Automatic meta-grammar extraction

Matías Guzmán Naranjo

21.02.2018, Düsseldorf

Two steps

- $\wedge$: split and cut trees into fragments
- $\vee$: rearrange fragments into meaningful groups

Splitting trees

What does not seem to work: combinatoric approaches.

- treat trees like sets.
- find the cover set of each tree (all possible decompositions)
- try to find a reasonable set of decompositions in the chaos of trillions of possible decompositions.

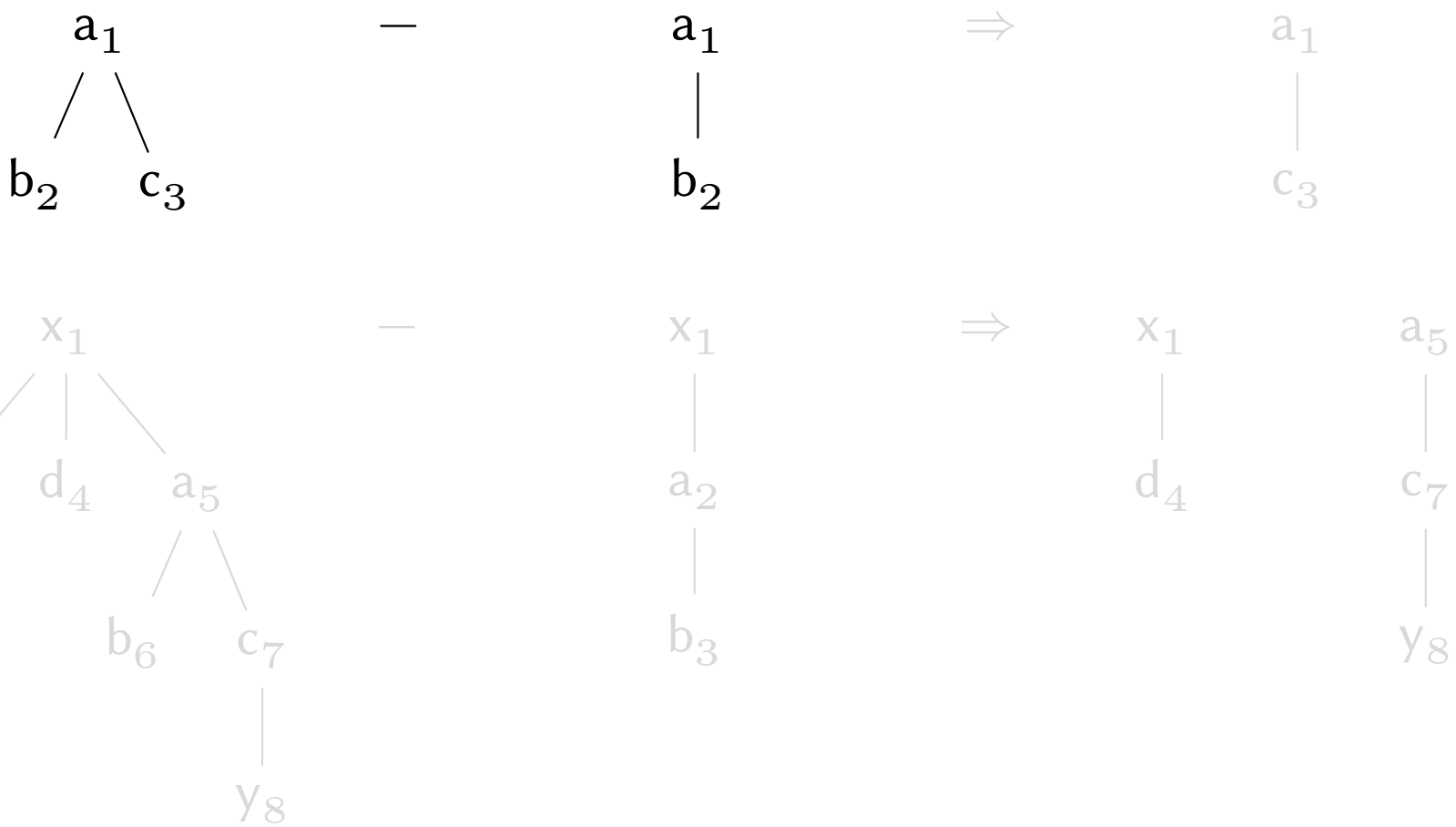
Splitting trees

What does seem to work: use small trees to cut bigger trees.

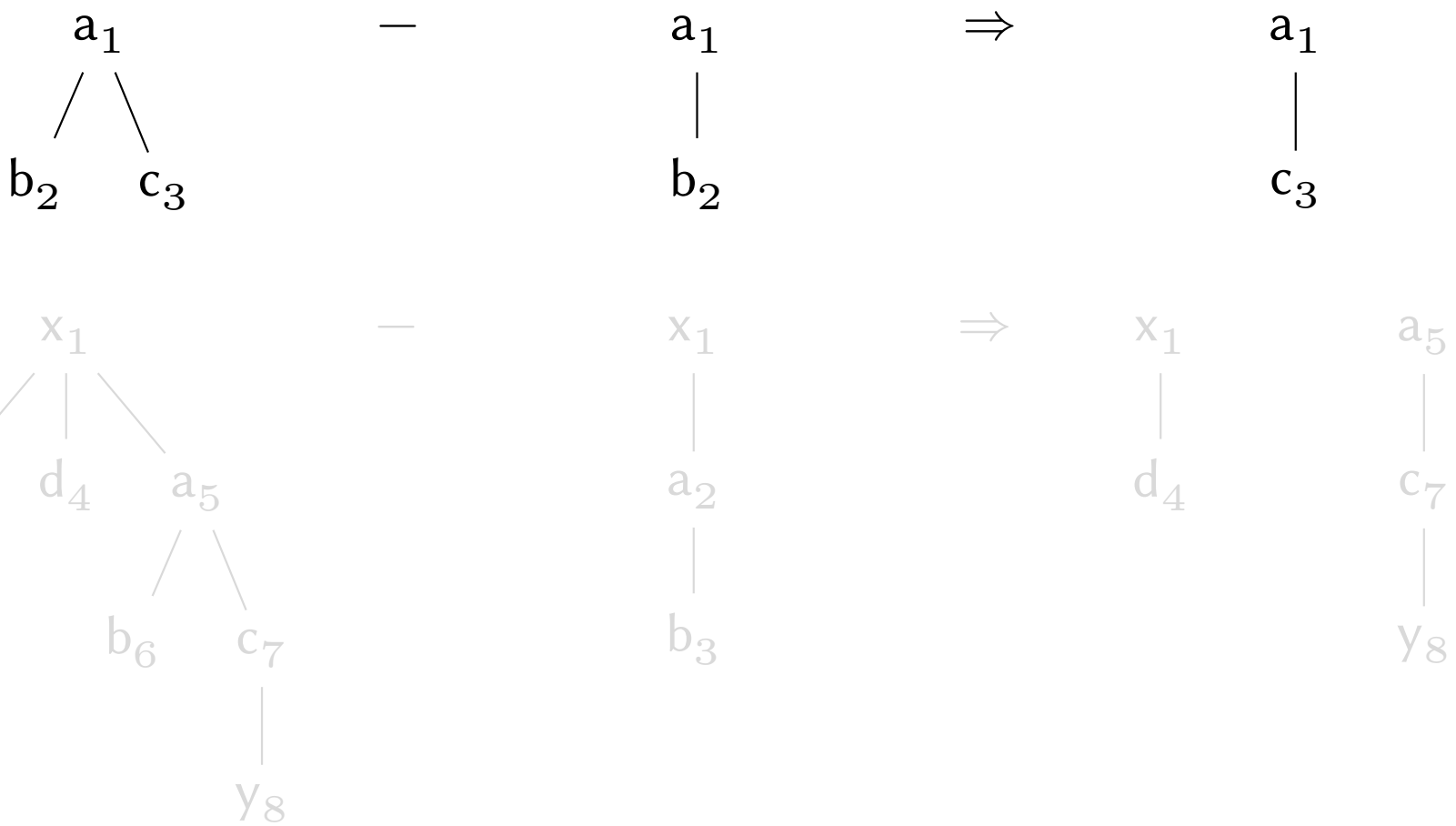
- start with a (possibly empty) set of tree fragments
- sort all trees by size (number of edges, small first)
- starting with the smallest tree, try to cut it into pieces found in the set of tree fragments (sorted from largest to smallest)
- add the 'left overs' (possibly the whole tree) to the inventory of tree fragments
- **(add the tree itself to the inventory of tree fragments)**
- repeat until you've cut all your trees

Cutting trees

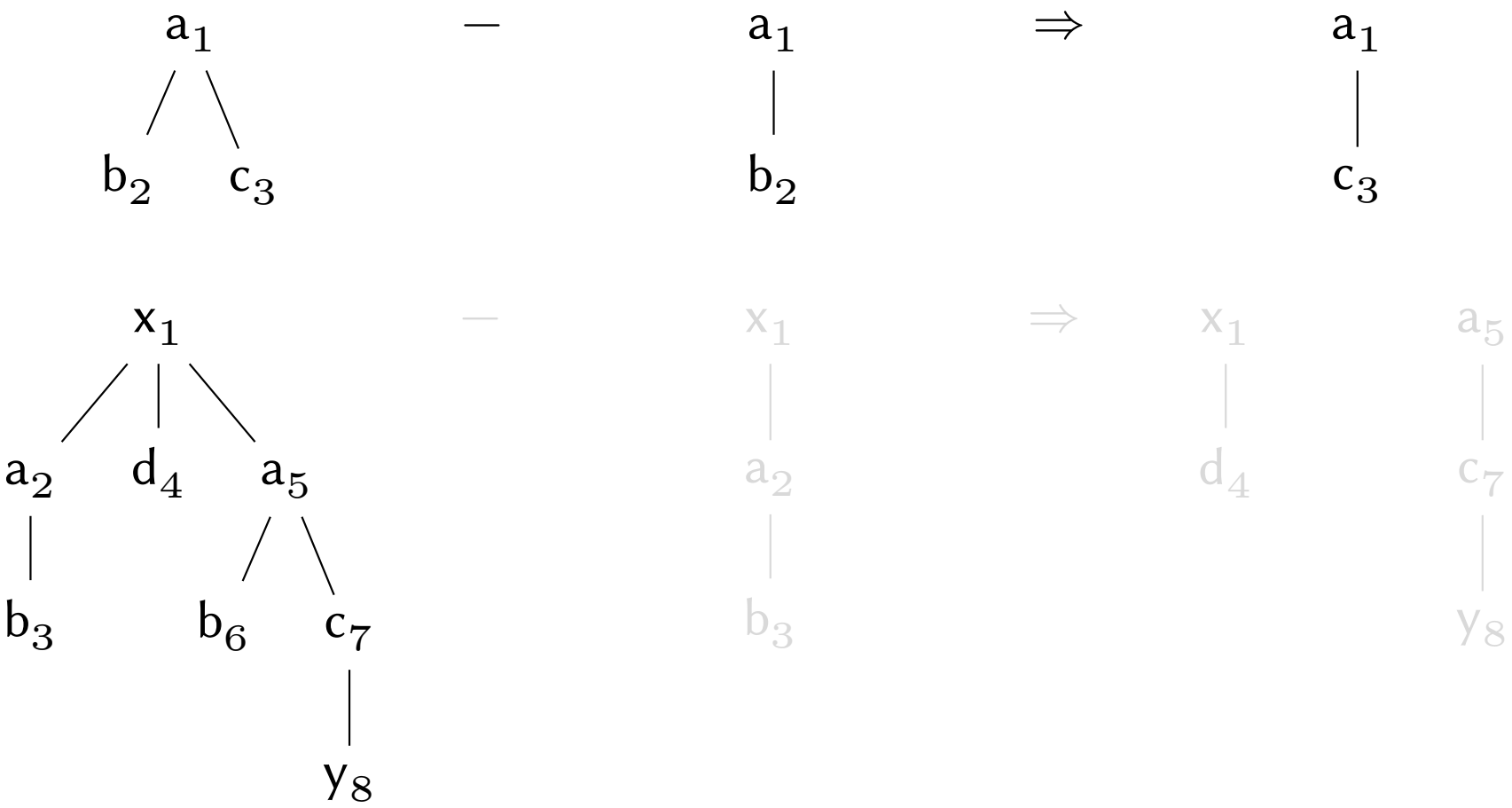
Cutting trees



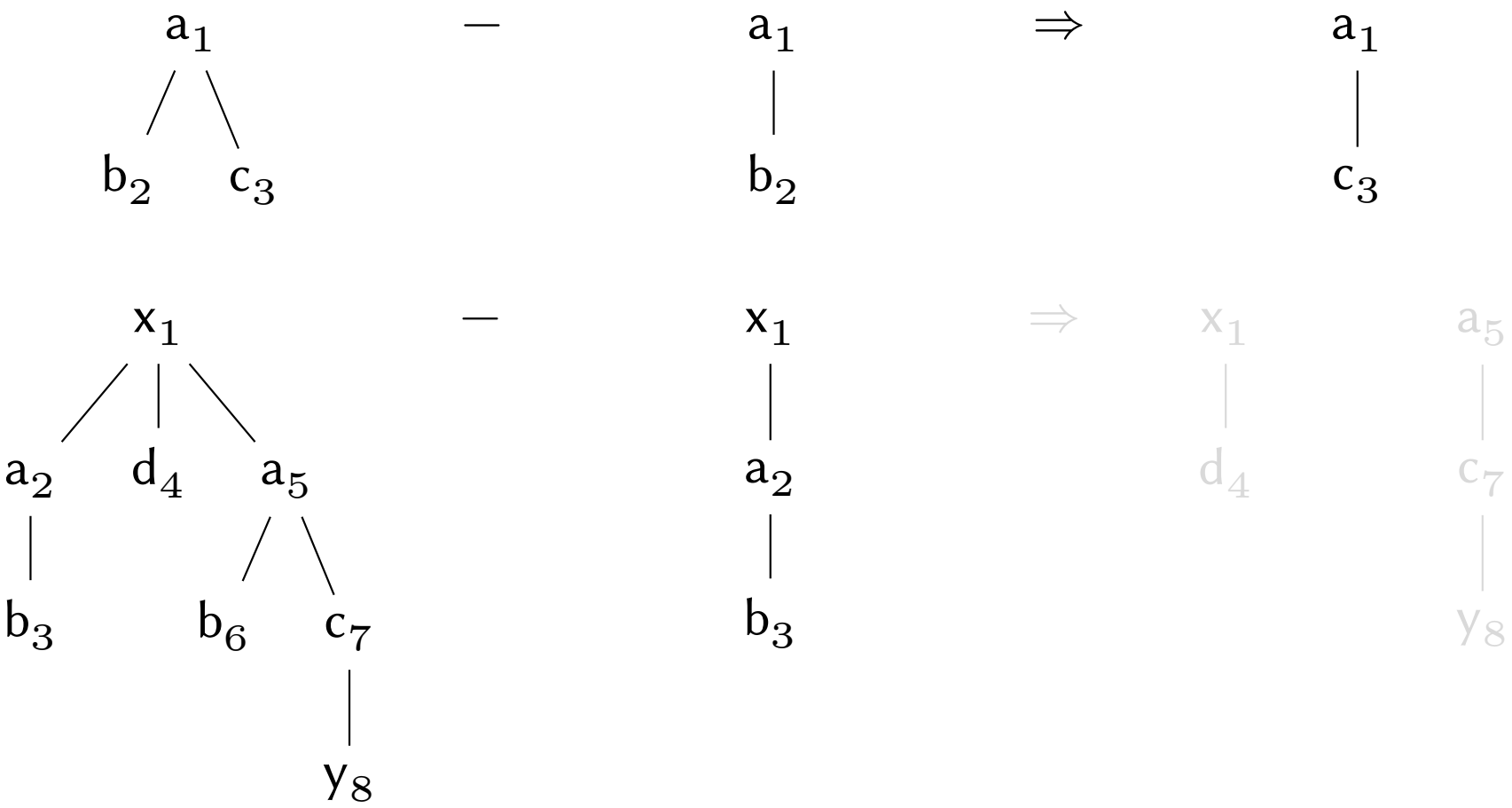
Cutting trees



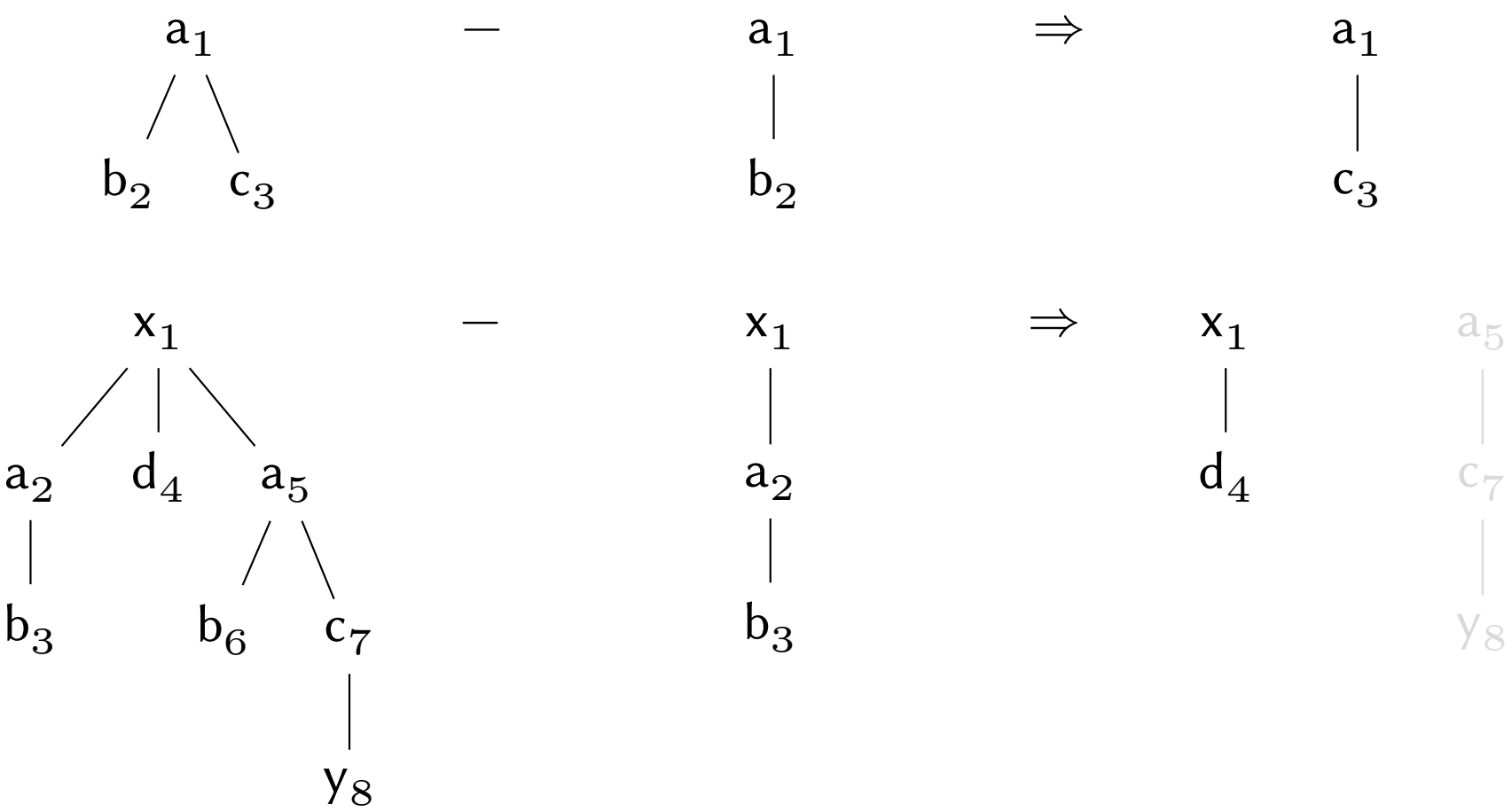
Cutting trees



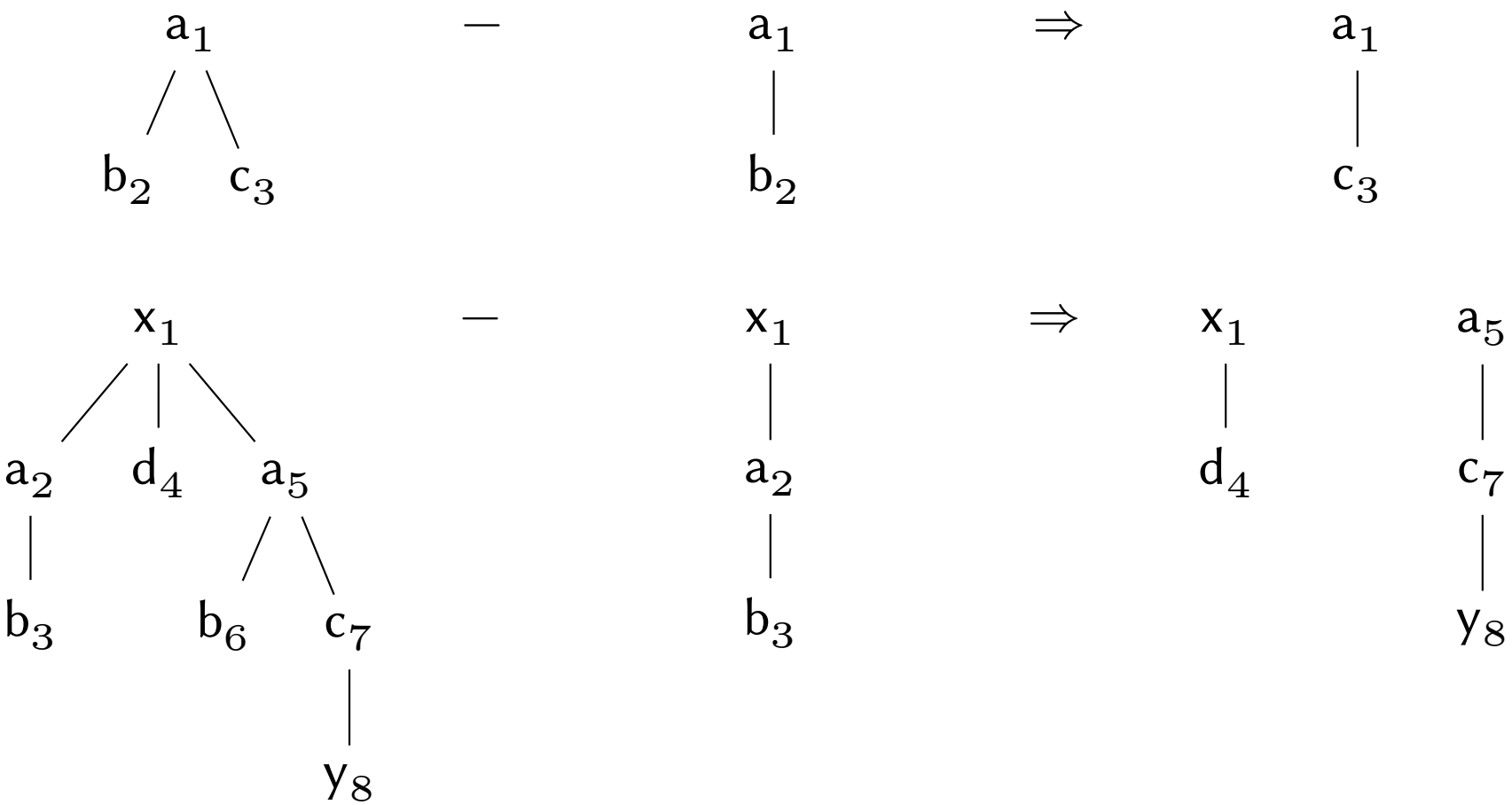
Cutting trees



Cutting trees



Cutting trees



Shortcomings

This approach has some clear shortcomings

- The decompositions ‘lose’ linear presedence
- There is no good way of naming the resulting fragments
- No abstraction across trees

Seeds

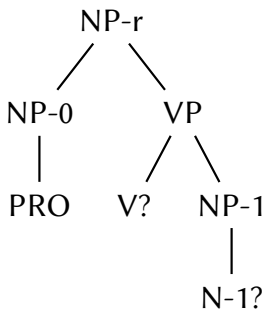
An interesting aspect of this approach is that it allows us to start the system with user defined seeds.

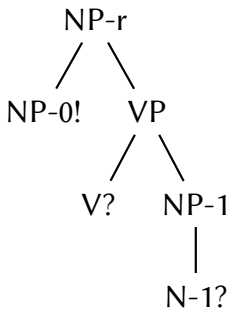
We can include things like the vp-spine as a basic fragment:

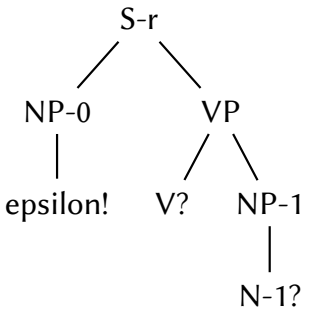
This guarantees that all trees where the vp-spine is present, will be decomposed as having this fragment.

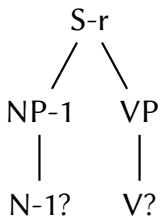
The final factorization has 719 unique fragments (for 1300 trees), but many of these are reused trees.

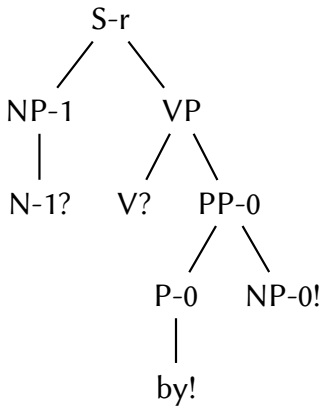
Tnx0VN1

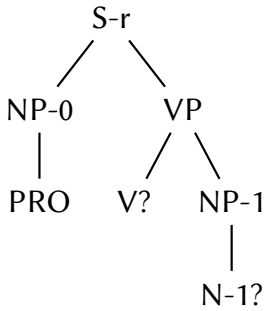

NP-r
NP-0
VP
PRO
V?
NP-1
N-1?

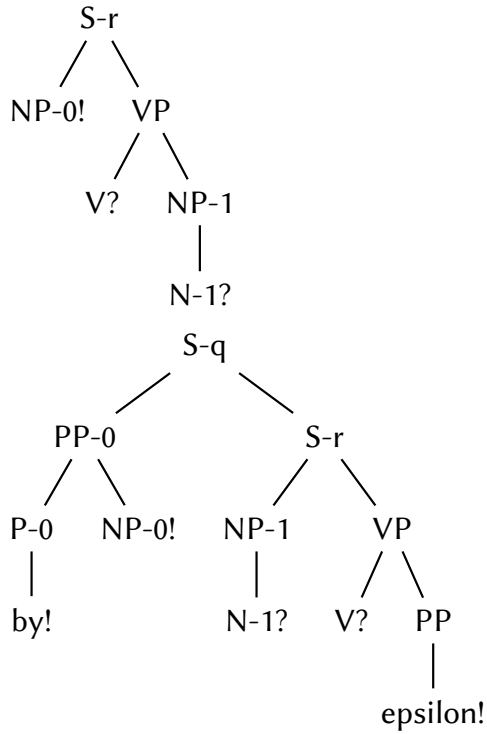

NP-r
NP-0!
VP
V?
NP-1
N-1?

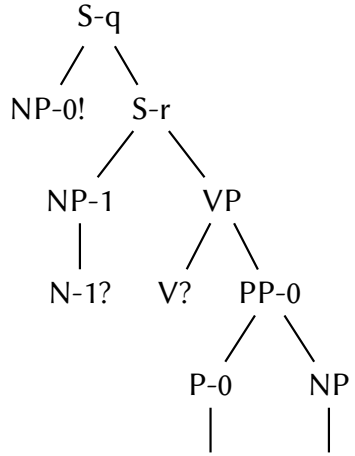

S-r
NP-0
VP
epsilon!
V?
NP-1
N-1?


S-r
NP-1
VP
N-1?
V?

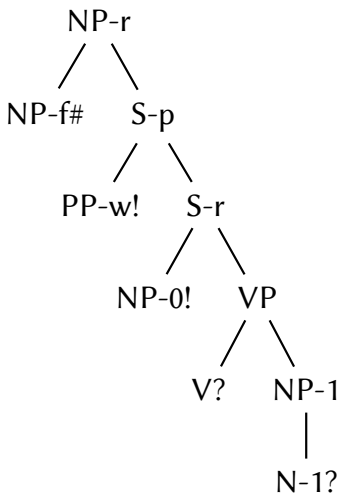

S-r
NP-1
VP
N-1?
V?
PP-0
P-0
NP-0!
by!

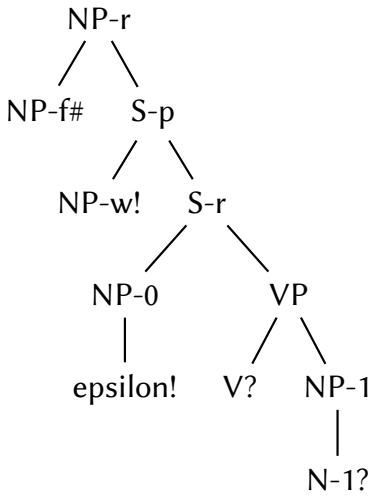

S-r
NP-0
VP
PRO
V?
NP-1
N-1?

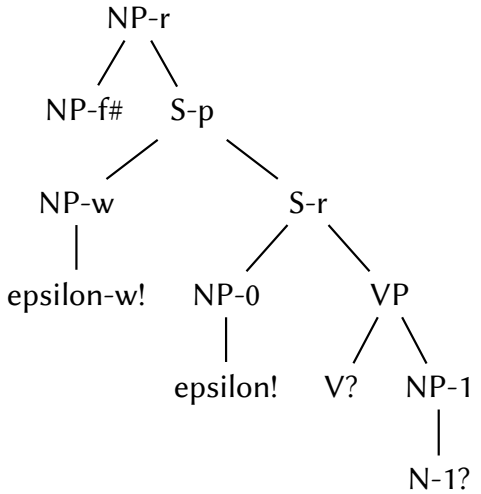

S-r
NP-0!
VP
V?
NP-1
N-1?
S-q
PP-0
S-r
P-0
NP-0!
NP-1
VP
by!
N-1?
V?
PP
epsilon!

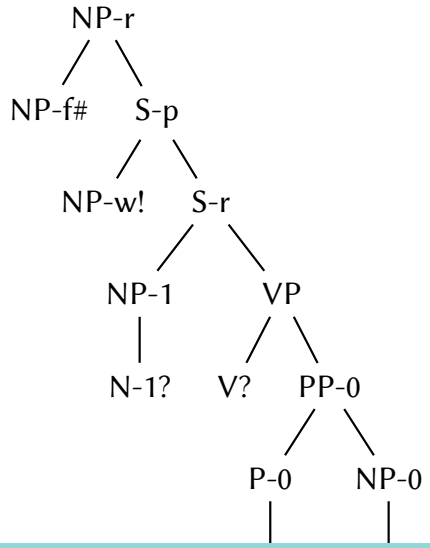

S-q
NP-0!
S-r
NP-1
VP
N-1?
V?
PP-0
P-0
NP

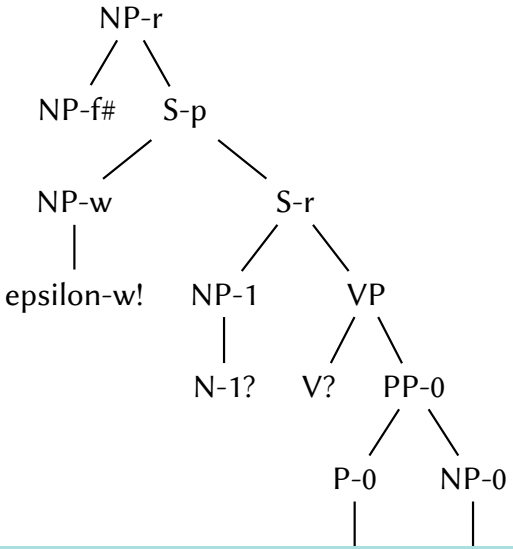
Tnx0VN1

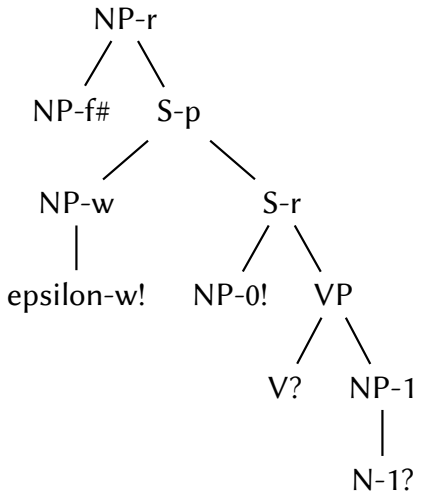

NP-r
NP-f#
S-p
PP-w!
S-r
NP-0!
VP
V?
NP-1
N-1?

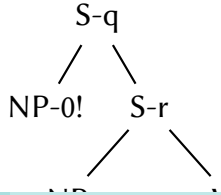

NP-r
NP-f#
S-p
NP-w!
S-r
NP-0
VP
epsilon!
V?
NP-1
N-1?


NP-r
NP-f#
S-p
NP-w
S-r
epsilon-w!
NP-0
VP
epsilon!
V?
NP-1
N-1?


NP-r
NP-f#
S-p
NP-w!
S-r
NP-1
VP
N-1?
V?
PP-0
P-0
NP-0


NP-r
NP-f#
S-p
NP-w
S-r
epsilon-w!
NP-1
VP
N-1?
V?
PP-0
P-0
NP-0


NP-r
NP-f#
S-p
NP-w
S-r
epsilon-w!
NP-0!
VP
V?
NP-1
N-1?

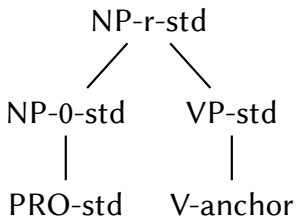

S-q
NP-0!
S-r

Tnx0VN1

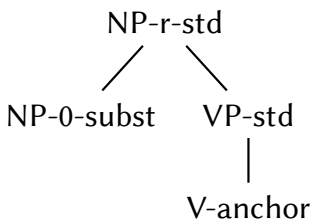
alphaGnx0VN1-PRO: **alphaGnx0V-PRO**, VP-std->NP-1-std|NP-1-std->N-1-anchor
alphaGnx0VN1: **alphaGnx0V**, VP-std->NP-1-std|NP-1-std->N-1-anchor
alphaInx0VN1: **alphaInx0V**, VP-std->NP-1-std|NP-1-std->N-1-anchor
alphaN1V: **vp-spine**, S-r-std->NP-1-std|NP-1-std->N-1-anchor
alphaN1Vbynx0: **alphaN1V**, VP-std->PP-0-std|PP-0-std->P-0-std|PP-0-std->NP-0-subst|P-0-std->by-flex6»7
alphanx0VN1-PRO: **alphanx0V-PRO**, VP-std->NP-1-std|NP-1-std->N-1-anchor
alphanx0VN1: **alphanx0V**, VP-std->NP-1-std|NP-1-std->N-1-anchor
alphapW0N1Vbynx0: S-q-std->PP-0-std|S-q-std->S-r-std|PP-0-std->P-0-std|PP-0-std->NP-0-subst|P-0-std->by-flex2»3-4»5, **alphaN1V**, VP-std->PP-std|PP-std->epsilon-flex
alphaW0N1Vbynx0: VP-std->PP-0-std|PP-0-std->P-0-std|PP-0-std->NP-std|P-0-std->by-flex|NP-std->epsilon-flex8»9, **alphaN1V**, S-q-std->NP-0-subst|S-q-std->S-r-std2»3
alphaW0nx0VN1: **alphaW0nx0V**, VP-std->NP-1-std|NP-1-std->N-1-anchor
betaN0N1Vbynx0: VP-std->PP-0-std|PP-0-std->P-0-std|PP-0-std->NP-0-std|P-0-std->by-flex|NP-0-std->epsilon-flex10»11, **alphaN1V**, NP-r-std->NP-f-foot|NP-r-std->S-p-std|S-p-std->NP-w-subst|S-p-std->S-r-std2»3-4»5
betaN0nx0VN1: **betaN0nx0V**, VP-std->NP-1-std|NP-1-std->N-1-anchor
betaNc0N1Vbynx0: NP-r-std->NP-f-foot|NP-r-std->S-p-std|S-p-std->NP-w-std|S-p-std->S-r-std|NP-w-std->epsilon-w-flex2»3-4»5, VP-std->PP-0-std|PP-0-std->P-0-std|PP-0-std->NP-0-std|P-0-std->by-flex|NP-0-std->epsilon-flex10»11, **alphaN1V**
betaNc0nx0VN1: **betaNc0nx0V**, VP-std->NP-1-std|NP-1-std->N-1-anchor
betaNcnx0VN1: **betaNcnx0V**, VP-std->NP-1-std|NP-1-std->N-1-anchor
betaNpxnx0VN1: **betaNpxnx0V**, VP-std->NP-1-std|NP-1-std->N-1-anchor

Tnx0VN1 fragments

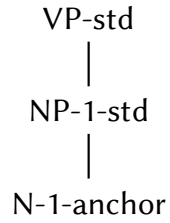
alphaGnx0V-PRO



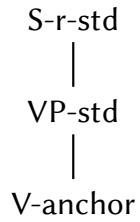
alphaGnx0V



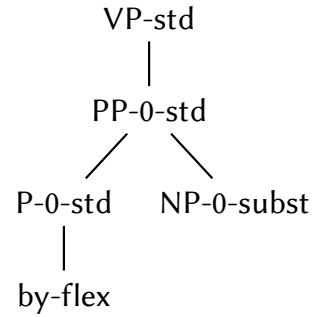
NN



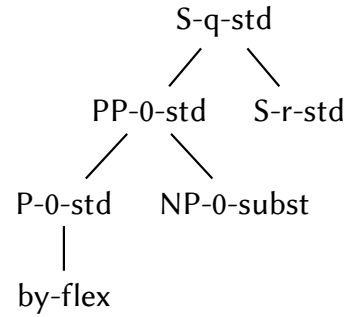
vp-spine

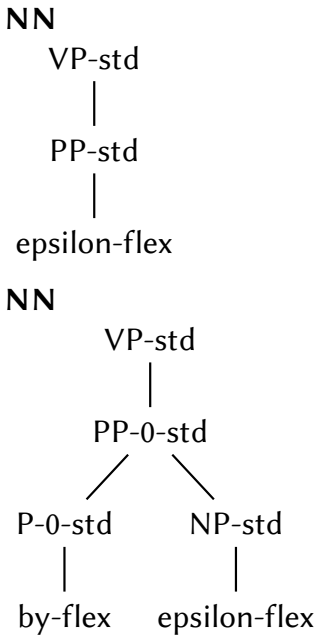


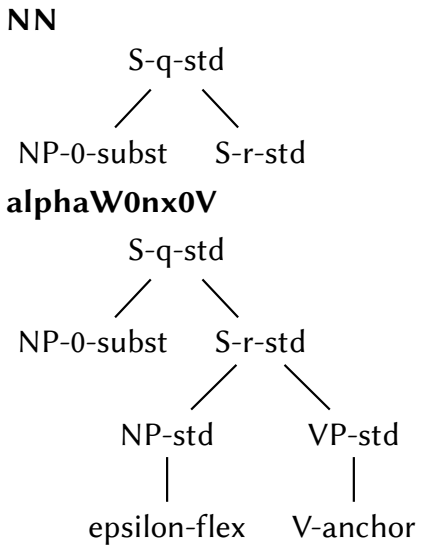
NN

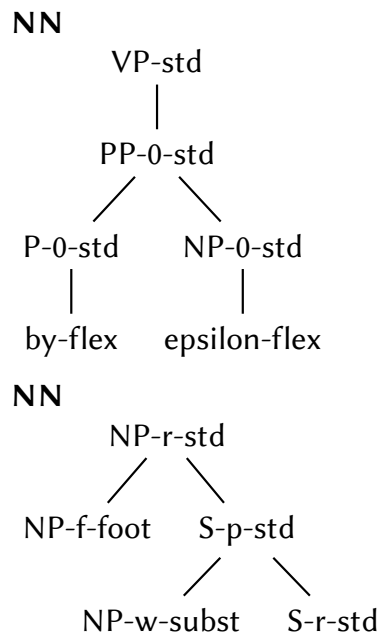


NN

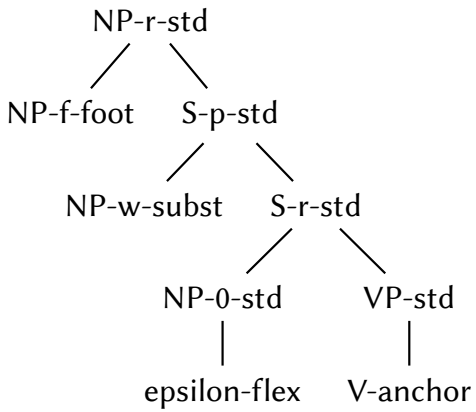



NN
VP-std
PP-std
epsilon-flex
NN
VP-std
PP-0-std
P-0-std
NP-std
by-flex
epsilon-flex

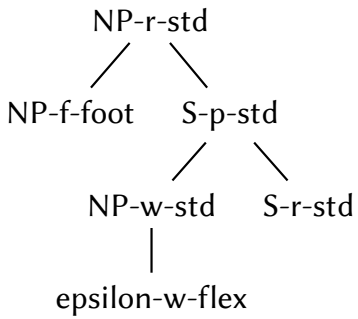

NN
S-q-std
NP-0-subst
S-r-std
alphaW0nx0V
S-q-std
NP-0-subst
S-r-std
NP-std
VP-std
epsilon-flex
V-anchor


NN
VP-std
PP-0-std
P-0-std
NP-0-std
by-flex
epsilon-flex
NN
NP-r-std
NP-f-foot
S-p-std
NP-w-subst
S-r-std

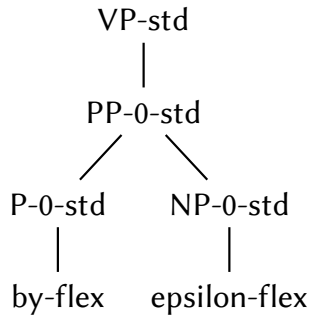
betaN0nx0V



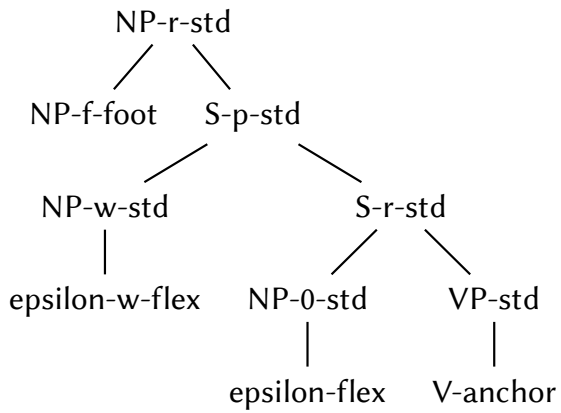
NN



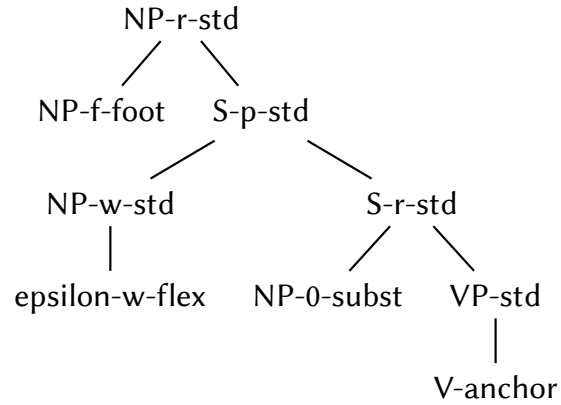
NN



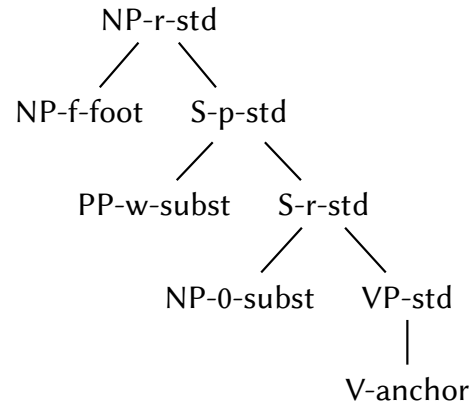
betaNc0nx0V



betaNcnx0V



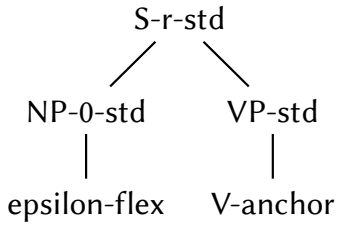
betaNpxnx0V



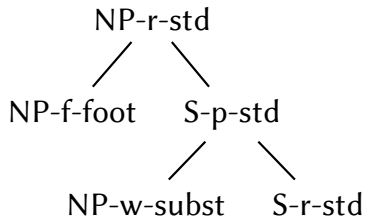
The biger pices are themselves factorized

So, just to show you, in the previous slide the fragment **betaN0nx0V** gets factorized as:

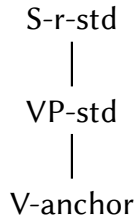
alphaInx0V



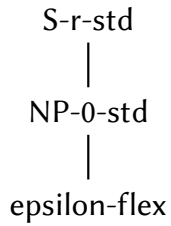
NN



Vspine



NN



Recompiling

When this factorization is exported back to xmg we get back almost all of the original trees (there are still some mistakes).

however:

- We have to explicitly define each tree
- add all missing node ordering constraints which are lost in the decomposition
- There are no $\vee$ relations (!)

∨

How to proceed?

Suppose the following tree decompositions:

- t1 = (a, b, c, d)
- t2 = (a, b, c, e)

Trivially, we can see that $a \wedge b \wedge c \wedge (d \vee e)$ produces both trees.

But if we have:

- t1 = (a, b, c, d)
- t2 = (a, b, c, e)
- t3 = (a, f, c, e)
- t4 = (a, f, c, d)
- t5 = (g, b, c, e)

What to do?!

In the intransitive family

alphanx0V-PRO	⇒	(ae, ag)
betaNc0nx0V	⇒	(**al**, aj)
betaVintransn	⇒	(ao)
alphaGnx0V-PRO	⇒	(ac)
alphaDnx0V	⇒	(ab)
alphaW0nx0V	⇒	(ae, ai)
alphaInx0V	⇒	(ae, af)
alphanx0V	⇒	(ae, ah)
betaNcnx0V	⇒	(**al**, **am**)
betaN0nx0V	⇒	(aj, ak)
betaNpxnx0V	⇒	(an, **am**)
alphaGnx0V	⇒	(ad)

In the transitive family

betaN0nx1Vbynx0	⇒	az, ap, bc
betaN0nx0Vnx1	⇒	ay, ag
alphanx1V-PRO	⇒	ao
alphaW0nx0Vnx1	⇒	as, ag
alphaW0nx1Vbynx0	⇒	ap, at, au
alphapW0nx1Vbynx0	⇒	ap, aq, ar
alphanx1Vbynx0-PRO	⇒	ao, ak
betaNc0nx0Vnx1	⇒	bh, ag
betaNcnx1Vbynx0	⇒	bi, ak
alphanx1Vbynx0	⇒	ak, ap
alphaInx0Vnx1	⇒	al, ag
alphanx0Vnx1	⇒	an, ag
betaN1nx1Vbynx0	⇒	be, ak
betaNpxnx0Vnx1	⇒	bl, ag
betaNc1nx0Vnx1	⇒	bj, bd
betaNpxnx1V	⇒	bm
alphaW1nx1Vbynx0	⇒	ax, ak
betaNc1nx1Vbynx0	⇒	bk, ak
betaN1nx1V	⇒	be

betaNbynx0nx1Vbynx0	⇒	ap, bf, bg
alphanx0Vnx1-PRO	⇒	am, ag
alphaAV	⇒	ab
alphaW1nx1V	⇒	ax
alphaW1nx0Vnx1	⇒	an, av, aw
alphaGnx1Vbynx0-PRO	⇒	ai, ak
betaN1nx0Vnx1	⇒	az, an, bd
alphaGnx0Vnx1	⇒	ah, ag
alphaGnx1V	⇒	aj
alphaGnx1V-PRO	⇒	ai
betaNc1nx1V	⇒	bk
alphaDnx0Vnx1	⇒	ac, ad, ae
betaNcnx1V	⇒	bi
alphaGnx1Vbynx0	⇒	aj, ak
alphaGnx0Vnx1-PRO	⇒	af, ag
betaVtransn	⇒	bn
betaNc0nx1Vbynx0	⇒	bi, bc
betaNcnx0Vnx1	⇒	bj, ag
alphanx1V	⇒	ap
betaNpxnx1Vbynx0	⇒	bm, ak

Fragment frequency

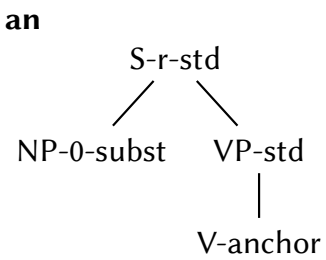
ag	⇒	10
ak	⇒	9
ap	⇒	6
bi	⇒	3
an	⇒	3
bd	⇒	2
aj	⇒	2
be	⇒	2
ao	⇒	2
bm	⇒	2
az	⇒	2
ax	⇒	2
ai	⇒	2
bc	⇒	2
bk	⇒	2
bj	⇒	2

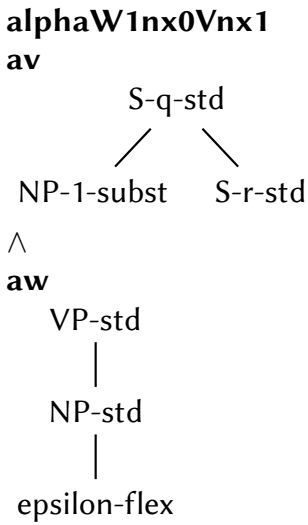
A trivial ∨ factorization

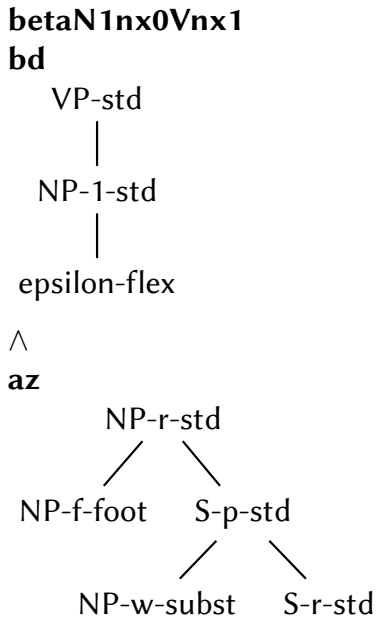
ak ∧ (ao ∨ bi ∨ ap ∨ be ∨ ax ∨ bk ∨ ai ∨ aj ∨ bm)
ap ∧ ((bc ∧ az) ∨ (at ∧ au) ∨ (ar ∧ aq) ∨ (bg ∧ bf))
ag ∧ (ay ∨ as ∨ bh ∨ al ∨ an ∨ bl ∨ am ∨ ah ∨ af ∨ bj)
an ∧ ((av ∧ aw) ∨ (bd ∧ az))

an ∧ ((av ∧ aw) ∨ (bd ∧ az))

⇒ alphaW1nx0Vnx1, betaN1nx0Vnx1







Many problems with this approach:

- We see that what both trees have in common is that they both have a subject
- Other trees with subjects are being ignored because they use larger trees as fragments
- Not very interesting linguistically (?)

Some alternatives (not tried so far)

- Try to infer ‘families’ with some distance based clustering
- (dis)Prefer more frequent trees as pivots for the ⋁s
- Use features to only group together fragments which are identical.
- All of the above (?)

Factorizing feature structures

We take a similar approach (maybe?)
For a single fragment type (e.g. all cases of the vp-spine):

- Find the fragment instances with largest feature structures that are a subset of the feature structures of other fragment instances
- Decompose those fragment instance as being made up of two feature structures
- repeat until done

Problems

This has a series of problems, however:

- It is not clear that the decompositions make much sense (though maybe?)
- Decompositions seem to remain mostly binary
- Decompositions seem to mostly be ((ft1) (features))
- There is no natural way to sort the porcess, thus many different possible results.
- Maybe the feature decomposition should be linked is some other way to the tree splitting process?

That's it...