Sharpening the empirical claims of generative syntax through formalization

Tim Hunter

University of Minnesota, Twin Cities

NASSLLI, June 2014
Part 1: Grammars and cognitive hypotheses
   What is a grammar?
   What can grammars do?
   Concrete illustration of a target: Surprisal

Parts 2–4: Assembling the pieces
   Minimalist Grammars (MGs)
   MGs and MCFGs
   Probabilities on MGs

Part 5: Learning and wrap-up
   Something slightly different: Learning model
   Recap and open questions
Sharpening the empirical claims of generative syntax through formalization

Tim Hunter — NASSLLI, June 2014

Part 3

MGs and MCFGs
Where we’re up to

We’ve seen:

- MGs with operations defined that manipulated trees
- that the structure that “really matters” (e.g. for recursion) can be boiled down to funny-looking “derivation trees” (with things like \( t, \{-k\} \) at the non-leaf nodes)

Now:

- A way to think of how these derivation trees relate to surface strings (without going via trees)
- In some ways not totally necessary for the rest of the course, but helpful

Later:

- Adding probabilities to MGs: in a way that sort of works, and does some good stuff, but doesn’t do everything we’d want
- Adding probabilities to MGs: in an even better way
Outline

9  A different perspective on CFGs

10  Concatenative and non-concatenative operations

11  MCFGs

12  Back to MGs
Trees

```
the boy likes cake :: S
the boy :: NP
likes cake :: VP
the :: D
boy :: N
likes :: V
cake :: NP

cake :: N
```

```
S

NP

V

D
the

N
boy

V
likes

N
cake

NP
```
Trees

How to think of a tree:

- less as a picture of a string
- more as a graphical representation of how a string was constructed, with the string “at” the top node
Two sides of a CFG rule

A rule like ‘S → NP VP’ says two things:

- What combines with what:
  An NP and a VP can combine to form an S
- How to produce a string of the new category:
  Put the NP-string to the left of the VP-string

More explicitly:

\[ st :: S \quad \rightarrow \quad s :: NP \quad t :: VP \]
Example: X-bar theory

Japanese
XP → Spec X'
X' → Comp X

English
XP → Spec X'
X' → X Comp
Example: X-bar theory

Japanese
XP → Spec X'
X' → Comp X

English
XP → Spec X'
X' → X Comp

Japanese
st :: XP → s :: Spec t :: X'
st :: X' → s :: Comp t :: X

English
st :: XP → s :: Spec t :: X'
ts :: X' → s :: Comp t :: X
Example: X-bar theory

Japanese
XP → Spec X’
X’ → Comp X

English
XP → Spec X’
X’ → X Comp

Japanese
st :: XP → s :: Spec t :: X’
st :: X’ → s :: Comp t :: X

English
st :: XP → s :: Spec t :: X’
ts :: X’ → s :: Comp t :: X

John-ga Mary-o mita :: VP
John-ga :: Spec Mary-o mita :: V’
Mary-o :: Comp mita :: V

John saw Mary :: VP
John :: Spec saw Mary :: V’
Mary :: Comp saw :: V
Outline

9 A different perspective on CFGs

10 Concatenative and non-concatenative operations

11 MCFGs

12 Back to MGs
Concatenative and non-concatenative operations

**Concatenative morphology:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>play + ed</td>
<td>played</td>
</tr>
<tr>
<td>play + ing</td>
<td>playing</td>
</tr>
<tr>
<td>play + s</td>
<td>plays</td>
</tr>
</tbody>
</table>

**Non-concatenative morphology:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k,t,b) + (i,aa)</td>
<td>kitaab</td>
<td>“book”</td>
</tr>
<tr>
<td>(k,t,b) + (aa,i)</td>
<td>kaatib</td>
<td>“writer”</td>
</tr>
<tr>
<td>(k,t,b) + (ma,uu)</td>
<td>maktuub</td>
<td>“written”</td>
</tr>
<tr>
<td>(k,t,b) + (a,i,a)</td>
<td>katiba</td>
<td>“document”</td>
</tr>
</tbody>
</table>
Concatenative and non-concatenative operations

**Concatenative morphology:**
- play + ed $\rightsquigarrow$ played
- play + ing $\rightsquigarrow$ playing
- play + s $\rightsquigarrow$ plays

**Non-concatenative morphology:**
- (k,t,b) + (i,aa) $\rightsquigarrow$ kitaab ("book")
- (k,t,b) + (aa,i) $\rightsquigarrow$ kaatib ("writer")
- (k,t,b) + (ma,uu) $\rightsquigarrow$ maktuub ("written")
- (k,t,b) + (a,i,a) $\rightsquigarrow$ katiba ("document")

**Concatenative syntax:**
- plays + tennis $\rightsquigarrow$ plays tennis
- plays + soccer $\rightsquigarrow$ plays soccer
- John + plays soccer $\rightsquigarrow$ John plays soccer
- Mary + plays soccer $\rightsquigarrow$ Mary plays soccer
Concatenative and non-concatenative operations

**Concatenative morphology:**

- play + ed ↦ played
- play + ing ↦ playing
- play + s ↦ plays

**Non-concatenative morphology:**

- (k,t,b) + (i,aa) ↦ kitaab (“book”)
- (k,t,b) + (aa,i) ↦ kaatib (“writer”)
- (k,t,b) + (ma,uu) ↦ maktuub (“written”)
- (k,t,b) + (a,i,a) ↦ katiba (“document”)

**Concatenative syntax:**

- plays + tennis ↦ plays tennis
- plays + soccer ↦ plays soccer
- John + plays soccer ↦ John plays soccer
- Mary + plays soccer ↦ Mary plays soccer

**Non-concatenative syntax:**

- seems + (John, to be tall) ↦ John seems to be tall
- seems + (Mary, to be intelligent) ↦ Mary seems to be intelligent
- did + (John see, who) ↦ who did John see
- did + (Mary meet, who) ↦ who did Mary meet
Non-concatenative morphology

$kitaab :: A$

$\langle k,t,b \rangle :: B$

$\langle i,aa \rangle :: C$

$kaatib :: A$

$\langle k,t,b \rangle :: B$

$\langle aa,i \rangle :: C$

$kutub :: A$

$\langle k,t,b \rangle :: B$

$\langle u,u \rangle :: C$

$stuvw :: A \rightarrow \langle s, u, w \rangle :: B$

$\langle t, v \rangle :: C$
Non-concatenative morphology

A different perspective on CFGs
Concatenative and non-concatenative operations
MCFGs
Back to MGs

kitaab :: A
  ⟨k,t,b⟩ :: B  ⟨i,aa⟩ :: C

kaatib :: A
  ⟨k,t,b⟩ :: B  ⟨aa,i⟩ :: C

kutub :: A
  ⟨k,t,b⟩ :: B  ⟨u,u⟩ :: C

stuvw :: A  →  ⟨s, u, w⟩ :: B  ⟨t, v⟩ :: C

gespielt :: E
  spiel :: A  ⟨ge,t⟩ :: D

gekauft :: E
  kauf :: A  ⟨ge,t⟩ :: D

gemacht :: E
  mach :: A  ⟨ge,t⟩ :: D

stu :: E  →  t :: A  ⟨s, u⟩ :: D
Non-concatenative morphology

\[
\begin{align*}
  stuvw &:: A \rightarrow \langle s, u, w \rangle :: B \quad \langle t, v \rangle :: C \\
stu &:: E \rightarrow t :: A \quad \langle s, u \rangle :: D
\end{align*}
\]
Non-concatenative morphology

\[ stuvw :: A \rightarrow \langle s, u, w \rangle :: B \quad \langle t, v \rangle :: C \]
\[ stu :: E \rightarrow t :: A \quad \langle s, u \rangle :: D \]

If our goal is to characterize the array of well-formed/derivable objects — not to pronounce them — then all we care about is “what’s built out of what”:
Non-concatenative morphology

\[
\begin{align*}
stuvw &:: A &\rightarrow& \langle s, u, w \rangle &:: B &\langle t, v \rangle &:: C \\
stu &:: E &\rightarrow& t &:: A &\langle s, u \rangle &:: D \\
\langle ts, u \rangle &:: D &\rightarrow& t &:: F &\langle s, u \rangle &:: D \\
gekitaabt &:: E \\
kitaab &:: A &\langle ge, t \rangle &:: D \\
\langle k, t, b \rangle &:: B &\langle i, aa \rangle &:: C \\
ausgekitaabt &:: E \\
kitaab &:: A &\langle ausge, t \rangle &:: D \\
\langle k, t, b \rangle &:: B &\langle i, aa \rangle &:: C &aus &:: F &\langle ge, t \rangle &:: D
\end{align*}
\]
Non-concatenative morphology

If our goal is to characterize the array of well-formed/derivable objects — not to pronounce them — then all we care about is “what’s built out of what”:

\[
\begin{align*}
A & \rightarrow B \quad C \\
E & \rightarrow A \quad D \\
D & \rightarrow F \quad D
\end{align*}
\]
Multiple Context-Free Grammars (MCFGs)

\[ st :: S \rightarrow s :: NP \quad t :: VP \]

An MCFG generalises to allow yields to be *tuples of strings*.

\[ t_2 t_1 :: Q \rightarrow s :: NP \quad \langle t_1, t_2 \rangle :: VPWH \]

This rule says two things:

- We can combine an NP with a VPWH to make a Q.
- The yield of the Q is \( t_2 t_1 \), where \( s \) is the yield of the NP and \( \langle t_1, t_2 \rangle \) is the yield of the VPWH.
Multiple Context-Free Grammars (MCFGs)

\[
st :: S \quad \rightarrow \quad s :: \text{NP} \quad t :: \text{VP}
\]

An MCFG generalises to allow yields to be *tuples of strings*.

\[
t_{2}s_{1} :: Q \quad \rightarrow \quad s :: \text{NP} \quad \langle t_{1}, t_{2} \rangle :: \text{VPWH}
\]

This rule says two things:
- We can combine an NP with a VPWH to make a Q.
- The yield of the Q is \( t_{2}s_{1} \), where \( s \) is the yield of the NP and \( \langle t_{1}, t_{2} \rangle \) is the yield of the VPWH.

\[
\text{which girl the boy says is tall} :: Q \quad \rightarrow \quad
\text{the boy} :: \text{NP} \quad \langle \text{says is tall}, \text{which girl} \rangle :: \text{VPWH}
\]
Each nonterminal has a rank $n$, and yields only $n$-tuples of strings.

So given this rule:

$$t_2s t_1 :: Q \rightarrow s :: NP \quad \langle t_1, t_2 \rangle :: VPWH$$

we know that anything producing a VPWH must produce a 2-tuple.

$$\langle \ldots, \ldots \rangle :: VPWH \rightarrow \ldots$$

and that anything producing an NP must produce a 1-tuple:

$$\ldots :: NP \rightarrow \ldots$$

Some technical details

- Each nonterminal has a rank $n$, and yields only $n$-tuples of strings.

So given this rule:

$$t_2 s t_1 :: Q \rightarrow s :: NP \quad \langle t_1, t_2 \rangle :: VPWH$$

we know that anything producing a VPWH must produce a 2-tuple.

$$\langle \ldots, \ldots \rangle :: VPWH \rightarrow \ldots$$

and that anything producing an NP must produce a 1-tuple:

$$\ldots :: NP \rightarrow \ldots$$

- The string-composition functions cannot copy pieces of their arguments.

<table>
<thead>
<tr>
<th>OK</th>
<th>$s \ t :: VP \rightarrow s :: V \ t :: NP$</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>$t \ s \ \text{himself} :: S \rightarrow s :: V \ t :: NP$</td>
</tr>
<tr>
<td>Not OK</td>
<td>$t \ s \ t :: S \rightarrow s :: V \ t :: NP$</td>
</tr>
</tbody>
</table>
Some technical details

- Each nonterminal has a rank $n$, and yields only $n$-tuples of strings.

So given this rule:

$$t_2 s t_1 :: Q \rightarrow s :: NP \quad \langle t_1, t_2 \rangle :: VPWH$$

we know that anything producing a VPWH must produce a 2-tuple.

$$\langle \ldots, \ldots \rangle :: VPWH \rightarrow \ldots$$

and that anything producing an NP must produce a 1-tuple:

$$\ldots :: NP \rightarrow \ldots$$

- The string-composition functions cannot copy pieces of their arguments.

  OK $s t :: VP \rightarrow s :: V \quad t :: NP$

  OK $t s \text{ himself} :: S \rightarrow s :: V \quad t :: NP$

  Not OK $t s t :: S \rightarrow s :: V \quad t :: NP$

- Essentially equivalent to linear context-free rewriting systems (LCFRSs).

Beyond context-free

Unlike in a CFG, we can ensure that the two “halves” are extended in the same ways without concatenating them together.
For comparison

\[ t_1 t_2 :: S \rightarrow t_1 :: A \ s :: S \ t_2 :: A \]

\[ t_1 t_2 :: S \rightarrow t_1 :: B \ s :: S \ t_2 :: B \]

\[ \epsilon :: S \]

\[ a :: A \]

\[ b :: B \]

\[ abaaaaba :: S \]

\[ a :: A \]
\[ baaaab :: S \]
\[ a :: A \]
\[ b :: B \]
\[ aaaa :: S \]
\[ b :: B \]
\[ aa :: S \]
\[ a :: A \]
\[ a :: A \]
\[ a :: A \]
\[ \epsilon :: S \]
\[ a :: A \]

\[ a :: A \]

\[ b :: B \]

\[ a :: A \]

\[ a :: A \]
Outline

9  A different perspective on CFGs

10  Concatenative and non-concatenative operations

11  MCFGs

12  Back to MGs
Merge and move

\[ = f \alpha \]

\[ \beta \]

\[ \text{MERGE} \]

\[ \alpha \]

\[ \beta \]

\[ > \]

\[ \text{MERGE} \]

\[ \alpha \]

\[ \beta \]

\[ > \]

\[ \text{MOVE} \]

\[ \beta \]

\[ \alpha \]
What matters in a (derived) tree

This tree:

becomes a tuple of categorized strings:

\[ \langle s :: x, \quad t :: -f, \quad u :: -g \rangle_0 \]

or, equivalently, a tuple-of-strings, categorized by a tuple-of-categories:

\[ \langle s, t, u \rangle :: \langle x, -f, -g \rangle_0 \]

Remember MG derivation trees?

\[
t, \{\}
\]

\[
+k \ t, \{-k\}
\]

\[
\text{will} :: = v +k \ t
\]

\[
v, \{-k\}
\]

\[
\text{think} :: = t = \text{subj} \ v
\]

\[
t, \{\}
\]

\[
+k \ t, \{-k\}
\]

\[
\text{will} :: = v +k \ t
\]

\[
v, \{-k\}
\]

\[
\text{eat} :: = \text{obj} = \text{subj} \ v
\]

\[
\text{John} :: \text{subj} -k
\]

\[
\text{Mary} :: \text{subj} -k
\]

\[
\text{cake} :: \text{obj}
\]
Remember MG derivation trees?

\[
\begin{array}{c}
t, \{\}
\end{array}
\]

\[
\begin{array}{c}
+kt, \{-k\}
\end{array}
\]

\[
\begin{array}{c}
\text{will} :: =v +kt
\end{array}
\]

\[
\begin{array}{c}
v, \{-k\}
\end{array}
\]

\[
\begin{array}{c}
=\text{subj } v, \{\}
\end{array}
\]

\[
\begin{array}{c}
\text{Mary} :: \text{subj } -k
\end{array}
\]

\[
\begin{array}{c}
\text{think} :: =t =\text{subj } v
\end{array}
\]

\[
\begin{array}{c}
t, \{\}
\end{array}
\]

\[
\begin{array}{c}
+kt, \{-k\}
\end{array}
\]

\[
\begin{array}{c}
\text{will} :: =v +kt
\end{array}
\]

\[
\begin{array}{c}
v, \{-k\}
\end{array}
\]

\[
\begin{array}{c}
=\text{subj } v, \{\}
\end{array}
\]

\[
\begin{array}{c}
\text{John} :: \text{subj } -k
\end{array}
\]

\[
\begin{array}{c}
\text{eat} :: =\text{obj } =\text{subj } v
\end{array}
\]

\[
\begin{array}{c}
\text{cake} :: \text{obj}
\end{array}
\]

\[
\begin{array}{c}
=\text{subj } v
\end{array}
\]

\[
\begin{array}{c}
\text{Mary} :: \text{subj } -k
\end{array}
\]

\[
\begin{array}{c}
\text{think} :: =t =\text{subj } v
\end{array}
\]

\[
\begin{array}{c}
\langle +kt, -k \rangle
\end{array}
\]

\[
\begin{array}{c}
\text{will} :: =v +kt
\end{array}
\]

\[
\begin{array}{c}
\langle v, -k \rangle
\end{array}
\]

\[
\begin{array}{c}
=\text{subj } v
\end{array}
\]

\[
\begin{array}{c}
\text{Mary} :: \text{subj } -k
\end{array}
\]

\[
\begin{array}{c}
\text{eat} :: =\text{obj } =\text{subj } v
\end{array}
\]

\[
\begin{array}{c}
\text{cake} :: \text{obj}
\end{array}
\]
Remember MG derivation trees?

Slight change of notation (sorry): internal node labels are now lists of feature-lists.
Slight change of notation (sorry): internal node labels are now lists of feature-lists.

- We can tell that this tree represents a well-formed derivation, by checking the feature-manipulations at each step.
- How can we work out which string it derives?
  - Build up a tree according to merge and move rules, and read off leaves of the tree.
  - But there’s a simpler way.
Producing a string from a derivation tree

What do we need to have computed at the \( \langle +k \ t, -k \rangle \) node, in order to compute the final string

\[ \text{Mary will think John will eat cake} \]

at the \( t \) node?
Producing a string from a derivation tree

What do we need to have computed at the \( \langle +k \ t, -k \rangle \) node, in order to compute the final string

\[
\text{Mary will think John will eat cake}
\]

at the \( t \) node?

This tree would do:

\[
\langle \\
\text{will} :: +k \ t \\
\text{Mary} :: -k \\
\text{think} :: \\
\text{John} :: \\
\text{will} :: \\
\text{eat} :: \\
\text{cake} :: \\
\rangle
\]

But all we actually need to know is:

- What’s the string corresponding to the part that’s going to move to check \(-k\)?
- What’s the string corresponding to the leftovers?

These questions are answered by the tuple

\[
\langle \text{will think John will eat cake, Mary} \rangle
\]
Producing a string from a derivation tree

What do we need to have computed at the \(<v, -k>\) node, in order to compute the desired tuple

\[\langle \text{will think John will eat cake, Mary} \rangle\]

at the \(\langle +k \ t, -k \rangle\) node?
Producing a string from a derivation tree

What do we need to have computed at the \( \langle v, -k \rangle \) node, in order to compute the desired tuple

\[
\langle \text{will think John will eat cake, Mary} \rangle
\]
at the \( \langle +k \ t, -k \rangle \) node?

This tree would do:

But all we actually need to know is:
- What’s the string corresponding to the part that’s going to move to check \(-k\)?
- What’s the string corresponding to the leftovers?

These questions are answered by the tuple

\[
\langle \text{think John will eat cake, Mary} \rangle
\]
Producing a string from a derivation tree

What do we need to have computed at the \(=\text{subj } v\) node, in order to compute the desired tuple

\[
\langle \text{think John will eat cake}, \text{Mary} \rangle
\]

at the \(\langle v, -k \rangle\) node?
Producing a string from a derivation tree

What do we need to have computed at the \( =\text{subj } v \) node, in order to compute the desired tuple

\[ \langle \text{think John will eat cake, Mary} \rangle \]

at the \( \langle v, -k \rangle \) node?

This tree would do:

\[ \langle \text{think John will eat cake} \rangle \]

But all we actually need to know is:

- What’s the string corresponding to the entire tree? (The “leftovers after no movement”.)

This question is answered by the string

\[ \text{think John will eat cake} \]
What matters in a (derived) tree

This tree:

becomes a tuple of categorized strings:
\[ \langle s :: x, t :: \neg f, u :: \neg g \rangle_0 \]

or, equivalently, a tuple-of-strings, categorized by a tuple-of-categories:
\[ \langle s, t, u \rangle :: \langle x, \neg f, \neg g \rangle_0 \]

MCFG rules

```
\begin{align*}
\langle t \rangle & \\
\langle +k \, t, -k \rangle & \\
\langle v, -k \rangle & \\
\langle \text{will} \, ::= \, v \, +k \, t \rangle & \\
\langle \text{Mary} \, ::= \, \text{subj} \, -k \rangle & \\
\ldots & \\
\end{align*}
```

\[ t_2 \, t_1 :\!:: \, t \quad \rightarrow \quad \langle t_1, t_2 \rangle :\!:: \, \langle +k \, t, -k \rangle \]

\[ \text{Mary will think John will eat cake} :\!:: \, t \quad \rightarrow \quad \langle \text{will think John will eat cake, Mary} \rangle :\!:: \, \langle +k \, t, -k \rangle \]

\[ \langle st_1, t_2 \rangle :\!:: \, \langle +k \, t, -k \rangle \quad \rightarrow \quad s :\!:: \, v \, +k \, t \quad \langle t_1, t_2 \rangle :\!:: \, \langle v, -k \rangle \]

\[ \langle \text{will think John will eat cake, Mary} \rangle :\!:: \, \langle +k \, t, -k \rangle \rightarrow \text{will} :\!:: \, v \, +k \, t \quad \langle \text{think John will eat cake, Mary} \rangle :\!:: \, \langle v, -k \rangle \]

\[ \langle s, t \rangle :\!:: \, \langle v, -k \rangle \quad \rightarrow \quad s :\!:: \, \text{subj} \, v \quad t :\!:: \, \text{subj} \, -k \]

\[ \langle \text{think John will eat cake, Mary} \rangle :\!:: \, \langle v, -k \rangle \rightarrow \text{think John will eat cake} :\!:: \, \text{subj} \, v \quad \text{Mary} :\!:: \, \text{subj} \, -k \]
One slightly annoying wrinkle

We know that this is a valid derivational step:

```
α
=α
f
```

What is the corresponding MCFG rule?

**Selected thing on the right?**

```
st :: α  →  s :: =fα  t :: f
```

**Selected thing on the left?**

```
ts :: α  →  s :: =fα  t :: f
```
One slightly annoying wrinkle

We know that this is a valid derivational step:

\[
\alpha
\]

\[
= \alpha \quad f
\]

What is the corresponding MCFG rule?

Selected thing on the right?

\[
st :: \alpha \rightarrow s :: = \alpha \quad t :: f
\]

Selected thing on the left?

\[
ts :: \alpha \rightarrow s :: = \alpha \quad t :: f
\]
One slightly annoying wrinkle

We know that this is a valid derivational step:

\[ \alpha \]

\[ =f \alpha \quad f \]

What is the corresponding MCFG rule?

Selected thing on the right?

\[ st :: \alpha \rightarrow s :: =f \alpha \quad t :: f \]

\[
\begin{array}{c}
\text{with :: p} \\
\text{John ::}
\end{array}
\]

\[
\begin{array}{c}
\text{with :: =d p} \\
\text{John :: d}
\end{array}
\]

Selected thing on the left?

\[ ts :: \alpha \rightarrow s :: =f \alpha \quad t :: f \]

\[
\begin{array}{c}
\text{John ::} \\
\text{with :: =d p} \\
\text{John :: d}
\end{array}
\]

\[
\begin{array}{c}
\text{eat :: =d v} \\
\text{cake ::}
\end{array}
\]
One slightly annoying wrinkle

Each type needs to record not only the unchecked features, but also whether the expression is lexical.

I’ll write lexical types as $\langle \ldots \rangle_1$ and non-lexical types as $\langle \ldots \rangle_0$.

So types of the form $\langle =f \alpha \rangle_1$ act slightly differently from those of the form $\langle =f \alpha \rangle_0$.

\[
\begin{align*}
st :: \langle \alpha \rangle_0 & \rightarrow \quad s :: \langle =f \alpha \rangle_1 & t :: \langle f \rangle_n \\
with \ John :: \langle p \rangle_0 & \rightarrow \quad with :: \langle =d p \rangle_1 & John :: \langle d \rangle_1
\end{align*}
\]

\[
\begin{align*}
\begin{align*}
ts :: \langle \alpha \rangle_0 & \rightarrow \quad s :: \langle =f \alpha \rangle_0 & t :: \langle f \rangle_n \\
John\ eat\ cake :: \langle v \rangle_0 & \rightarrow \quad eat\ cake :: \langle =d v \rangle_0 & John :: \langle d \rangle_1
\end{align*}
\]
Context-free structure

Schemas for **MERGE** steps:

\[
\langle \gamma, \alpha_1, \ldots, \alpha_j, \beta_1, \ldots, \beta_k \rangle \rightarrow \langle =f \gamma, \alpha_1, \ldots, \alpha_j \rangle \quad \langle f, \beta_1, \ldots, \beta_k \rangle \\
\langle \gamma, \alpha_1, \ldots, \alpha_j, \delta, \beta_1, \ldots, \beta_k \rangle \rightarrow \langle =f \gamma, \alpha_1, \ldots, \alpha_j \rangle \quad \langle f \delta, \beta_1, \ldots, \beta_k \rangle
\]

Schemas for **MOVE** steps:

\[
\langle \gamma, \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k \rangle \rightarrow \langle +f \gamma, \alpha_1, \ldots, \alpha_{i-1}, -f, \alpha_{i+1}, \ldots, \alpha_k \rangle \\
\langle \gamma, \alpha_1, \ldots, \alpha_{i-1}, \delta, \alpha_{i+1}, \ldots, \alpha_k \rangle \rightarrow \langle +f \gamma, \alpha_1, \ldots, \alpha_{i-1}, -f \delta, \alpha_{i+1}, \ldots, \alpha_k \rangle
\]
Context-free structure

Schemas for MERGE steps:

\[
\langle \gamma, \alpha_1, \ldots, \alpha_j, \beta_1, \ldots, \beta_k \rangle \rightarrow \langle f \gamma, \alpha_1, \ldots, \alpha_j \rangle \langle f, \beta_1, \ldots, \beta_k \rangle
\]

\[
\langle \gamma, \alpha_1, \ldots, \alpha_j, \delta, \beta_1, \ldots, \beta_k \rangle \rightarrow \langle f \gamma, \alpha_1, \ldots, \alpha_j \rangle \langle f \delta, \beta_1, \ldots, \beta_k \rangle
\]

Schemas for MOVE steps:

\[
\langle \gamma, \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k \rangle \rightarrow \langle +f \gamma, \alpha_1, \ldots, \alpha_{i-1}, -f, \alpha_{i+1}, \ldots, \alpha_k \rangle
\]

\[
\langle \gamma, \alpha_1, \ldots, \alpha_{i-1}, \delta, \alpha_{i+1}, \ldots, \alpha_k \rangle \rightarrow \langle +f \gamma, \alpha_1, \ldots, \alpha_{i-1}, -f \delta, \alpha_{i+1}, \ldots, \alpha_k \rangle
\]

- MOVE steps **change** something without **combining** it with anything
- Compare with unary CFG rules, or type-raising in CCG, or ...
Three schemas for **merge** rules:

\[
\langle st, t_1, \ldots, t_k \rangle :: \langle \gamma, \alpha_1, \ldots, \alpha_k \rangle_0 \rightarrow \\
\quad s :: \langle f \gamma \rangle_1 \quad \langle t, t_1, \ldots, t_k \rangle :: \langle f, \alpha_1, \ldots, \alpha_k \rangle_n
\]

\[
\langle ts, s_1, \ldots, s_j, t_1, \ldots, t_k \rangle :: \langle \gamma, \alpha_1, \ldots, \alpha_j, \beta_1, \ldots, \beta_k \rangle_0 \rightarrow \\
\quad \langle s, s_1, \ldots, s_j \rangle :: \langle f \gamma, \alpha_1, \ldots, \alpha_j \rangle_0 \quad \langle t, t_1, \ldots, t_k \rangle :: \langle f, \beta_1, \ldots, \beta_k \rangle_n
\]

\[
\langle s, s_1, \ldots, s_j, t, t_1, \ldots, t_k \rangle :: \langle \gamma, \alpha_1, \ldots, \alpha_j, \delta, \beta_1, \ldots, \beta_k \rangle_0 \rightarrow \\
\quad \langle s, s_1, \ldots, s_j \rangle :: \langle f \gamma, \alpha_1, \ldots, \alpha_j \rangle_n \quad \langle t, t_1, \ldots, t_k \rangle :: \langle f \delta, \beta_1, \ldots, \beta_k \rangle_n'
\]

Two schemas for **merge** rules:

\[
\langle s_i s, s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_k \rangle :: \langle \gamma, \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k \rangle_0 \rightarrow \\
\quad \langle s, s_1, \ldots, s_i, \ldots, s_k \rangle :: \langle +f \gamma, \alpha_1, \ldots, \alpha_{i-1}, -f, \alpha_{i+1}, \ldots, \alpha_k \rangle_0
\]

\[
\langle s, s_1, \ldots, s_i, \ldots, s_k \rangle :: \langle \gamma, \alpha_1, \ldots, \alpha_{i-1}, \delta, \alpha_{i+1}, \ldots, \alpha_k \rangle_0 \rightarrow \\
\quad \langle s, s_1, \ldots, s_i, \ldots, s_k \rangle :: \langle +f \gamma, \alpha_1, \ldots, \alpha_{i-1}, -f \delta, \alpha_{i+1}, \ldots, \alpha_k \rangle_0
\]


