |  merge
        |  merge
          |  merge
            |  merge

```
Top-down parsing

Items represent cuts of derivation tree

```
move
  |   merge
  |     □
  □   merge
  □     □
        merge
        every
```
Top-down parsing

Items represent cuts of derivation tree

move
  merge
    □ merge
      □ merge
every boy
Top-down parsing

Items represent cuts of derivation tree

```
move
  merge
    will
      merge
        □
        merge
          every
          boy
```
Top-down parsing

Items represent cuts of derivation tree

move
merge
will
merge

laugh
merge
every
boy
Local trees

this exploits:
MG derivation trees form a local set
Local trees

this exploits:
MG derivation trees form a local set

move
l
+k.s;−k
Local trees

this exploits:
MG derivation trees form a local set

move
  |          merge
  |          
=v.+k.s   v;−k
this exploits:
MG derivation trees form a local set

```
move
  merge
    =v.+k.s
        merge
          =d.v
            d.-k
```
Local trees

this exploits:
MG derivation trees form a local set

```
move
  merge
    =v.+k.s
  merge
    =d.v
    merge
      =n.d.-k
      n
```
Local trees

this exploits:
MG derivation trees form a local set

move
merge
\[=v.+k.s\]
merge
\[=d.v\]
merge
every
\[n\]
Local trees

this exploits:
MG derivation trees form a local set

```
move
  /
merge
  /
  =v.+k.s
merge
  /
  =d.v
merge
  /
every
  /
boy
```
Local trees

**this exploits:**
MG derivation trees form a local set

```
move
    |
merge
    |
will
    |
merge
    |
=d.v
    |
merge
    |
every
    |
boy
```

22
Local trees

this exploits:
MG derivation trees form a local set

```
move
  └─ merge
    └─ will
        └─ merge
            └─ laugh
                └─ merge
                    └─ every
                        └─ boy
```
Undoing movement

- When we hypothesize a move node:
  
  move
  
  +k.s;−k
Undoing movement

- When we hypothesize a move node:
  \[
  \text{move} \quad \text{merge} \quad +k.s; -k
  \]

- We next must hypothesize where the mover is:
  \[
  \begin{array}{c}
  \text{move} \\
  \quad \text{merge} \\
  \quad \text{merge} \\
  \quad \text{merge} \\
  \end{array}
  \]
  \[
  \begin{array}{c}
  +k.s; -k \\
  d.; -k
  \end{array}
  \]
Appearances can be deceiving

*Every boy will (seem to)* laugh
If only

move

merge

merge

☐ ☐

d.−k

Might work in this case, but is there a non-analysis specific principle?
If only...

- Might work in this case,
  - but is there a non-analysis specific principle?
Structure in derivations

MG derivations are *subregular*  
(Tier-based) *strictly local*  

**strict locality** conjunction of negative literals  
(with immediate successor)

**tier-based** relativized successors  
($\triangleleft_T$, where $T \subseteq \Sigma$)
Example (strings)

Primary stress
\(\triangleleft \equiv \triangleleft_\sigma\)

Have primary stress \(\neg(\$ \triangleleft \$)\)

Have at most one stress \(\neg(\dot{\sigma} \triangleleft \dot{\sigma})\)
Example (trees)

Movement
\(\triangleq \triangleleft +_k, -_k\)

Movers gonna move \(\lnot (\$ \triangleleft \ell)\)

No movement without movement \(\lnot (\text{move} \triangleleft \$)\)

No competition \(\lnot (\text{move} \triangleleft \ell_1, \ell_2)\)
Argument structure via $n$-grams

Every lexical item $\ell$ appears in a derivation with a unique local context

- depends exclusively on positive feature sequence
  
  ($=x$ and $+y$)

  
  (will, =$v$.+$k$.s)
Every lexical item \( \ell \) appears in a derivation with a unique local context

- depends exclusively on positive feature sequence
  \((=x \text{ and } +y)\)

```
merge
\(\text{(will, } =v. +k.s)\)
```
Argument structure via \( n \)-grams

Every lexical item \( \ell \) appears in a derivation with a unique local context

- depends exclusively on positive feature sequence
  \((=x \text{ and } +y)\)

\[
\begin{align*}
\text{move} \\
\downarrow \\
\text{merge} \\
\quad \downarrow \\
(will, =v. +k.s) \\
\end{align*}
\]
Exploiting regularities in derivations

- When we hypothesize a move node:

  move
  └ └
     +k.s;−k
When we hypothesize a move node:

```
move
| +k.s;−k
```

We know it immediately dominates a mover (on the relevant tier):

```
move
\ (+k.s)
\   | 
\   d.−k
```
A sketch
A sketch

move
A sketch

move

merge

□ □
A sketch

move
merge
every
A sketch

move
merge
every
boy
move

merge

merge

every

boy
A sketch

move

merge

will

merge

every

boy
A sketch

move
  ↓
merge
  ↓
will
  ↓
merge
  ↓
merge
  ↓
every
  ↓
boy
A sketch

move

merge

will

merge

laugh

merge

every

boy
A sketch

move
  /  
merge

will merge

laugh merge

every boy
A basic ‘hole’ data structure

- $g$ is a gorn address
  where we are in the derived tree

- $xs$ is a (finite) list of derivations with holes
  elements in separate tiers
  ... paired with feature bundles
  information about the occupied tier

\[
\alpha^g
\]
\[
|\quad xs
\]

data Hole t b x = Hole t [(b,x)]
A basic 'hole' data structure

\[ \alpha^g \]
\[ \begin{array}{c} \alpha^g \\ | \\ xs \end{array} \]

- \( g \) is a gorn address where we are in the derived tree
- \( xs \) is a (finite) list of derivations with holes paired with feature bundles information about the occupied tier

```haskell
data Hole t b x = Hole t [(b, x)]
```
A basic ’hole’ data structure

- \( \alpha \)
  - \( g \) is a gorn address
  - where we are in the derived tree
- \( xs \)
  - is a (finite) list of
    - derivations with holes
      - elements in separate tiers

\[
\text{data Hole } t \ b \ x = \text{Hole } t \ [(b,x)]
\]
A basic 'hole' data structure

\[ \alpha^g \]

\[ \begin{array}{c}
\alpha \\
\downarrow \\
xs
\end{array} \]

- \( g \) is a gorn address
  
  where we are in the derived tree
- \( xs \) is a (finite) list of
  - derivations with holes
    
    elements in separate tiers
  - \( \ldots \) paired with feature bundles
    
    information about the occupied tier

\[
data \ \text{Hole} \ t \ b \ x = \text{Hole} \ t \ [(b, x)]
\]
Given

\[\alpha : g\]

\[\text{sort (us ++ vs)}\]
• Given

\[ \alpha : g \]

\[ \downarrow \]

\[ XS \]

• `merge` could have applied

\[
\text{merge} \\
\text{=} \ x \cdot \alpha : g0 \\
\text{=} \ x : g1 \\
\downarrow \\
\text{US} \\
\downarrow \\
\text{VS}
\]
• Given

\( \alpha : g \)

\( \downarrow \)

\( XS \)

• \textbf{merge} could have applied

\[ \text{merge} \]

\[ \text{\( =x.\alpha : g0 \)} \quad \text{x : g1} \]

\[ \downarrow \quad \downarrow \]

\( US \quad VS \)
Unmerge1

- Given

\[
\alpha : g \\
\downarrow \\
xS
\]

- merge could have applied

\[
\text{merge} \\
\downarrow \\
=x.\alpha : g0 \\
\downarrow \\
x : g1 \\
\downarrow \\
us \\
\downarrow \\
vs
\]

\[
xs = \text{sort (us ++ vs)}
\]
Unmove

- Given

\[
\alpha : g \quad | \\
\text{xs}
\]
Unmove

- Given

  \[
  \alpha : g \\
  \downarrow \\
  xs
  \]

- move could have applied

  \[
  \text{move} \quad \downarrow \\
  +y.\alpha : g1 \\
  \text{x.-y} \\
  \text{x.-y : g0} \\
  \downarrow \\
  xs
  \]
Given

\[ \alpha^g \]

\[ | \]

\[ XS \]
• Given

\[
\begin{array}{c}
\alpha^g \\
| \\
xs
\end{array}
\]

• merge could have applied to a mover

\[
\begin{array}{c}
\text{merge} \\
\alpha^g \\
= x.\alpha^{g_1} \\
\text{us} \\
x.\neg y \\
\text{vs}
\end{array}
\]
Given

merge could have applied to a mover

= \varsigma \alpha \Gamma_1

\x - \gamma

us

vs
Given

\[ \alpha \ g \ x \ s \]

merge could have applied to a mover:

\[
\begin{array}{c}
\alpha^g \\
\downarrow \\
xs
\end{array}
\]

\[=x.\alpha^{g_1} \]

\[x.-y\]

\[
\begin{array}{c}
\downarrow \\
us
\end{array}
\]

\[
\begin{array}{c}
\downarrow \\
vs
\end{array}
\]

\[xs = \text{sort (us ++ vs)}\]
Completion (I)

- Given

\[
\begin{align*}
\text{x} - \text{y} \\
\text{x} - \text{y}
\end{align*}
\]
Completion (I)

- Given

- This is the tree you’re looking for
ATNs and filling gaps

- Psycholinguists
  - you process moved items (fillers)
  - and then you try to find where they moved from (gap)
- TD MG parsing
  to process filler, first find gap!
- Here
  - \textit{unmove} constructs a filler
  - \textit{unmerge2} constructs a gap
  - \textit{complete} fills the gap
Remnant movement

\[
\text{move} \rightarrow \text{merge} \rightarrow \varepsilon \rightarrow \text{move} \rightarrow \text{merge} \rightarrow \varepsilon \rightarrow \text{move} \rightarrow \text{merge} \rightarrow \varepsilon \rightarrow \text{move} \rightarrow \text{merge} \rightarrow \varepsilon \rightarrow \text{move} \rightarrow \text{merge} \rightarrow \text{every} \rightarrow \text{boy}.
\]

\[
\begin{align*}
\text{boy} & \rightarrow n \\
\text{every} & \rightarrow =n.d.-k \\
\text{laugh} & \rightarrow =d.v \\
\text{will} & \rightarrow =v.+k.s \\
\varepsilon & \rightarrow =v.v.-\text{top} \\
\varepsilon & \rightarrow =s.+\text{top.s}
\end{align*}
\]
Remnant movement

$S^n$
Remnant movement

\[
\text{move} \\
+\text{top.s}^1 \\
\text{v.-top} \\
\text{v.-top}^0
\]
Remnant movement

move

+top.s\[1\]

v.-top

merge

=v.v.-top\[00\] \v[01]
Remnant movement

\[
\text{move} \\
\text{merge} \\
\epsilon \\
\text{v}^{01} \\
\text{v}^{01} \\
\text{v}^{01} \\
\text{v}^{01} \\
\text{v}^{01}
\]

\[+\text{top.s}^1\]
Remnant movement

\[
\text{move} \\
\text{+top.$s^1$} \\
\text{v.-top} \\
\text{merge} \\
\epsilon \\
\text{merge} \\
\text{=d.v} \\
\text{d.-k}
\]
Remnant movement

move
\[ +\text{top.s}^1 \]
\[ v\text{-top} \]
merge
\[ \epsilon \text{merge} \]
\[ \text{laugh} \quad \text{d.-k} \]
Remnant movement
Remnant movement

move
  merge
    \(\epsilon\)
      v.\text{-top}
        merge
          \(s^{11}\)
          \(\epsilon\)
            merge
              \(\epsilon\)
                \(\text{laugh}\)
                  \(d.-k\)
Remnant movement

move
\[
\begin{array}{c}
\text{merge} \\
\epsilon \\
\text{move} \\
\text{\texttt{+k.s}}^\text{111} \\
\text{d.-k/} \\
\text{d.-k}^\text{110} \\
\text{v.-top} \\
\text{merge} \\
\epsilon \\
\text{merge} \\
\text{laugh} \\
\text{d.-k}
\end{array}
\]
Remnant movement

\[
\begin{array}{c}
\text{move} \\
\text{merge} \\
\epsilon \quad \text{move} \\
+k.s^{111} \\
d.-k \\
\text{merge} \\
\text{merge} \\
=n.d.-k^{1100} n^{1101} e \\
\text{laugh} \\
d.-k
\end{array}
\]
Remnant movement

move
  merge
    \( \epsilon \)
    move
      \(+k.s^{111}\)
    d.-k
    merge
      every
        n^{1101}
      \( \epsilon \)
      merge
        merge
          \( \epsilon \)
          merge
            laugh
              d.-k
v.-top
merge

Remnant movement

move
  |
merge
  |
  ε
  |
  move
  |
    +k.s
    |
    d.-k
    |
merge
  |
  every
  |
boy
  |
  ε
  |
merge
  |
laugh
  |
d.-k
Remnant movement

\[
\begin{align*}
\text{move} & \quad | \\
\text{merge} & \\
\epsilon & \quad \text{move} \\
\text{merge} & \\
=v+k.s & 1110 \\
\text{v.-top} & \\
\text{d.-k} & \quad \text{v.-top} \\
\text{merge} & \quad \text{merge} \\
\text{every} & \quad \text{boy} \\
\epsilon & \quad \text{merge} \\
\text{laugh} & \quad \text{d.-k}
\end{align*}
\]
Remnant movement

move
  |merge
  |ε
move
  |merge
  |will
  |v.-top
d.-k
merge
  v.-top
merge
every
boy
ε
merge
laugh
d.-k
Remnant movement

move
  └── merge
    └── move
        └── merge
            └── will
                └── merge
                    └── laugh
                        └── d→k
                            └── d→k
                                └── merge
                                    └── every
                                        └── boy
Remnant movement

move
  /   
mmerge
  
  \   
  \move
  
  \merge
  
  \ε

  \will

  \merge
  
  \ε

  \merge

  \ ε

  \merge

  \laugh

  \merge

  \ every

  \merge

  \ boy
Enforcing the SMC

Recall:

1. Movers gonna move : \( \neg (\$ \vartriangleleft \ell) \)
2. No movement without movement : \( \neg (\text{move} \vartriangleleft \$) \)
3. No competition : \( \neg (\text{move} \vartriangleleft \ell_1, \ell_2) \)

How are these being enforced?
Enforcing the SMC

Recall:

1. Movers gonna move: \( \neg (\$ \triangleleft \ell) \)
2. No movement without movement: \( \neg (\text{move} \triangleleft \$) \)
3. No competition: \( \neg (\text{move} \triangleleft \ell_1, \ell_2) \)

How are these being enforced?

1. Two ways of generating a mover:
Enforcing the SMC

Recall:

1. Movers gonna move : \( \neg ($ \triangleleft \ell) \)
2. No movement without movement : \( \neg (\text{move} \triangleleft $) \)
3. No competition : \( \neg (\text{move} \triangleleft \ell_1, \ell_2) \)

How are these being enforced?

1. Two ways of generating a mover:
   - via unmerge2 (i.e. a gap)
     must be filled
   - via unmmove (i.e. a filler)
     born dominated
Enforcing the SMC

Recall:

1. Movers gonna move: \( \neg (\$ \triangleleft \ell) \)
2. No movement without movement: \( \neg (\text{move} \triangleleft \$) \)
3. No competition: \( \neg (\text{move} \triangleleft \ell_1, \ell_2) \)

How are these being enforced?

1. Two ways of generating a mover:
2. move nodes and movers postulated simultaneously
Enforcing the SMC

Recall:

1. Movers gonna move : \( \neg (\$ \triangleleft \ell) \)
2. No movement without movement : \( \neg (\text{move} \triangleleft \$) \)
3. No competition : \( \neg (\text{move} \triangleleft \ell_1, \ell_2) \)

How are these being enforced?

1. Two ways of generating a mover:
2. \text{move} nodes and movers postulated simultaneously
3. via restrictions
Restricting Unmove

As long as nothing in \( xs \) is on the \(-y\) tier
If there is something on the $-y$ tier in $xs$ it must complete this gap

*in other words, the $-y$ tier is hereby blocked!*
A partial proof tree with an \( n \)-ary hole is an operation of type

\[
(\alpha_1 \to \cdots \to \alpha_n \to t) \to t
\]

The \( \alpha \)'s are the types of the arguments to the hole

**Upper bounds on**

1. number of holes
2. their arity

depending on number of \(-y\) feature types in lexicon
Completion (III)

\[
\begin{array}{c}
\text{x. -y} \\
\Delta \\
x. -y
\end{array}
\quad \Rightarrow \quad \Delta[us_1, \ldots, us_k]
\]

Conditions

\[xs = \text{sort ( us1 ++ \ldots ++ usk )}\]

each substitution path is free for the relevant tier
A Note on Semantic Interpretation

\[
\begin{align*}
[\text{merge}] & \mapsto \lambda m, n.(|m \oplus n|) \\
[\text{merge}] & \mapsto \lambda m, n.(|m \oplus \Box n|) \\
[\text{move}] & \mapsto \lambda m.m \\
[\text{move}] & \mapsto \lambda m.\langle m \rangle^k \\
[I(\ell)] & = I(\ell)^\uparrow
\end{align*}
\]

\[
(\mid f \ m \ n \mid) = \text{do} \\
\quad x \leftarrow m \\
\quad y \leftarrow n \\
\quad \text{return} \ (f \ x \ y)
\]
The meaning of partial parse trees

\[
\lambda f_\bigcirc. \langle f_\bigcirc (\{ every \} \{ boy \}) \uparrow \rangle^k_{FA} = \lambda f_\bigcirc. [move](f_\bigcirc ([merge] [every] [boy]))
\]
Conclusion: Exploiting structure

- MGs have more structure in their derivations than is being made use of
  - how can we take advantage of it?
- Simple intersection w/ regular sets:
  \[ (\text{will}, r^{v_s} + p^q + c^s), \text{ where } \delta(q, \text{will}) = r \]

- how to do scheduling to obtain a version of the present algorithm?
- Left-corner parsing (for CFGs) has similar looking partial proof trees
  - can we use these ideas to get a left-corner parser for MGs and solve the problem of left branch movement?