CHAPTER SIX: SCOPE, ABSTRACTION, COMPOSITION

6.1 Scope ambiguities, quantifying-in and storage.

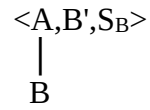
We will now add a mechanism for creating scope ambiguities.
Following Cooper 1975, 1983, the mechanism we will use will be a storage mechanism.
We will first introduce the basic mechanism and show its working in several examples.
Up to now our grammar generates pairs of syntactic structures and semantic representation. We will extend this now.

The grammar will now generate triple <A,A',S> where:

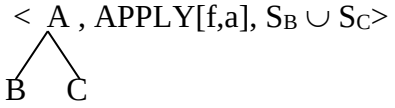
-A is a syntactic tree.
-A', the translation of A, is an IL expression.
-S, **the quantifier store**, is a set of pairs <n,α> where n is an index and α a TY_2 expression.

The quantifier store will be used for creating scope ambiguities.
We will assume that for lexical entries the quantifier store is empty.

We assume that stores are inherited upward in the following way:



For unary branching, both the meaning and the store inherit up.



For binary branching, the stores corresponding to the daughters are united.

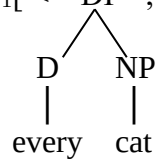
We now introduce two operations $STORE_n$ and $RETRIEVE_n$

$$STORE_n[<DP, DP', S_{DP}>] = <DP, x_n, S_{DP} \cup \{<n,DP'>\}>$$

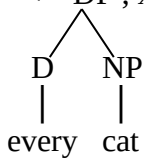
$STORE_n$ replaces the interpretation DP' of the DP by a variable x_n (of type e), and stores the interpretation DP' under index n in the quantifier store.
Thus, $STORE_n$ tells you to use a variable x_n for the time being as the interpretation of the DP, instead of the translation that we have just stored.

Example:

$STORE_1[<$ DP , $\lambda P.\forall x[CAT_w(x) \rightarrow P(x)], \emptyset >] =$



$<$ DP , x_1, $\{<1,\lambda P.\forall x[CAT_w(x) \rightarrow P(x)]>\} >$



Our second new rule is **quantifying in**, $RETRIEVE_n$.

> Let <IP, IP', S_{IP}> be a triple, with IP a syntactic tree, IP' its interpretation, S_{IP} the corresponding store.
> Let $<n,\alpha> \in S_{IP}$, and assume that no other interpretation is stored under index n in S_{IP}.
>
> $RETRIEVE_n[<IP, IP', S_{IP}>] = <IP, APPLY[\lambda x_n.IP',\alpha], S_{IP}-\{<n,\alpha>\}>$

$RETRIEVE_n$ turns the IP meaning IP'of type t into a **predicate** $\lambda x_n.IP'$ of type <e,t>, by abstracting over variable x_n of type e.
It takes the DP meaning α, which was stored under index n, out of the quantifier store S_{IP} and it applies the predicate $\lambda x_n.IP'$ to α with APPLY.

We assume that we always want to end up with an empty store.

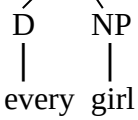
With this mechanism, we are able to derive translations of sentences where the meaning of an NP takes wider semantic scope than its syntactic scope.

(1) Every girl hugs a cat.

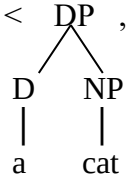
I will use this example to demonstrate the effects of the scope mechanism.
This sentence has five types of derivations. I will go through all five, and show that as a result, the sentence will be generated by the grammar with two meanings:
one where the meaning of *every girl* takes scope over the meaning of *a cat*,
and one where the meaning of *a cat* takes scope over the meaning of *every girl*.

All derivations start out with the same structure, interpretation and store for *every girl* and *some cat*:

< DP , $\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$, Ø >



< DP , $\lambda P \exists z[CAT_w(z) \wedge P(z)]$, Ø >



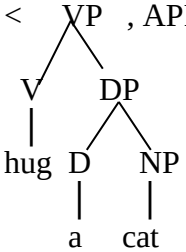
(1) Every girl hugs a cat.
DERIVATION 1

From the lexical item *hug* we generate:

<V, HUG_w, Ø >

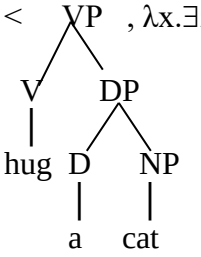
The V takes the DP as a complement, we apply the V interpretation to the interpretation of [DP a cat], and we unite the stores (giving Ø). This gives:

< VP , APPLY[HUG_w, $\lambda P.\exists z[CAT_w(z) \wedge P(z)]$] , Ø >



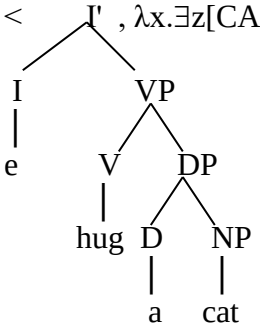
$APPLY[HUG_w, \lambda P.\exists z[CAT_w(z) \wedge P(z)]]$
$= LIFT[HUG_w]\ (\lambda P.\exists z[CAT_w(z) \wedge P(z)])$ [def APPLY]
$= [\lambda T \lambda x.T(\lambda y.HUG_w(x,y))](\lambda P.\exists z[CAT_w(z) \wedge P(z)])$ [def LIFT + rel notation]
$= \lambda x.[\ \lambda P.\exists z[CAT_w(z) \wedge P(z)](\lambda y.HUG_w(x,y))\ ]$ [λ-con]
$= \lambda x.\exists z[CAT_w(z) \wedge (\lambda y.HUG_w(x,y)(z)]$ [λ-con]
$= \lambda x.\exists z[CAT_w(z) \wedge HUG_w(x,z)]$ [λ-con]

So we get:

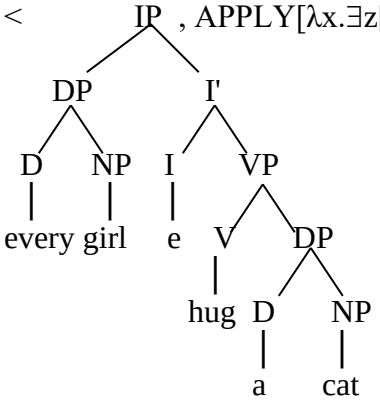


We have: <I, λP.P, Ø >

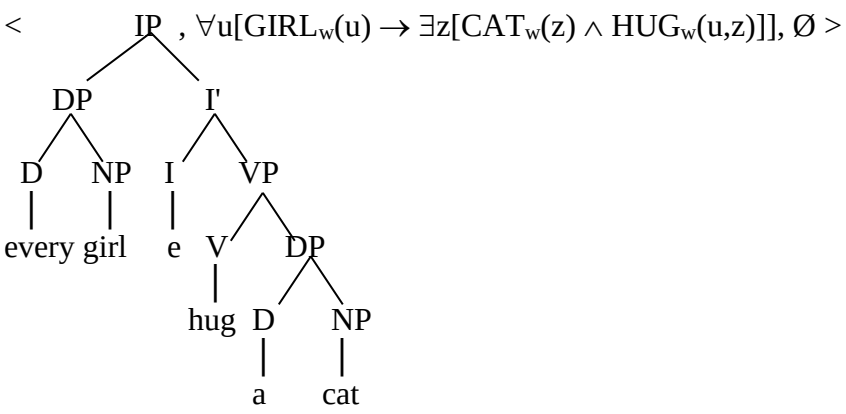
The I takes the VP as a complement, and we get:



The I' takes [$_{DP}$ every girl] as a specifier, and we get:



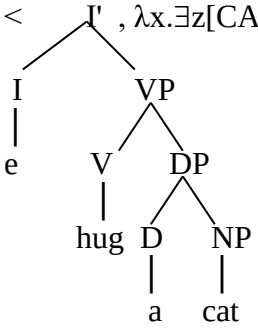
APPLY[$\lambda x.\exists z[CAT_w(z) \wedge HUG_w(x,z)]$, $\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$]
= LIFT[$\lambda x.\exists z[CAT_w(z) \wedge HUG_w(x,z)]$] ($\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$) [def APPLY]
= $\lambda T.T(\lambda x.\exists z[CAT_w(z) \wedge HUG_w(x,z)])$ ($\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$) [def LIFT]
= $\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$ ($\lambda x.\exists z[CAT_w(z) \wedge HUG_w(x,z)]$) [λ-con]
= $\forall u[GIRL_w(u) \rightarrow [\lambda x.\exists z[CAT_w(z) \wedge HUG_w(x,z)]](u)]$ [λ-con]
= $\forall u[GIRL_w(u) \rightarrow \exists z[CAT_w(z) \wedge HUG_w(u,z)]$ [λ-con]
So we get:



(1) Every girl hugs a cat.
DERIVATION 2

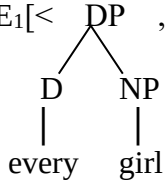
We build up the same I' as under derivation 1:

< I' , $\lambda x.\exists z[CAT_w(z) \wedge HUG_w(x,z)]$, Ø >

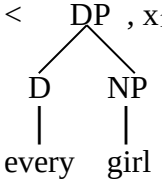


We apply $STORE_1$ to *every girl*:

$STORE_1$[< DP , $\lambda P.\forall u[GIRL_w(u) \rightarrow P(u)]$, Ø >] =

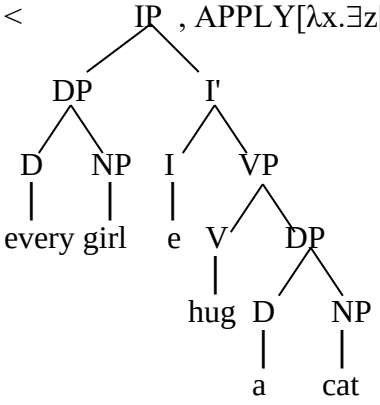


< DP , x_1, {<1,$\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$>} >



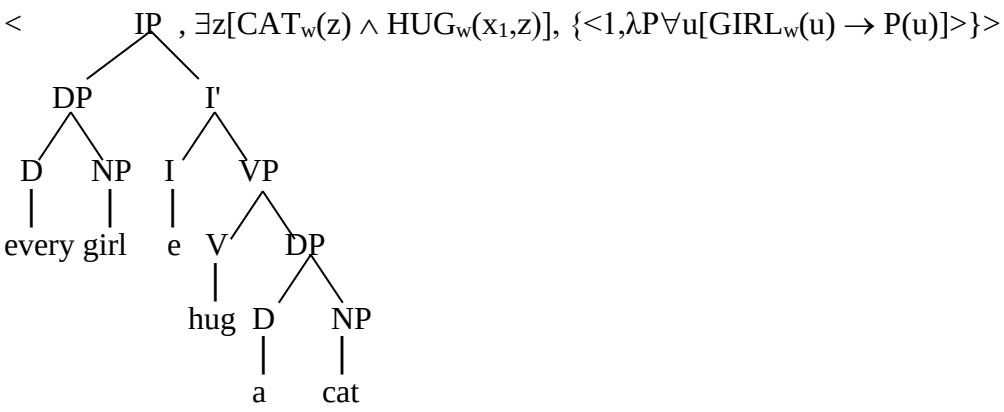
The I' takes this DP as a specifier, applies I' interpretation to the interpretation of the DP, which is x_1, and unites the stores. We get:

< IP , APPLY[$\lambda x.\exists z[CAT_w(z) \wedge HUG_w(x,z)]$, x_1], {<1,$\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$>}>

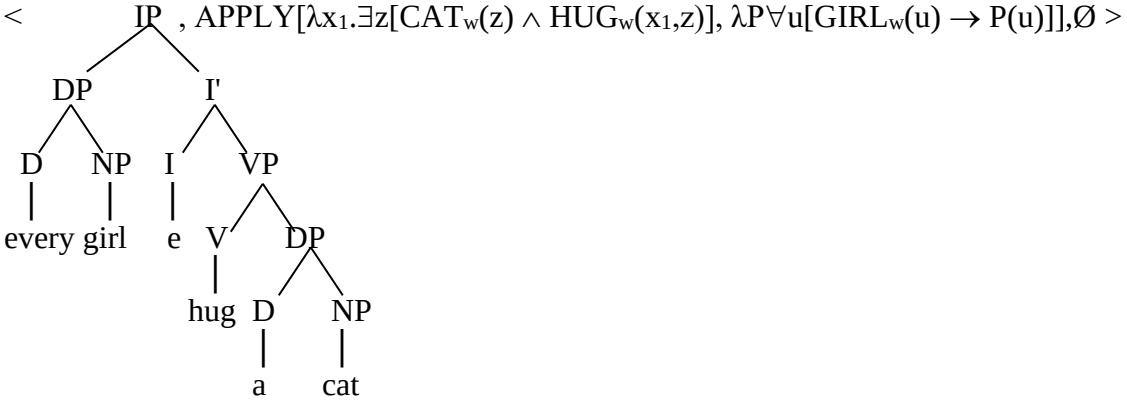


$\text{APPLY}[\lambda x.\exists z[\text{CAT}_w(z) \wedge \text{HUG}_w(x,z)],x_1]$
$= \lambda x.\exists z[\text{CAT}_w(z) \wedge \text{HUG}_w(x,z)]\ (x_1)$ [def APPLY]
$= \exists z[\text{CAT}_w(z) \wedge \text{HUG}_w(x_1,z)]$ [λ-con]

So we get:

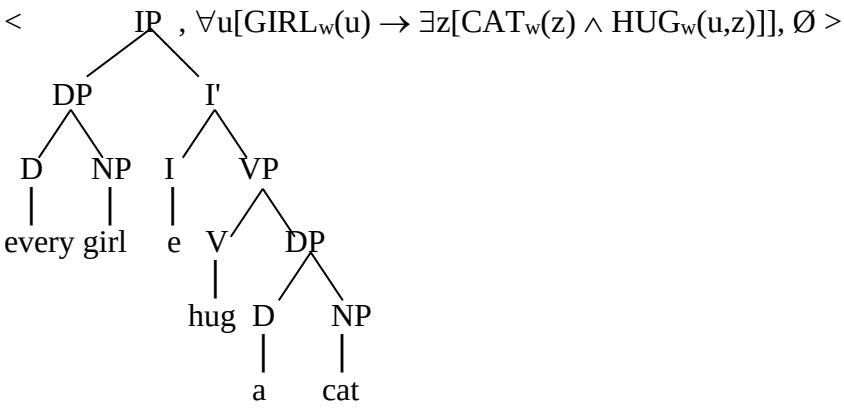


To this we apply RETRIEVE_1, which gives:



$\text{APPLY}[\lambda x_1.\exists z[\text{CAT}_w(z) \wedge \text{HUG}_w(x_1,z)], \lambda P\forall u[\text{GIRL}_w(u) \rightarrow P(u)]]$
$=\text{LIFT}[\lambda x_1.\exists z[w\ (z) \wedge w(x_1,z)]]\ (\lambda P\forall u[\text{GIRL}_w(u) \rightarrow P(u)])$ [def APPLY]
$=\lambda T.T(\lambda x_1.\exists z[\text{CAT}_w(z) \wedge \text{HUG}_w(x_1,z)])\ (\lambda P\forall u[\text{GIRL}_w(u) \rightarrow P(u)])$ [def LIFT]
$= [\lambda P\forall u[\text{GIRL}_w(u) \rightarrow P(u)]]\ (\lambda x_1.\exists z[\text{CAT}_w(z) \wedge \text{HUG}_w(x_1,z)])$ [λ-con]
$= \forall u[\text{GIRL}_w(u) \rightarrow [\lambda x_1.\exists z[\text{CAT}_w(z) \wedge \text{HUG}_w(x_1,z)]](u)]$ [λ-con]
$= \forall u[\text{GIRL}_w(u) \rightarrow \exists z[\text{CAT}_w(z) \wedge \text{HUG}_w(u,z)]]$ [λ-con]

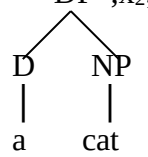
So we get the same as in derivation 1:



(1) Every girl hugs a cat.
DERIVATION 3

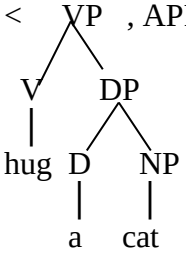
This time we start by applying STORE$_2$ to *a cat* and get:

< DP ,x_2, {<2,λP∃z[CAT$_w$(z) ∧ P(z)]>} >



The V *hug* takes this as a complement, and we get:

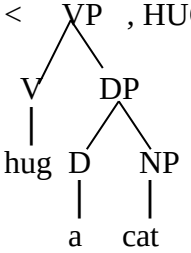
< VP , APPLY[HUG$_w$,x_2], {<2,λP.∃z[CAT$_w$(z) ∧ P(z)]>} >



APPLY[HUG,x_2]
= HUG(x_2) [def APPLY]

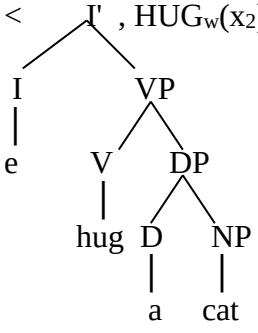
So we get:

< VP , HUG$_w$(x_2), {<2,λP.∃z[CAT$_w$(z) ∧ P(z)]>} >



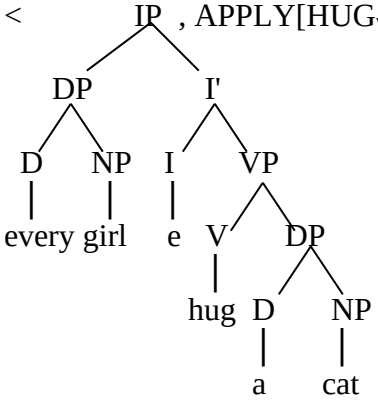
The empty I takes this as a complement, and we get:

< I' , HUG$_w$(x_2), {<2,λP.∃z[CAT$_w$(z) ∧ P(z)]>} >



This I' takes *every girl* as a specifier, and we get:

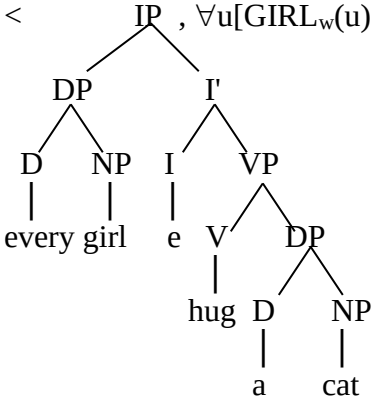
< IP , APPLY[$HUG_w(x_2)$,$\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$], {<2,$\lambda P.\exists z[CAT_w(z) \wedge P(z)]$>}>



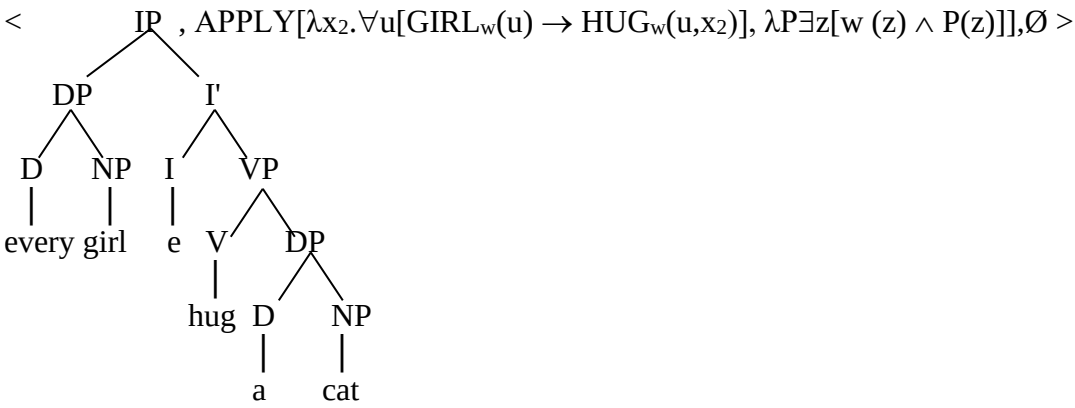
APPLY[$HUG_w(x_2)$,$\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$]
= LIFT[$HUG_w(x_2)$] ($\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$) [def APPLY]
= $\lambda T.T(HUG_w(x_2))$ ($\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$) [def LIFT]
= $\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$ ($HUG_w(x_2)$) [λ-con]
= $\forall u[GIRL_w(u) \rightarrow [HUG_w(x_2)](u)]$ [λ-con]
= $\forall u[GIRL_w(u) \rightarrow HUG_w(u,x_2)]$ [rel notation]

So we get:

< IP , $\forall u[GIRL_w(u) \rightarrow w(u,x_2)]$, {<2,$\lambda P.\exists z[CAT_w(z) \wedge P(z)]$>}>

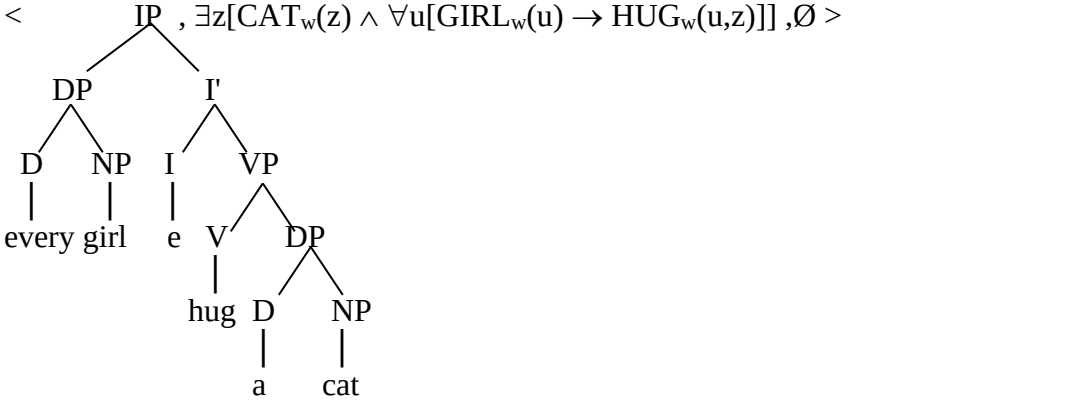


To this we apply RETRIEVE$_2$ and we get:



APPLY[$\lambda x_2.\forall u[GIRL_w(u) \rightarrow HUG_w(u,x_2)]$,$\lambda P\exists z[CAT_w(z) \wedge P(z)]$]]
= LIFT[$\lambda x_2.\forall u[GIRL_w(u) \rightarrow HUG_w(u,x_2)]$]($\lambda P\exists z[CAT_w(z) \wedge P(z)]$)[def APPLY]
= $\lambda T.T(\lambda x_2.\forall u[GIRL_w(u) \rightarrow HUG_w(u,x_2)])(\lambda P\exists z[CAT_w(z) \wedge P(z)])$[def LIFT]
= $\lambda P\exists z[CAT_w(z) \wedge P(z)]$ ($\lambda x_2.\forall u[GIRL_w(u) \rightarrow HUG_w(u,x_2)]$) [λ con]
= $\exists z[CAT_w(z) \wedge [\lambda x_2.\forall u[GIRL_w(u) \rightarrow HUG_w(u,x_2)]](z)]$ [λ-con]
= $\exists z[CAT_w(z) \wedge \forall u[GIRL_w(u) \rightarrow HUG_w(u,z)]]$ [λ-con]

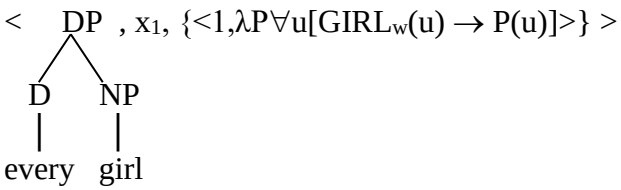
So we get:



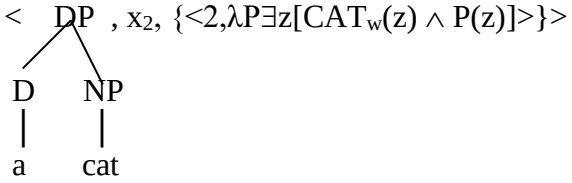
This time the derivation produces the sentence with a different meaning: the object NP takes wide scope over the subject NP.

(1) Every girl hugs a cat.
DERIVATIONS 4 and 5

In both derivations 4 and 5 we apply STORE$_1$ to *every girl* and STORE$_2$ to *a cat*, so these derivations use:

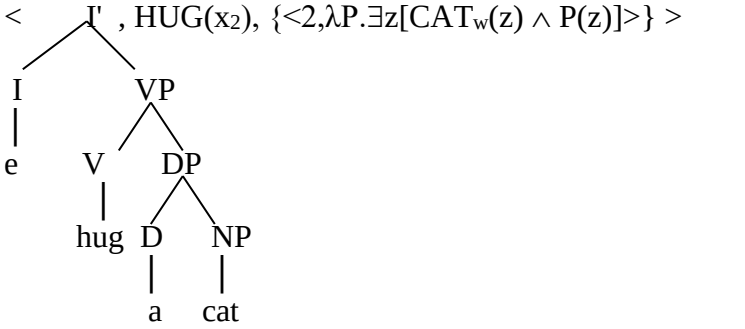


and



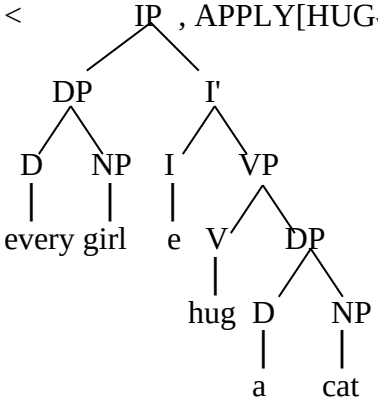
DERIVATION 4

We build up the I' *hug a cat* in the same way as in derivation 3 and get:



This I' takes *every girl* as a specifier, and we get:

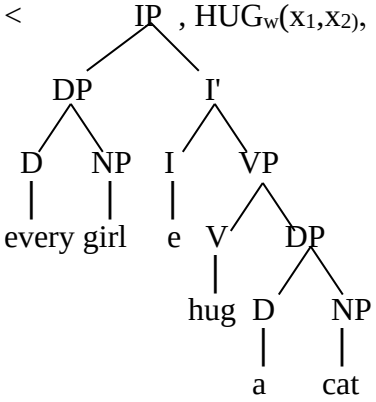
< IP , APPLY[$HUG_w(x_2),x_1$], {<1,$\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$>,
<2,$\lambda P.\exists z[CAT_w(z) \wedge P(z)]$>} >



APPLY[$HUG_w(x_2),x_1$]
= [$HUG_w(x_2)$](x_1) [def APPLY]
= $HUG_w(x_1,x_2)$ [rel notation]

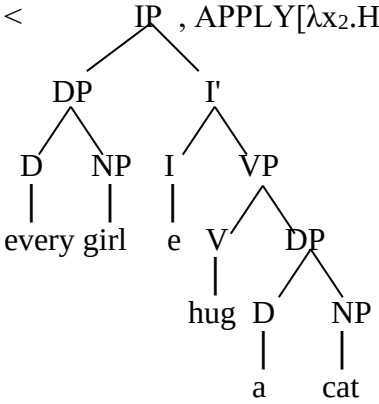
So we get:

< IP , $HUG_w(x_1,x_2)$, {<1,$\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$>, <2,$\lambda P.\exists z[CAT_w(z) \wedge P(z)]$>} >



To this we apply $RETRIEVE_2$
and we get:

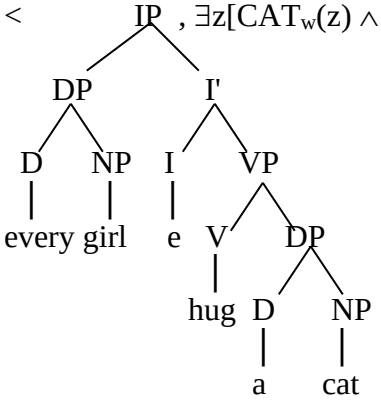
< IP , APPLY[$\lambda x_2.HUG_w(x_1,x_2)$, $\lambda P.\exists z[CAT_w(z) \wedge P(z)]$],
{<1,$\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$>} >



APPLY[$\lambda x_2.HUG_w(x_1,x_2)$, $\lambda P.\exists z[CAT_w(z) \wedge P(z)]$]
= LIFT[$\lambda x_2.HUG_w(x_1,x_2)$]($\lambda P.\exists z[CAT_w(z) \wedge P(z)]$) [def APPLY]
= $\lambda T.T(\lambda x_2.HUG_w(x_1,x_2))(\lambda P.\exists z[CAT_w(z) \wedge P(z)])$ [def LIFT]
= [$\lambda P.\exists z[CAT_w(z) \wedge P(z)]$]($\lambda x_2.HUG_w(x_1,x_2)$) λ con]
= $\exists z[CAT_w(z) \wedge [\lambda x_2.HUG_w(x_1,x_2)](z)]$ [λ-con]
= $\exists z[CAT_{we}(z) \wedge HUG_w(x_1,z)]$ [λ-con]

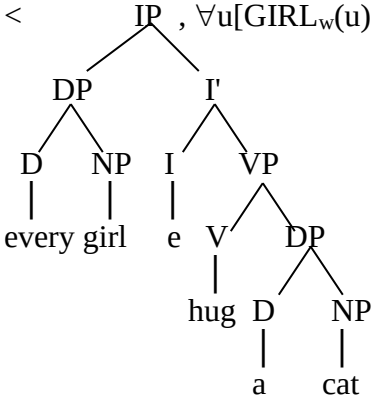
So we get:

< IP , $\exists z[CAT_w(z) \wedge HUG_w(x_1,z)]$, {<1,$\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$>} >



At this moment, we have reached the same stage as we did in derivation 1, $RETRIEVE_1$ will produce, after reduction:

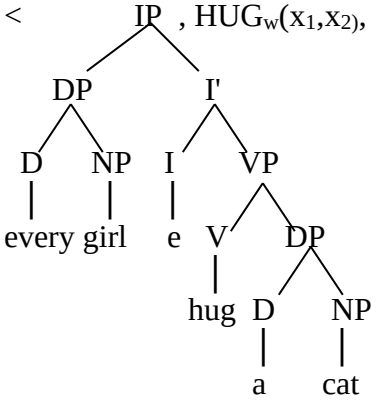
< IP , $\forall u[GIRL_w(u) \rightarrow \exists z[CAT_w(z) \wedge HUG_w(u,z)]]$, Ø>



DERIVATION 5

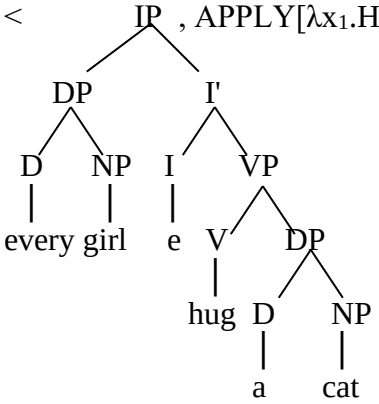
Derivation 5 proceeds in exactly the same way as derivation 4 and produces:

< IP , $HUG_w(x_1,x_2)$, {<1,$\lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$>, <2,$\lambda P.\exists z[CAT_w(z) \wedge P(z)]$>} >



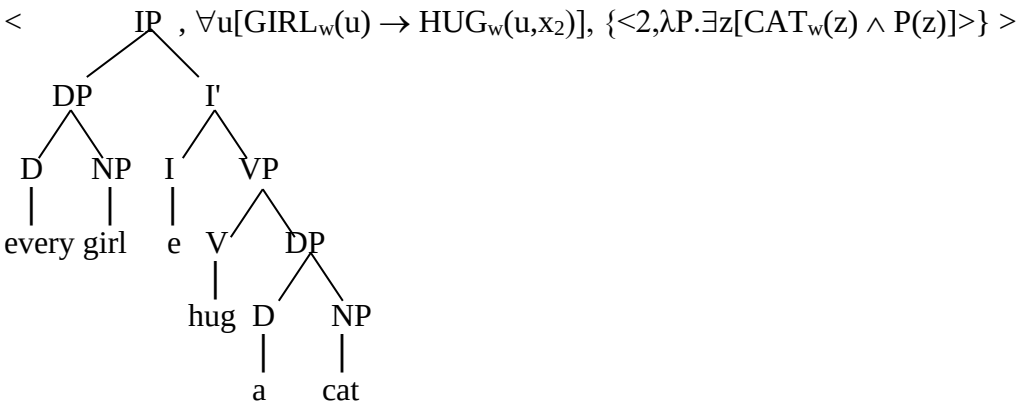
This time, instead of applying $RETRIEVE_2$ we apply $RETRIEVE_1$ and get:

< IP , $APPLY[\lambda x_1.HUG_w(x_1,x_2), \lambda P \forall u[GIRL_w(u) \rightarrow P(u)]$],
{<2,$\lambda P.\exists z[CAT_w(z) \wedge P(z)]$>} >

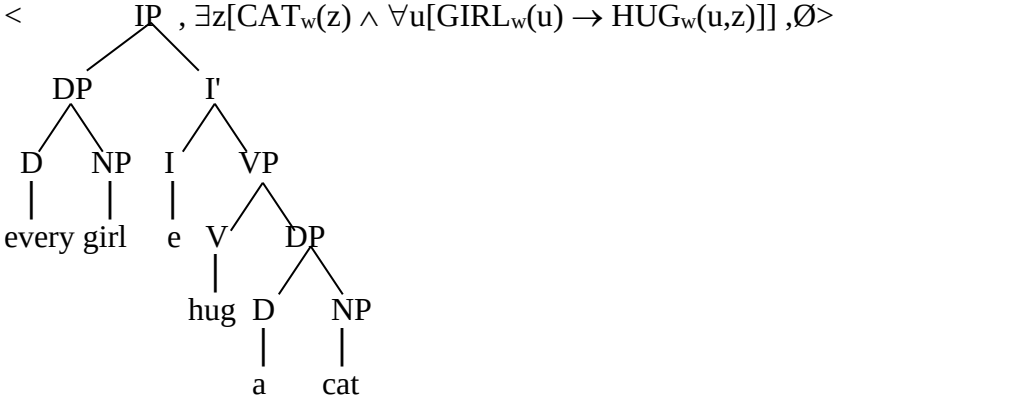


$APPLY[\lambda x_1.HUG_w(x_1,x_2), \lambda P \forall u[GIRL_w(u) \rightarrow P(u)]]$
$= LIFT[\lambda x_1.HUG_w(x_1,x_2)](\lambda P \forall u[GIRL_w(u) \rightarrow P(u)])$[def APPLY]
$= \lambda T.T(\lambda x_1.HUG_w(x_1,x_2))(\lambda P \forall u[GIRL_w(u) \rightarrow P(u)])$[def LIFT]
$= \lambda P \forall u[GIRL_w(u) \rightarrow P(u)] (\lambda x_1.HUG_w(x_1,x_2))$ [λ con]
$= \forall u[GIRL_w(u) \rightarrow [\lambda x_1.HUG_w(x_1,x_2)](u)]$ [λ-con]
$= \forall u[GIRL_w(u) \rightarrow HUG_w(u,x_2)]$ [λ-con]

So we get:



We have now joined derivation 3, applying RETRIEVE$_2$ gives after reduction:

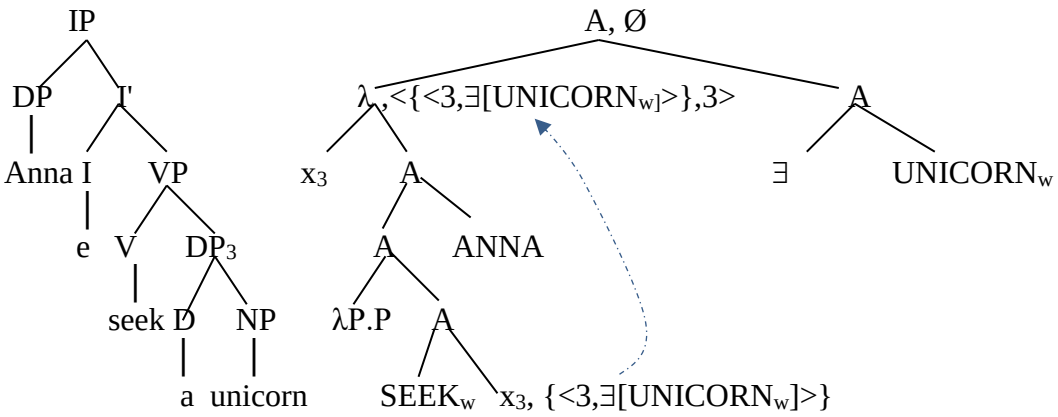


We conclude that we derive *every girl hugs a cat* with two meanings:
for every girl there is a cat that she hugs,
and: there is a cat such that every girl loves him.

(2) Anna seeks a unicorn.
DE RE.

We gave an interpretation strategy that could derive a wide scope reading for *a unicorn* by interpreting *seek* at type <e,<e,t>>. The scope mechanism will derive the same reading:

For ease of representation I will write ∃ for λQλP.∃x[Q(x) ∧ P(x)]
and ∃[UNICORN$_w$] for the application tree with ∃ and UNICORN$_w$ as daughter



APPLY[SEEK$_w$,x$_3$]
= SEEK$_w$(LIFT[x$_3$]) [def. APPLY]
= SEEK(^$_w$λP.P(x$_3$)) [def. LIFT]

APPLY[λP.P,SEEK$_w$(^$_w$λP.P(x$_3$))]
= SEEK$_w$(^$_w$λP.P(x$_3$))

APPLY[SEEK$_w$(^$_w$λP.P(x$_3$)), ANNA]
= SEEK$_w$(^$_w$λP.P(x$_3$)) (ANNA) [def. APPLY]
= SEEK$_w$(ANNA, ^$_w$λP.P(x$_3$)) [rel. notation.]
= [λx.SEEK$_w$(x, ^$_w$λP.P(x$_3$))](ANNA) [backwards λ-con]
[[λyλx.SEEK$_w$(x,⟧(^$_w$λP.P(y))](x$_3$)] (ANNA) [backwards λ-con, x$_3$ is rigid]
= SEEK$_{*,w}$(ANNA,x$_3$) [def SEEK$_*$]

LAMBDA[x$_3$, SEEK$_{*,w}$(ANNA,x$_3$)]
= λx$_3$.SEEK$_{*,w}$(ANNA,x$_3$) [def. LAMBDA]
= λx.SEEK$_{*,w}$(ANNA,x) [alphabetic var.]

APPLY[λx.SEEK$_{*,w}$(ANNA,x), λP.$\exists$z[UNICORN$_w$(z) $\wedge$ P(z)]]

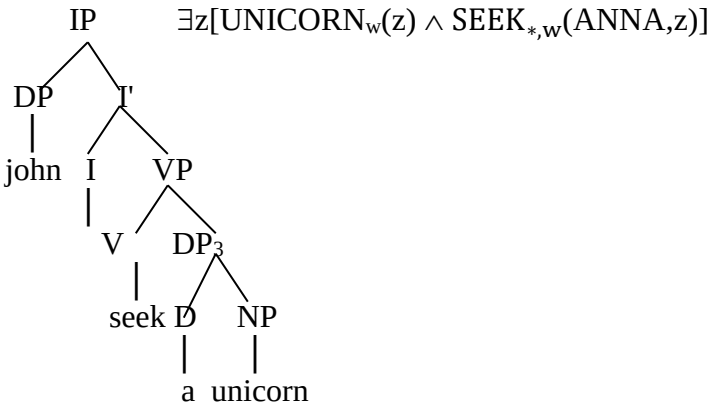
= LIFT[λx.SEEK$_{*,w}$(ANNA,x)](λP.$\exists$z[UNICORN$_w$(z) $\wedge$ P(z)]) [def. APPLY]

= λT.T(λx.SEEK$_{*,w}$(ANNA,x))(λP.$\exists$z[UNICORN$_w$(z) $\wedge$ P(z)]) [def. LIFT]

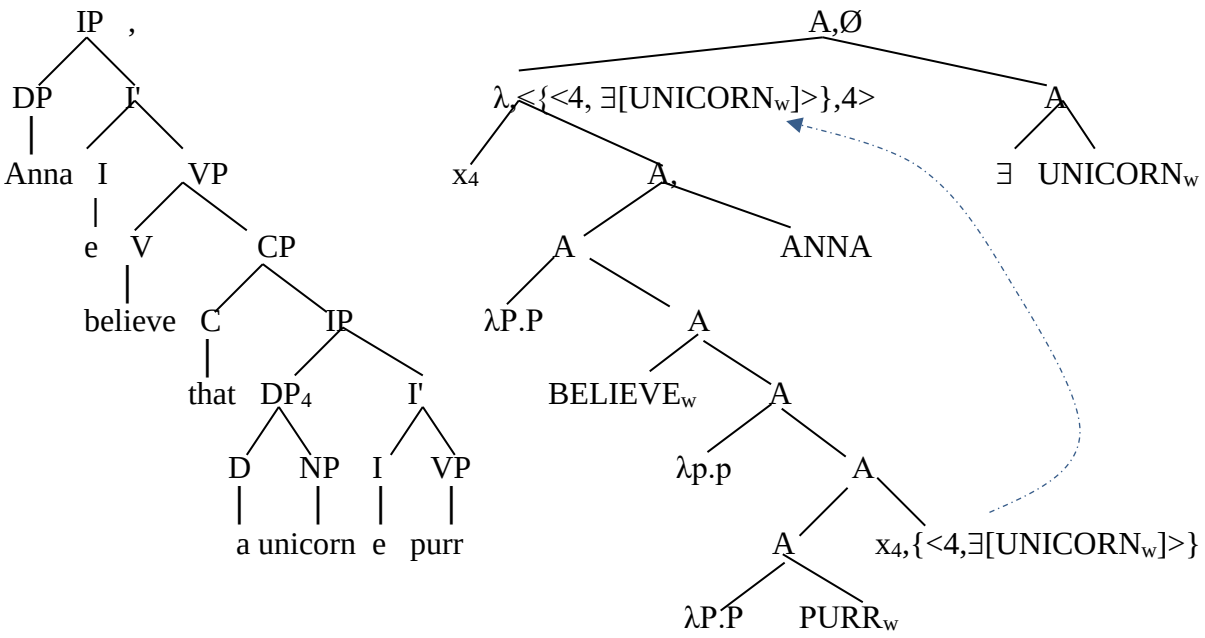
= λP.$\exists$z[UNICORN$_w$(z) $\wedge$ P(z)] (λx.SEEK$_{*,w}$(ANNA,x)) [λ-con]

= $\exists$z[UNICORN$_w$(z) $\wedge$ [λx.SEEK$_{*,w}$(ANNA,x)](z)] [λ-con]

= $\exists$z[UNICORN$_w$(z) $\wedge$ SEEK$_{*,w}$(ANNA,z)] [λ-con, z is rigid]



(3) Anna believes that a unicorn purred.
De RE *a unicorn*



APPLY[λP.P, $PURR_w$],x_4]
= $PURR_w$

APPLY[$PURR_w$, x_4]
= $PURR_w(x_4)$

APPLY[λp.p, $PURR_w(x_4)$] =
$^{\wedge}{}_vPURR_v(x_4)$

APPLY[$BELIEVE_w$, $^{\wedge}{}_vPURR_v(x_4)$]
= BELIEVE($^{\wedge}{}_vPURR_v(x_4)$)

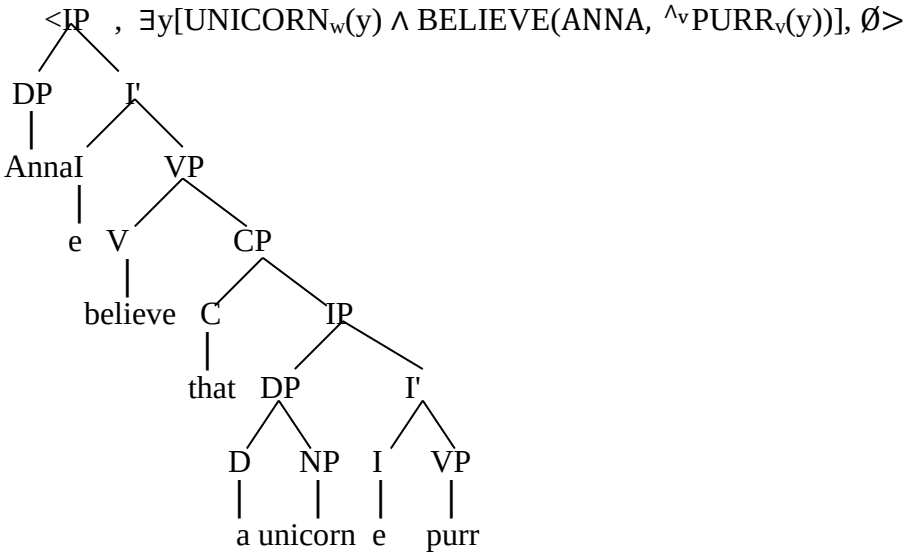
APPLY[λP.P, BELIEVE($^{\wedge}{}_vPURR_v(x_4)$)]
= BELIEVE($^{\wedge}{}_vPURR_v(x_4)$)

APPLY[BELIEVE($^{\wedge}{}_vPURR_v(x_4)$), ANNA]
= BELIEVE(ANNA, $^{\wedge}{}_vPURR_v(x_4)$)

LAMBDA[x_3, BELIEVE(ANNA, $^{\wedge}{}_vPURR_v(x_4)$)]
= λx_4.BELIEVE(ANNA, $^{\wedge}{}_vPURR_v(x_4)$)]
= λz.BELIEVE(ANNA, $^{\wedge}{}_vPURR_v(z)$)]

APPLY[λz.BELIEVE(ANNA, $^{\wedge}{}_vPURR_v(z)$)], λP∃y[$UNICORN_w(y) \wedge P(y)$]]
= ∃y[$UNICORN_w(y) \wedge$ BELIEVE(ANNA, $^{\wedge}{}_vPURR_v(y)$)]

So we get:


<IP , ∃y[UNICORN$_w$(y) ∧ BELIEVE(ANNA, ^$_v$PURR$_v$(y))], Ø>
DP
I'
AnnaI
VP
e
V
CP
believe
C
IP
that
DP
I'
D
NP
I
VP
a unicorn
e
purr

6.2. Some thoughts about wide scope.

I am concerned here with scoping out of the complement of propositional attitude verbs. As is well-known, while indefinite noun phrases in the complements of propositional attitude verbs easily get *de re*-interpretations, such interpretations are generally much harder to get for quantificational noun phrases.

(1) a. Every schoolgirl believes that *a mathematician* wrote *Through the Looking Glass.*
-*de dicto*: this is the kind of stuff mathematicians write.
-*de re*, widest scope: they all believe of Charles Dodgson that he wrote Through the Looking glass under the pseudonym of Lewis Caroll.
-*de re*, narrow scope: Anna believes Hilbert wrote it, Emma believes Hardy wrote it,…
b. Anna believes that every linguist knows many languages.

Several kinds of mechanisms have been proposed to deal the scoping of indefinites (for instance, mechanisms of 'specific interpretation' - not always with a clear understanding of what *specific means*), mechanisms of choice functions (See the L&P volume with papers by Tanya Reinhart and by Yoad winter), skolem functions, individual concepts, etc.

The scope mechanism provided above is unrestricted, allows wide scope over the intensional context for any noun phrase. Other scope mechanisms, like movement, are clause bound and do not extend beyond the IP level. Even if we assume that this is a good idea as a mechanism that applies to scope of true quantifiers, and hence explain why they don't scope as freely as indefinites, we *do* have to face the fact that *de re* interpretations of quantificational elements *are in fact possible*:

(2) a. Anna thinks that every girl in Dana's class is pretty.

Situation: Anna sees a group of girls in Park HaYarkon. She doesn't know them, but she thinks all of them are pretty, and says so (mumbling). You ask me: what did Anna say? Now I know that this is a party of all the girls in Dana's class. You don't know those girls either, but you know Dana, so I say: Anna thinks that every girls in Dana's class is pretty. This is felicitous and it shows a *de re* reading: it's not literally what she said, and there is no reason to assume that she thinks *de dicto* that the girls in Dana's class are pretty.

What can we do about this?

Here is a first suggestion:

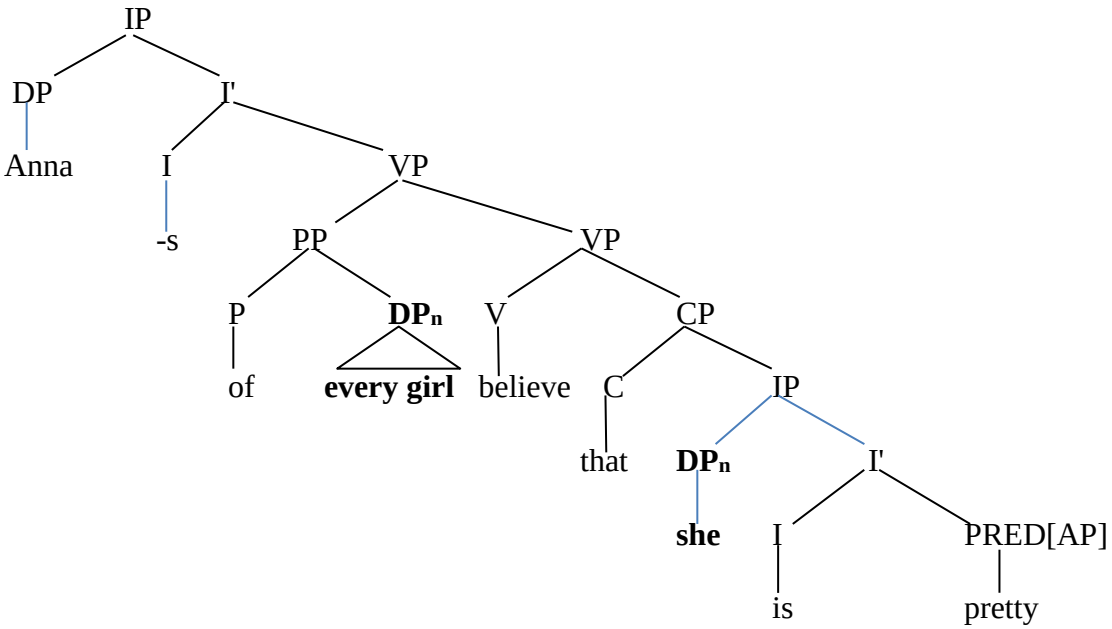
> **Retrieval assumption (Default):**
> Retrieval of noun phrases in store takes place **as soon as** the types fit for retrieval.

With this principle, the store is by default emptied when the derivation reaches type t, i.e. at the IP level. With this principle, you will not expect wide scope readings.
How do we get them when we do?
Well, let us first look at the following *de re* construction:

(2) b. Anna believes *of every girl in Dana's class* that *she* is pretty.

Here there is an adjunct on the higher verb which takes the quantificational noun phrase as a complement, and the CP-complement of the propositional attitude verb contains a variable (a pronoun). Note too that the variable is obligatory, (2c) is infelicitous:

(2) c. Anna believes *of every girl in Dana's class* that *Emma* is pretty.



This structure only allows for a *de re* reading. How does it come about?

> **Binding in the *de re* structure:**
> 1. We interpret the pronoun in the CP-complement as a variable x_n
> 2. We require that this variable is **bound by the complement of the higher adjunct.**

Quantifiers don't bind variables, so this is interpreted in the following way:

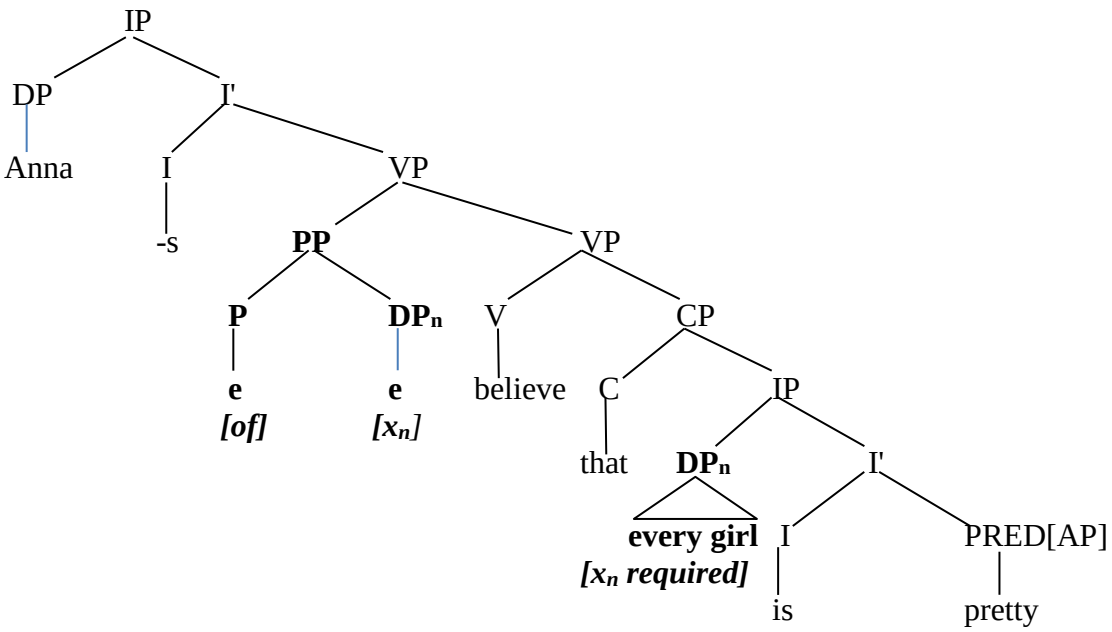
> 3. We **store** the DP-complement in the higher adjunct under **the same variable** x_n as the variable in the CP-complement: x_n {<n, *every girl in Dana's class*>}

This is all.

Now when we retrieve the stored meaning at the matrix level, abstraction will bing both variables. Obviously, this gives the correct *de re* reading.

Suggestion: What if we assume that (2a) allows a structure along the lines of the *de re structure*?

(2) a. Anna thinks that every girl in Dana's class is pretty.



For concreteness:

1. Assume a null P with the same meaning as *of*, and assume that it triggers the conditions of the *de re* structure.
2. Assume that the null complement of P is a variable x_n.
3. The *de re* structure requires a variable in the CP complement. This condition is satisfied by **storing** the DP-subject inside the CP under **the same** variable x_n {<n,*every girl in Dana's class*>}.

We have now created a situation in which there is **grammatical pressure on the retrieve operation**:

-By default, *every girl* comes out of store at the lower IP level.
-There is pressure to keep *every girl* in store to resolve the higher variable.
We assume that this pressure can override the default.

[Note that if this is represented in the syntax, we are creating a weak cross-over violation. But then, weak cross-over violations are so common anyway that there is little reason to assume a principle forbidding weak cross-over in the syntax.

cf:
(3) The string that ties *it* to *its* mother keeps *every foetus* alive.]

We can go one step further, and observe that for corresponding propositional attitude nouns the *de re* structure cannot be formed with the preposition *of* but requires *about*:

(4) a. Anna is bothered by the belief *of every girl in Dana's class* that *she* is pretty.
b. Anna is bothered by the belief *about every girl in Dana's class* that *she* is pretty.
c. Anna is bothered by the belief that *every girl in Dana's class* is pretty.

In (4a) *every girl in Dana's class* expresses the believer. The *de re structure* here can only be expressed as (4b).

Let us now make the assumption that the null-preposition is a variant of a lexically realized preposition, and in fact, a null-variant of a *flexible* preposition, one that easily switches between multiple interpretations, like *of*:

Null *of*-assumption: the null-PP is a null-version of *of*.

Since *of* allows the *about*-interpretation in the verbal domain, its null-version is available for *de re* interpretations of propositional attitude verbs; since *of* does not allow the *about*-interpretation in the nominal domain, its null-version is not available for *de re* interpretations of propositional attitude nouns. It follows that cases like (4c) do not allow the same wide scope interpretations as do the corresponding propositional attitude verb cases.
Note finally, that the escape strategy suggested here for propositional attitude verbs is unavailble for relative clauses:

(5) I want you to make me a list of all cases of a boy that invited *every girl in the class* to his birthday party.

(5) invites you to write down cases of all-girl inviting boys, not a list of girl-boy pairs, specifying per girl the boys that invited her. *every girl in the class* does not scope out of the relative clause. The suggestions made here provide no reason to expect otherwise.

6.3. Relative clauses and scope

6.3.1. Relative clauses

(1) The cat that - purred
(2) The woman that Ronya likes –

gaps: relativization gaps are semantically interpreted as variables. The variable is stored and so that its type can be retrieved later..

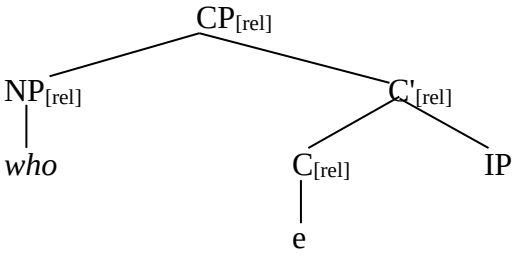
$\langle DP_{[rel]}, x_n, \{\langle n,x_n\rangle\}\rangle$ where $x_n \in VAR_e$

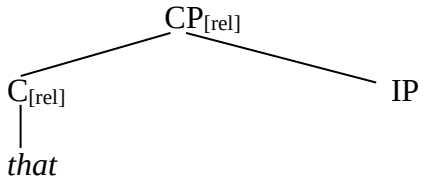
e

We start here with a filled store.

Possible respons to islands: if the derivation goes through an island and $\langle n,x_n\rangle \in S$, where $\alpha \neq x_n$ replace x_n by x. This will leave something in store that cannot be retrieved later, so the store does not end up empty, and the derivation blocks.

Relative clause structure:





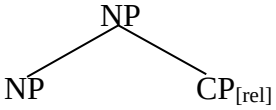
$\langle C_{[rel]}, \lambda P.P, \emptyset\rangle$ with $P \in VAR_{\langle e,t\rangle}$

$\langle$*who*, $PERSON_w, \emptyset\rangle$ with $PERSON \in CON_{\langle s,\langle e,t\rangle\rangle}$

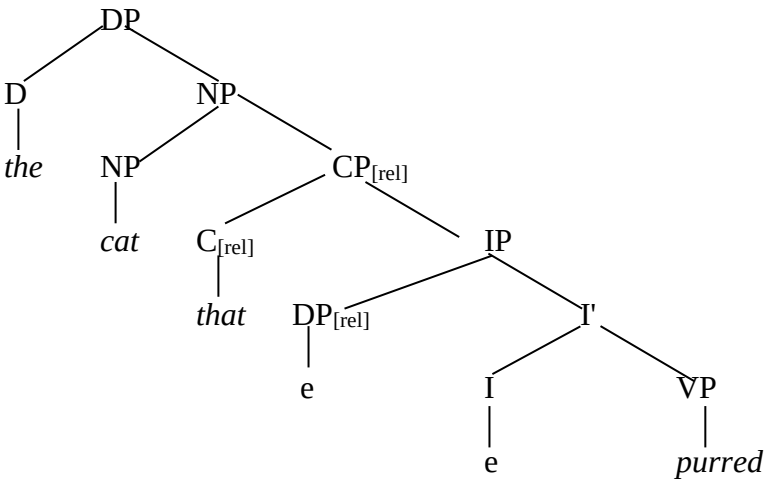
$BIND_n$:
If <IP, IP', S> is generated and $\langle n,x_n\rangle$ is the only element stored under index n in S, then:
$BIND_n[\langle IP, IP', S\rangle] = \langle IP, \lambda x_n.IP', S - \{\langle n,x_n\rangle\}\rangle$

Assumption: $BIND_n$ can resolve type mismatch.

Modifier structure:



(1) the cat that purred



We derive:

$\langle DP_{[rel]}, x_n, \{\langle n,x_n\rangle\}\rangle + \langle I', PURR_w, \emptyset\rangle \rightarrow \langle IP, PURR_w(x_n), \{\langle n,x_n\rangle\}\rangle$
t

$\langle C_{[rel]}, \lambda P.P, \emptyset\rangle + \langle IP, PURR_w(x_n), \{\langle n,x_n\rangle\}\rangle \rightarrow ?$

The structure $[_{CP[rel]}\ C[_{rel]}\ IP]$ is a complement structure, this means it gets interpreted via:

APPLY[$\lambda P.P$, $PURR_w(x_n)$]
<<e,t>,<e,t>> t

There is a type mismatch which is resolved by applying $BIND_n$ at the IP level:

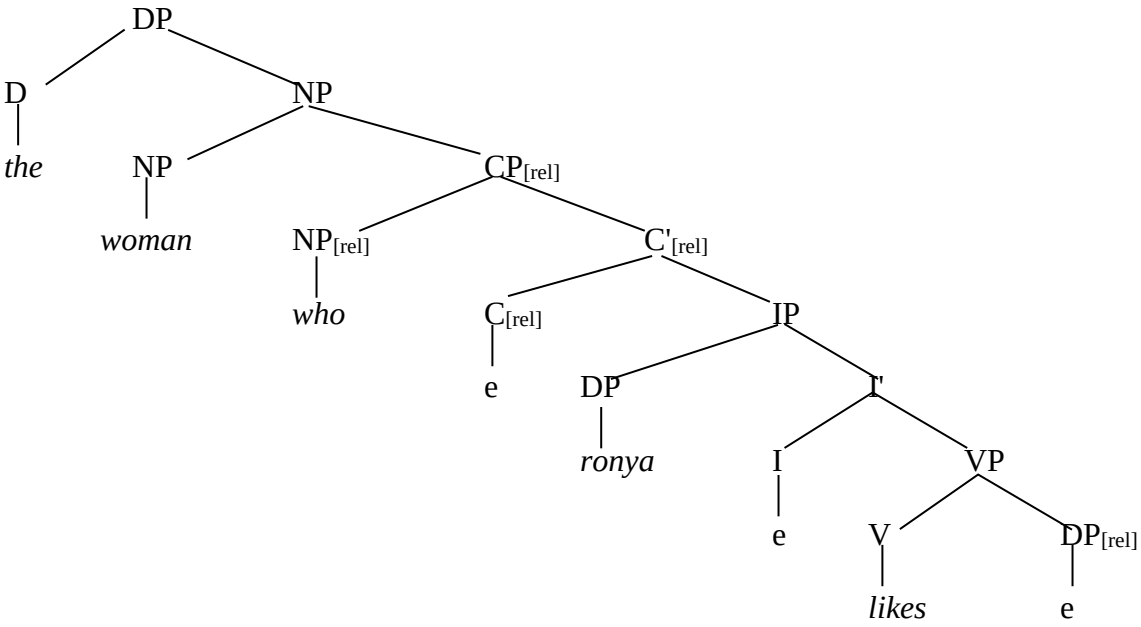
$BIND_n[\langle IP, PURR_w(x_n), \{\langle n,x_n\rangle\}\rangle] = \langle IP, \lambda x_n.PURR_w(x_n), \emptyset\rangle$

$\langle C_{[rel]}, \lambda P.P, \emptyset\rangle + \langle IP, \lambda x_n.PURR_w(x_n), \emptyset\rangle \rightarrow \langle CP_{[rel]}, \lambda x_n.PURR_w(x_n), \emptyset\rangle$

At the next stage, the relative clause interpretation shifts with the adjunction shift to a modifier interpretation, applies to the head noun, and the definite article applies, and we derive:

$\langle DP, \sigma(\lambda x.CAT_w(x) \wedge PURR_w(x), \emptyset\rangle$

(1) the woman who Ronya likes -



We derive <IP, $LIKE_w(RONYA,x_n)$, {<n,x_n>}> and with $BIND_n$:

<$C'_{[rel]}$ $\lambda x_n.LIKE_w(RONYA,x_n)$, Ø>

The $CP_{[rel]}$ is a specifier structure, so here too adjunction lift applies to the relative, and we derive:

<$CP_{[rel]}$, $\lambda x.PERSON_w(x) \wedge LIKE_w(RONYA,x)$, Ø>

This puts a selection restriction on the possible head, which *woman* satisfies. We derive in the same way as above:

<DP, $\sigma(\lambda x.WOMAN_w(x) \wedge LIKE_w(RONYA,x))$, Ø>

6.3.2 Scope Islands and functional readings

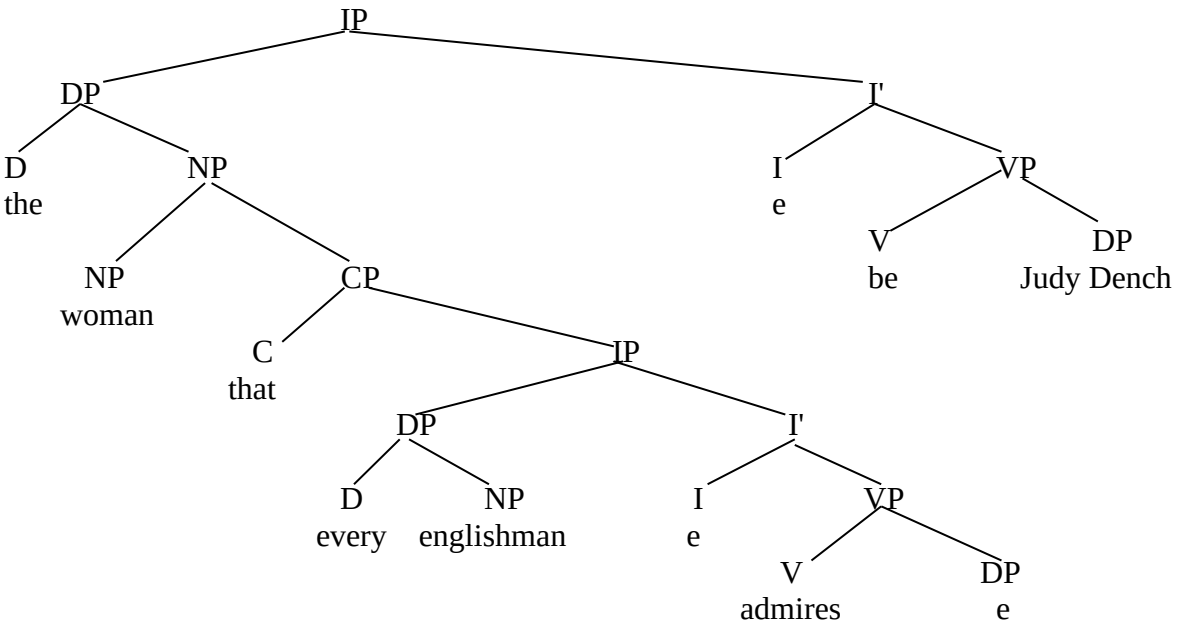
Since Rodman 1972 (1975): Relative clauses are scope islands:

(3) a. Some girl dated every rockstar
b. Some girl who dated every rockstar was Japanese.

Observation: (3a) allows with the right intonation an inverse scope reading.
(3b) does not: i.e. (3b) does not allow a reading where what is expressed that
Yoko dated John
Yuko dated Mick
Ruriko dated Robert ….

So (3) only has a reading with *every englishman* interpreted *in situ*.

(3) The woman that every englishman admires is Judy Dench.

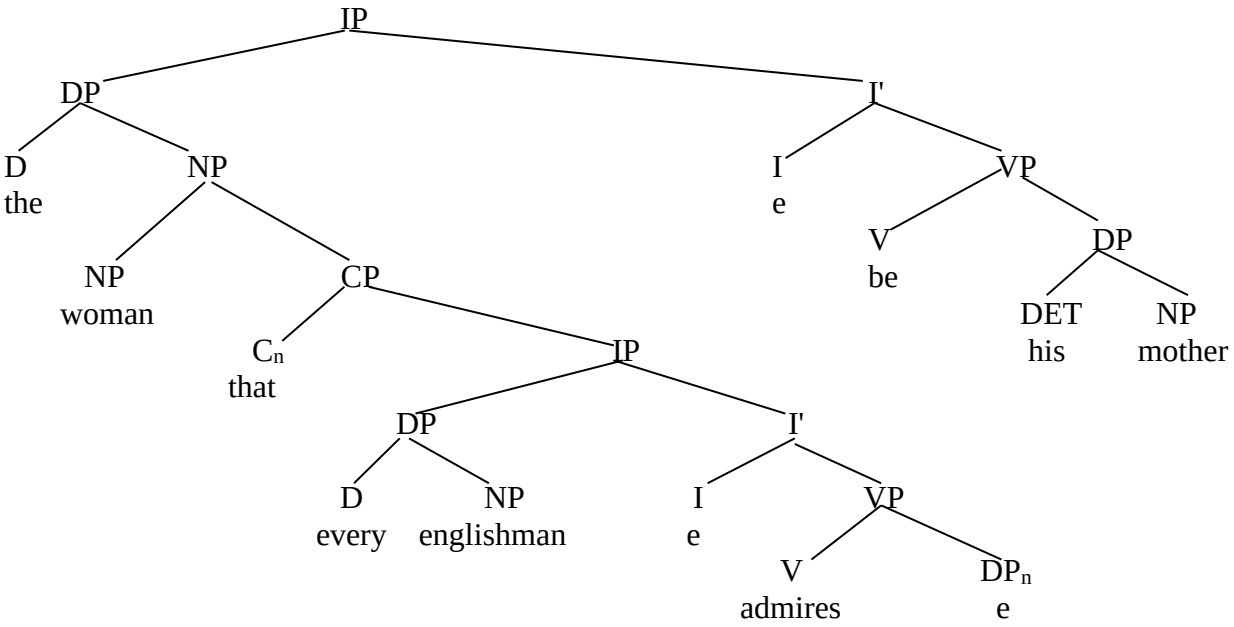


We derive, just like in the cases above:

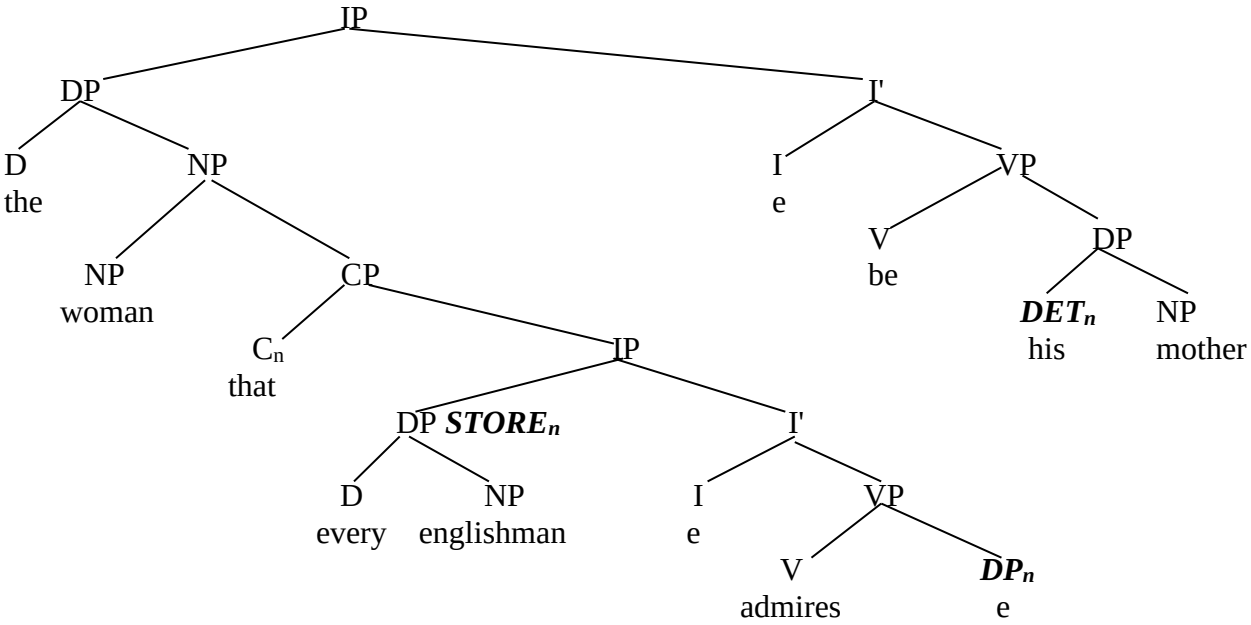
$\sigma(\lambda x.\mathrm{WOMAN}_w(x) \wedge \forall y[\mathrm{ENGLISHMAN}_w(y) \rightarrow \mathrm{ADMIRE}_w(x,y)]) = \mathrm{JUDY\ DENCH}$

We come to (4):

(4) The woman that every englishman admires is his mother.



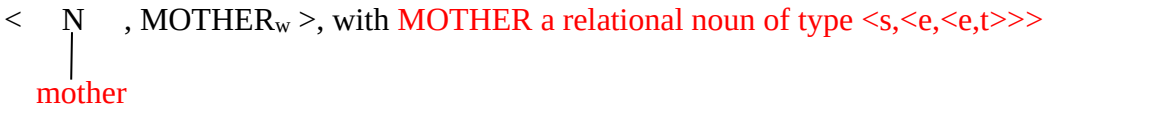
Suggestion:



Problem: No syntactic or semantic theory allows retrieval of this DP stored in the relative clause at the level where it takes scope over *his mother*.
-Rodman 1972 observed that relative clauses are scope islands.
-QR can not extract out of DPs.
-Yael Sharvit: applying QR anyway predicts wrong interpretations.

Alternative: functional readings: Groenendijk and Stokhof 198?, Jacobson 198?, Sharvit 199?
Main idea:
1. DP_n can be interpreted as a functional variable of type <e,e> (functions of this type are called Skolem functions in logic).
2. *his* in *his mother* is **not** semantically bound by the quantifier (or semantically co-indexed with the gap).

Let's start with 2.
We assume that *mother* is a relational noun, what is a noun of type <e,<e,t>>.

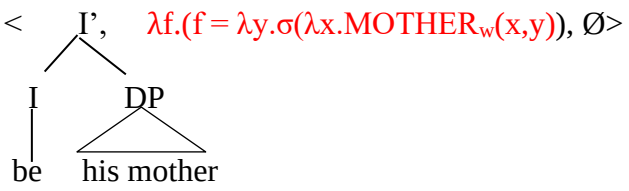


We assume that in *his mother*, *his* is not a free pronoun, but is semantically bound to the mother function: *his/her mother* denotes the function that maps individuals onto their mother:

$<[_{DP}$ his mother], $\lambda y.\sigma(\lambda x.MOTHER_w(x,y)>$

$\lambda y.\sigma(\lambda x.MOTHER_w(x,y)$ is the function in $D_{<e,e>}$ that maps y onto the mother of y.

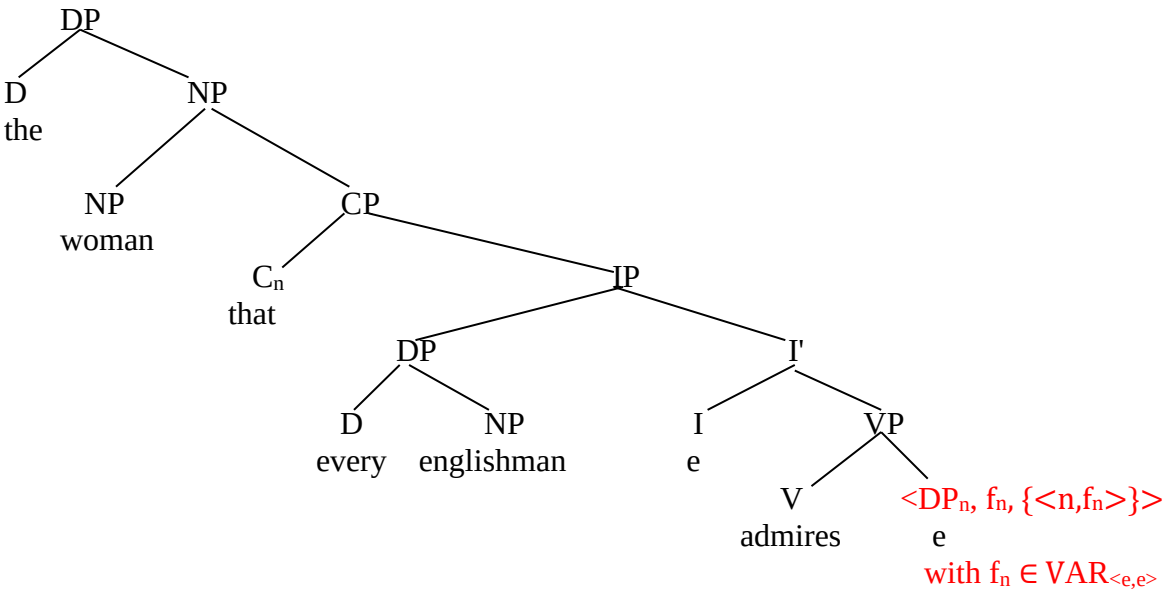
With this we assume that the predicate *be his mother* denotes the obvious predicate of type <<e,e>,t>, with $f \in VAR_{<e,t>}$



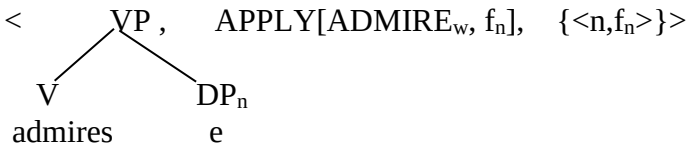
At this point, what we need to do is:
derive an interpretation f of *the woman that every englishman admires* of type <e,e>.
The sentence will then denote the claim that
$f = \lambda y.\sigma(\lambda x.MOTHER_w(x,y)$.

the woman that every englishman admires

We make really only one assumption, and derive most of the rest of the analysis from general reasoning. The assumption is that the relativixation gap can be a functional variable, a variable of type <e,e>.



At the VP level we derive:



where $ADMIRE \in CON_{<s,<e,<e,t>>>}$ and $f_n \in VAR_{<e,e>}$
So, we have a type mismatch:
$ADMIRE_w$ gobbles up individuals, and it is fed a function.
This type mismatch needs to be resolved with a type shifting rule.
Which rule?

Let's reason. We just assumed that *his/her mother* can denote the mother function. What does it mean that *Hilary admires his/her mother* in terms of *Hilary* and the mother function? It means, of course, that Hilary admires his/her value of the mother function. Thus it is natural to interpret the non-welformed: $ADMIRE_w$(HILARY, M) as the wellformed: $ADMIRE_w$(HILARY, M(HILARY)).
Abstracting away Hilary, we get: $\lambda x.ADMIRE_w(x,M(x))$.
Abstracting away M, we get: $\lambda f \lambda x.ADMIRE_w(x,f(x))$
Abstracting away $ADMIRE_w$, we get: $\lambda R \lambda f \lambda x.R(x,f(x))$

This operation, called BIND, was proposed as a type shifting operation by Polly Jacobsen (but it goes back to the Combinatory Logic of Curry).

BIND: $\langle e,\langle e,t\rangle\rangle \rightarrow \langle\langle e,e\rangle,\langle e,t\rangle\rangle$
BIND[α] = λf λx.α(x, f(x))

With this, we resolve the type mismatch:

APPLY[ADMIRE$_w$, f_n] = BIND[ADMIRE$_w$](f_n) =
λfλx.ADMIRE$_w$(x,f(x))(f_n) =
λx.ADMIRE$_w$(x,f_n(x)) of type <e,t>
the property that you have if you admire your f_n-value.

From here we build up the IP:

<[$_{IP}$ *every Englishman admires* e_n],
∀x[ENGLISHMAN$_w$(x) → ADMIRE$_w$(x,f_n(x))] , {<n,f_n>}>

A the next level variable f_n comes out of store and is abstracted over, and we get:

<[$_{C'}$ *that*$_{[rel]}$ *every Englishman admires* e_n],
λf_n. ∀x[ENGLISHMAN$_w$(x) → ADMIRE$_w$(x,f_n(x))] , Ø>
The set of all functions that map every Englishman onto his f_n value, of type <<e,e>,t>

Next, we combine this with the head noun woman, *woman* with interpretation:
WOMAN ∈ CON$_{\langle s,\langle e,t\rangle\rangle}$

Again, we have a type mismatch, because for intersective adjunction, we need *woman* to have an interpretation at type <<e,e>,t>.
So we need to lift WOMAN$_w$ from type <e,t> to type <<e,e>,t>.

Here we are guided by the intution that
while *every Englishman* constrains the *domain* of the functions in question,
woman constrains the *range* of the functions: they are, for Englishmen, woman-valued functions.
We can let the type shifting rule express exactly that, the function RANGE:

RANGE: <e,t> → <<e,e>,t>
RANGE = λPλf.∀x[P(f(x))]

RANGE[WOMAN$_w$] = λf.∀x[WOMAN$_w$(f(x)] of type <<e,e>,t>
The set of functions that are woman-valued.

Now we do the adjunction, at type type <<e,e>,t> and derive:

<$[_{CP}$ *woman that every englishman admires* e_n],
$\lambda f.\forall x[WOMAN_w(f(x)) \wedge \forall x[ENGLISHMAN_w(x) \rightarrow ADMIRE_w(x,f(x))],\ \emptyset>$
The set of functions that are woman-valued such that every Englishman admires his value.

At this point we need to move from the technical domain to a normal context.
At the next level the definite article applies.
But it requires a singleton set.
But the above set is not going to be a singleton set, because you can arbitrarily build such functions by linking Englishmen with women.

For instance, suppose every englishman admires two women, his mother and either his grandmother or his primary schoolteacher.
Then the mother function f_1 is in the above set.
Now take the function f_2 that maps every englishman except Dan onto his mother, and it maps Dan onto his grandmother, who he admires. Then f_2 is also in this set.
Take the function f_3 that only differs from f_2 in that it maps Joe onto his schoolteacher, who he admires: then f_3 is also in that set.
It should be clear that with permutations the above set of functions, under the pretty normal conditions given is massively big.
Which means that the σ operation would never be defined, even if the set of Englishmen and Woman are contextually restricted, and we allow partial functions, so that we can restrict ourselves to functions whose domain is Englishmen.

The standard way out, which I will follow here (though a different analysis is given in my handlout: *Two tier semantics for relative clauses* from 2007, which is on my webpage) is to assume that the context restricts the set of functions derived to:
a set of *contextually salient natural functions*: **C**.

<$[_{NP}$ *woman that every englishman admires* e_n],
$\lambda f.\mathbf{C}(f) \wedge \forall x[WOMAN_w(f(x)) \wedge \forall x[ENGLISHMAN_w(x) \rightarrow ADMIRE_w(x,f(x))],\ \emptyset>$
The set of contextually relevant natural woman-valued functions that map every Englishman onto a woman that he admires.

Thus **C** would typically include such natural functions as *his mother*, *his grandmother*, *his primary school teacher*, *his first girlfriend*,... etc. standard contextually salient functional roles.
But not crazy ones, like the function that maps me on my mother, you on your sister, Buck on his girlfriend, Chuck on Madonna, etc...
And not the function that maps some englishmen onto their mother, others onto their grandmother and yet others onto their schoolteacher,
because that function would not be a natural function.

In the example, **C** would contain the mother function, the grandmother function and the schoolteacher function, and only the first one, the mother function is such that every englishman admires his value, since not every englishman admires his grandmother and not every englishman admires his schoolteacher.

Given this, we can plausibly assume that in context the definite DP is well defined:

$\langle[_{DP}$ *the woman that every englishman admires* $e_n]$,
$\sigma(\lambda f.\mathbf{C}(f) \wedge \forall x[WOMAN_w(f(x)) \wedge \forall x[ENGLISHMAN_w(x) \rightarrow ADMIRE_w(x,f(x))]),\ \emptyset\rangle$

The unique contextually salient woman-valued function that satisfies the requirement that every Englishman admires his value for this function.

The statement then express that this function is the mother function:

$\langle[_{IP}$ *the woman that every englishman admires* e_n *is his mother*$]$,
$\sigma(\lambda f.\mathbf{C}(f) \wedge \forall x[WOMAN_w(f(x)) \wedge \forall x[ENGLISHMAN_w(x) \rightarrow ADMIRE_w(x,f(x))])$
$=$
$\lambda y.\sigma(\lambda x.MOTHER_w(x,y),\quad \emptyset\rangle$

The unique contextually salient woman-valued function that satisfies the requirement that every Englishman admires his value for this function, is the mother function.

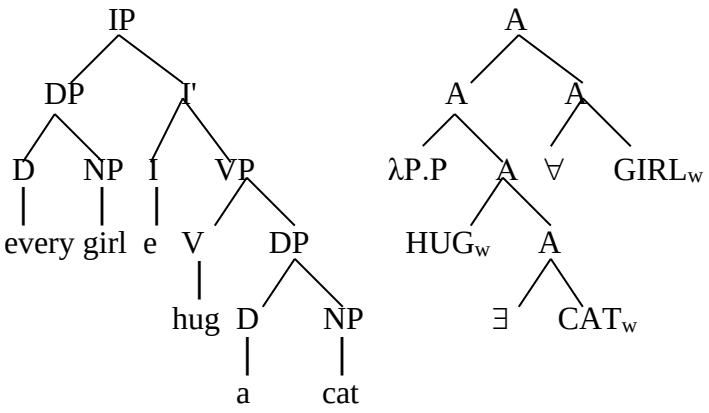
In sum: we see that we have solved the problem of 'binding' the pronoun without relying on the scope mechanism:
-the pronoun *his* is bound locally in *his mother*: $\lambda x.\sigma(\lambda y.MOTHER_w(x,y))$
-the value of the gap function is bound by *every englishman* via the typeshifting operation BIND, this means that it is bound locally in *admire*: $\lambda f\lambda x.ADMIRE_w(x,f(x))$
-We derive a definite that denotes a natural function. The function identity then identifies the functions and gets the reading right.

6.4. Function composition

6.4.1. Some remarks on dislocation in the syntax and in the semantics.

The X-bar theory I have introduced provides us with a theory of syntactic trees. For the present discussion it is useful to give the semantic interpretations associated with these syntactic trees the form of a tree as well. This is done along the following lines:

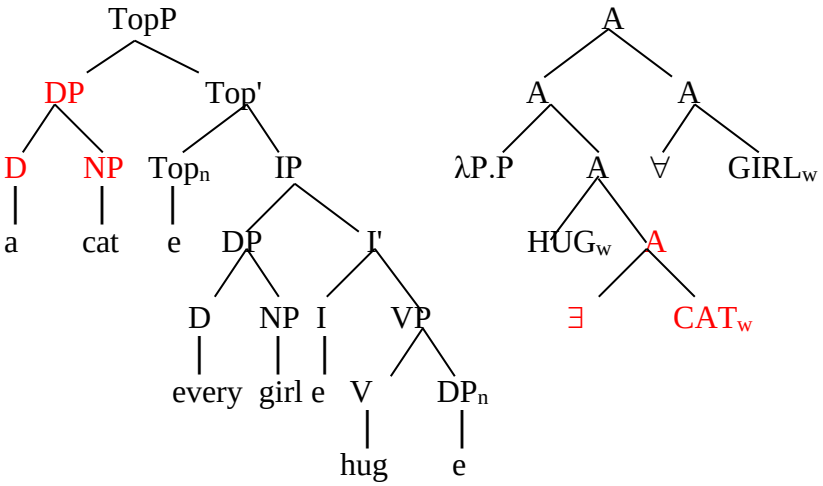
Let: $\forall := \lambda Q \lambda P.\forall x[Q(x) \rightarrow P(x)]$
$\exists := \lambda Q \lambda P.\exists x[Q(x) \wedge P(x)]$
A := APPLY
$[_A\ \alpha, \beta\ ]$:= APPLY[α,β]



Thus, the semantic tree consists of the meanings of the basic expressions at the leaves, and the semantic operations labeling the non-leaves, under the convention that the function is on the left branch.

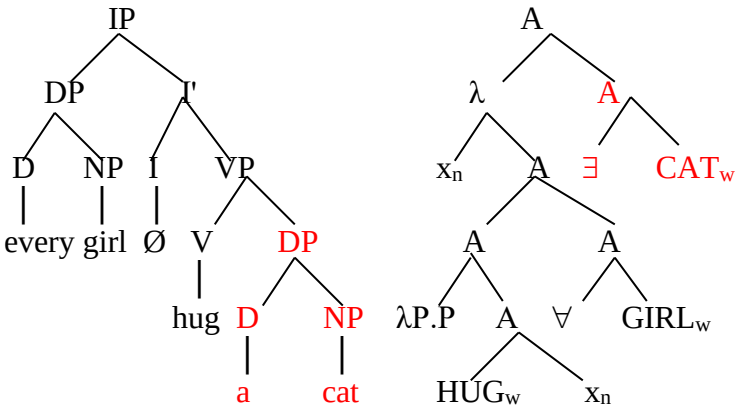
Now, we are interested in two kinds of **dislocation**: **syntactic dislocation** and **semantic dislocation**:

Syntactic dislocation: e.g. topicalization:
the topic is sitting at the top of the syntactic tree while its interpretation is (or can be) sitting in its normal place down in the semantic tree:



Semantic dislocation: e.g. inverse scope:
the meaning of *a cat* is sitting at the top of the semantic tree while the DP *a cat* is sitting in its normal place down in the syntactic tree:

Let: $[_{\lambda}\ x_n, \beta] := \lambda x_n.\beta$



When you calculate the meaning, you'll get out:

$\text{APPLY}[\lambda x_n.\forall x[\text{GIRL}_w(x) \rightarrow \text{HUG}_w(x,x_n)], \lambda P.\exists y[\text{CAT}_w(y) \wedge P(y)]]$
$= \exists y[\text{CAT}_w(y) \wedge \forall x[\text{GIRL}_w(x) \rightarrow \text{HUG}_w(x,y)]]$

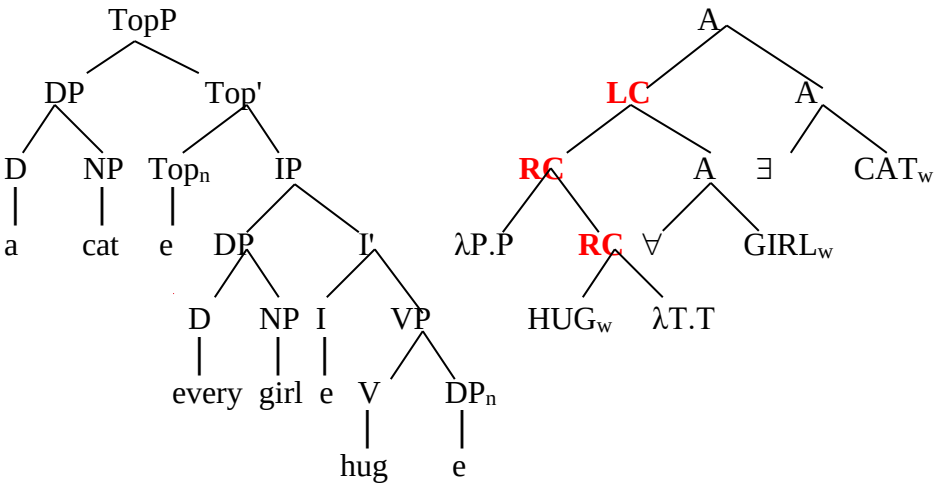
For each of these two dislocation situations there are three strategies to deal with them:

1. **Syntactic dislocation**: a syntactic expression is sitting at the top of the syntactic tree, while its meaning is sitting at the bottom of the semantic tree:

STRATEGY 1.1:
Generate both [$_{DP}$ a cat] and its meaning ∃[CAT_w] at the bottom of the respective trees, and **move** [$_{DP}$ a cat] up to the top of the syntactic tree.
This is **syntactic movement**.

STRATEGY 1.2:
Generate both [$_{DP}$ a cat] and its meaning ∃[CAT_w] at the top of the respective trees, and **move** the meaning ∃[CAT_w] down to the bottom of the semantic tree.
This is what is called **reconstruction**.

STRATEGY 1.3:
Generate both [$_{DP}$ a cat] and its meaning ∃[CAT_w] at the top of the respective trees, but assume a different semantic tree, which will have the effect of movement, without movement:
This is a **non-movement function composition analysis**: (e.g. GPSG, Categorial Grammar, Jacobson)



Here: [$_{RC}$ α, β] := RC[α,β] and RC is **right function composition**:

Right Function Composition:
RC[α,β] = λx.APPLY[α, β(x)]

[$_{LC}$ α, β] := LC[α,β] and LC is **left function composition**:

Left Function Composition:
LC[α,β] = λx.APPLY[α(x), β]

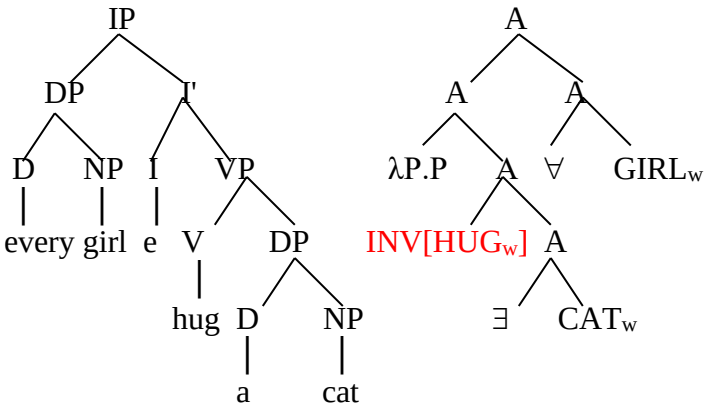
We discuss this below.

2. **Semantic dislocation**: a meaning is sitting at the top of the semantic tree, while the syntactic expression of which it is the meaning is sitting at the bottom of the syntactic tree:

STRATEGY 2.1:
Generate both [$_{DP}$ a cat] and its meaning $\exists[CAT_w]$ at the bottom of the respective trees, and **move** $\exists[CAT_w]$ up to the top of the semantic tree.
This is **semantic movement**: Cooper's storage mechanism, and also Quantifier Raising.

STRATEGY 2.2:
Generate both [$_{DP}$ a cat] and its meaning $\exists[CAT_w]$ at the top of the respective trees, and **move** the [$_{DP}$ a cat] down to the bottom of the syntactic tree. An example of this strategy is (in essence) Montague's original quantifying-in rule.

STRATEGY 2.3:
Generate both [$_{DP}$ a cat] and its meaning $\exists[CAT_w]$ at the bottom of the respective trees, but assume a different semantic tree, which will have the effect of movement, without movement:
This is a **non-movement type shifting analysis** (Hendriks in Categorial Grammar):



Here: INV[α] = λTλU.T(λy.U(λx.α(x,y)))
This means that you shift the verb meaning to the function which will put the arguments it receives in **inverse** scope order:

$INV[HUG_w] = \lambda T \lambda U.T(\lambda y.U(\lambda x.HUG_w(x,y)))$

$APPLY[INV[HUG_w], \lambda P.\exists y[CAT_w(y) \wedge P(y)]]$
$= \lambda U.\exists y[CAT_w(y) \wedge U(\lambda x.HUG_w(x,y))]$

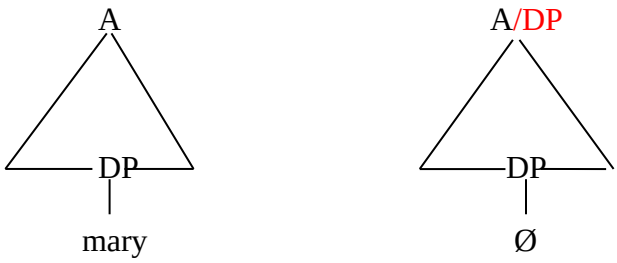
we take λP.P here to stands for the identity function at the required type (which is a higher type in this case). We get:

APPLY[λU.$\exists$y[CAT_w(y) $\wedge$ U(λx.HUG_w(x,y))], λP.$\forall$x[CAT_w(x) $\rightarrow$ P(x)]]
= $\exists$y[CAT_w(y) $\wedge$ $\forall$x[$GIRL_w$(x) $\wedge$ HUG_w(x,y)]

6.4.2. Categorial grammar and function composition

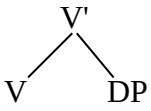
We add to the syntactic theory developed sofar a treatment of gaps in categorial grammar. Our main interest here is to introduce function composition and small and big lambdas. (Our discussion here is indepted to many works, but I single out the work on surface compositionality by Polly Jacobson as the main inspiration for the current discussion.)

Let A stand for a tree with topnode A.
A/DP stands for a tree with topnode A, and with node DP missing in it, i.e. with a DP-gap:

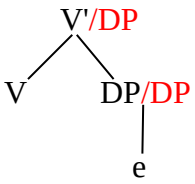


On this model, a tree of the form DP/DP is a DP with a DP missing in it. The smallest kind of this is a DP with *itself* missing, this is a DP-gap:

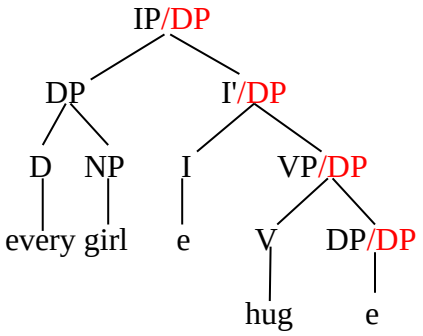
The normal X-bar rules that we have dictate the natural principles for percolating slashes up the tree: X-bar theory allows a tree of the form:



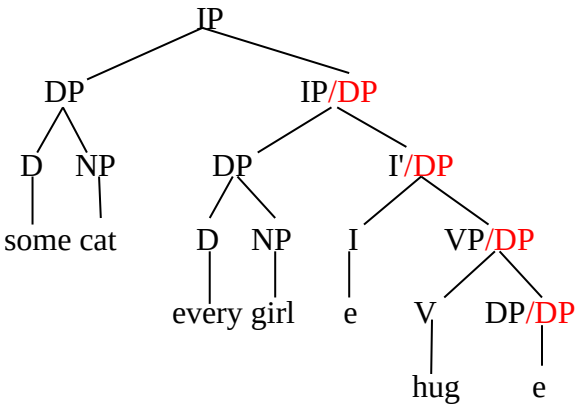
If we replace the DP by a DP-gap, then the top V' is itself no longer a V', but a V' with a DP missing in it:



Following this strategy, we can build up an IP/DP structure containing a *chain*:



At this point we can allow an operation that provides the head of the chain, and we get a topicalization structure. (all this can, of course, be made syntactically sophisticated).



We are interested in the semantic interpretation.

We start with the gap. DP/DP. Standardly, we would interpret a DP-gap as a free variable x. Here we think of the gap as a tree with 'itself' missing in it. We carry over that idea into the semantics: We would like to think of the meaning of the gap as a variable meaning, but it must really be a variable meaning with itself missing it, i.e. something that would be a variable if you applied it to a variable. This is just a complex way of describing the identity function, and since we are concerned with a variable of type e, we interpret the gap as:

DP/DP	→	λx.x <e,e>	with x a variable of type e
V	→	HUG_w <e,<e,t>>	

We would get at the VP level:

VP → APPLY[HUG_w, DP]
<e,t>

But we don't get that, we get:

VP/DP → HUG_w + DP/DP

And the reasoning about types tells us that VP/DP is not of type <e,t>, like VP, but has a DP meaning missing in it, i.e. is a function from DP-meanings to VP-meanings:

VP/DP → HUG_w + DP/DP
<e,<e,t>> <e,<e,t>> <e,t>
The operation involved is **function composition**:

$$\alpha \circ \beta = \lambda x.\alpha(\beta(x))$$

-Apply β to a variable x
-Apply α to the result β(x)
-Abstract over x.

You get the function that maps every x onto the result of **first** applying β to x, and **then** applying α to the result.

Instead of simple function application, we write APPLY:

Right Function Composition:
RC[α,β] = λx.APPLY[α, (β(x))]

VP/DP → HUG_w + DP/DP
<e,<e,t>> <e,<e,t>> <e,t>

RC[HUG_w, λx.x] = λz.APPLY[HUG_w, λx.x(z)]
= λz.APPLY[HUG_w, z]
= λz.HUG_w(z)
= HUG_w

Thus, we derive, correctly, a relational meaning for the VP/DP.

I'/DP → λP.P + VP/DP
<e,<e,t>> <<e,t>,<e,t>> <e,<e,t>>

The same operation of RC builds the correct meaning for I'/DP:
(the gap is, as before, inside the argument, and we use the same operation of RC)

$RC[\lambda P.P, HUG_w] = \lambda z.APPLY[\lambda P.P, HUG_w(z)]$
$= \lambda z.\lambda P.P(HUG_w(z))$
$= \lambda z.HUG_w(z)$
$= HUG_w$

At the next stage, the gap is in the function, rather than in the argument, since we chose the I' to be a function on the specifier. We pay the price for that decision here.
If we had switched the function-argument structure around, we could have continued with R-C. Now we must use L-C (since the gap is inside the function) and do a lot of type shifting:

$L\text{-}C = \lambda x.APPLY[\alpha(x), \beta]$

DP $\lambda P.\forall x[GIRL_w(x) \rightarrow P(x)]$

IP/DP $\rightarrow$ $L\text{-}C[HUG_w, \lambda P.\forall x[GIRL_w(x) \rightarrow P(x)]]$
$L\text{-}C[HUG_w, \lambda P.\forall x[GIRL_w(x) \rightarrow P(x)]] =$
$\lambda z.APPLY[HUG_w(z), \lambda P.\forall x[GIRL_w(x) \rightarrow P(x)]] = LIFT$
$\lambda z. \lambda T.T(HUG_w(z)) (\lambda P.\forall x[GIRL_w(x) \rightarrow P(x)]) =$
$\lambda z. \lambda P.\forall x[GIRL_w(x) \rightarrow P(x)](HUG_w(z)) =$
$\lambda z. \forall x[GIRL_w(x) \rightarrow HUG_w(x,z)]$

IP/DP $\rightarrow$ $\lambda z. \forall x[GIRL_w(x) \rightarrow HUG_w(x,z)]$
<e,t>
The property that you have if every girl hugs you

What function composition does in each stage of the derivation is:
-Temporarily apply the thing with the gap to a variable (of type type of the gap).
-Then do the operation that you would do if there were no gap (i.e. APPLY).
-After that, abstract over the variable again.

This way you keep the lambda-operation $\lambda z.$ on the outside of the expression all the time during the derivation.
So, not surprisingly, since IP is of type t, IP/DP is of type <e,t>.
This combines as a function to the DP-top of the chain:
Now the final operation is APPLY rather than compose. The types get balanced again at the top of the chain (normality is re-established, anything else is just your own problem):

IP $\rightarrow$ $APPLY[\lambda z. \forall x[GIRL_w(x) \rightarrow HUG_w(x,z)]. \lambda P\exists y[CAT_w(y) \wedge P(y)]$
This gets resolved in standard way, and we derive:

IP $\rightarrow$ $\exists y[CAT_w(y) \wedge \forall x[GIRL_w(x) \rightarrow HUG_w(x,y)]]$

We have derived one interpretation.

However, the topicalization sentence *Some cat every girl hugged* is scopally ambiguous, it allows both scope readings. The problem is: how do we get the narrow scope reading?

We know that the gap DP/DP has to have the interpretation of an identity function.
We also know that its interpretation will have to take narrow scope.
Above, we interpreted the gap as low as we could as λx.x, with an individual variable missing in it.
But for narrowest scope we want to hide inside the gap not an interpretation of type e, but a full interpretation at type <<e,t,>,t>, because that is the expression that takes narrow scope. So we allow an interpretation of the gap with an interpretation at the highest relevant type missing:

DP/DP → λT.T with T a variable of type <<e,t,>,t>
<<<e,t>,t>,<<e,t>,t>>

We follow the derivation:

VP/DP → RC[HUG_w, λT.T]

RC[HUG_w, λU.U] = [alphabetic variable]
λT.APPLY[HUG_w, λU.U(T)] =
λT.APPLY[HUG_w, T] = [$_{LIFT}$ HUG_w]
λT ([λUλx.U(λy.HUG_w(x,y))](T)) =
λTλx.T(λy.HUG_w(x,y)

It will not come as a surprise that this interpretation inherits with R-C up to I'/DP.
At the next step L-C will give:

IP/DP → L-C[λTλx.T(λy.HUG_w(x,y)), λP.∀x[$GIRL_w$(x) → P(x)]]

-we apply the function to a variable T, and get λx.T(λy.HUG_w(x,y)) of type <e,t>.
This applies in the normal way to the DP meaning, deriving:

∀x[$GIRL_w$(x) → T(λy.HUG_w(x,y))]

Finally, we abstract over T and get:

IP/DP → λT.∀x[$GIRL_w$(x) → T(λy.HUG_w(x,y))]

This applies to the topicalized DP:

IP → λT.∀x[$GIRL_w$(x) → T(λy.HUG_w(x,y))] (λP.∃y[CAT_w(y) ∧ P(y)])

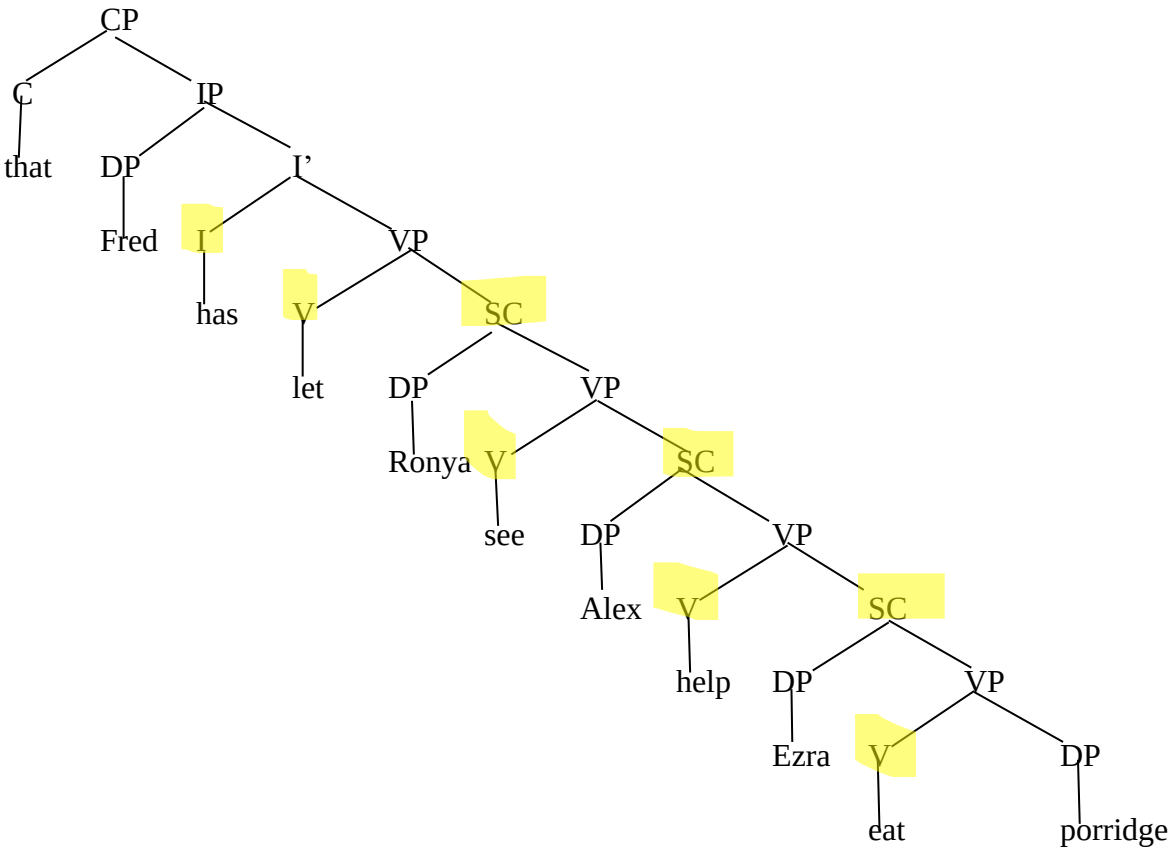
λ-conversion gives:

IP → $\forall x[GIRL_w(x) \rightarrow \exists y[CAT_w(y) \wedge HUG_w(x,y)]]$

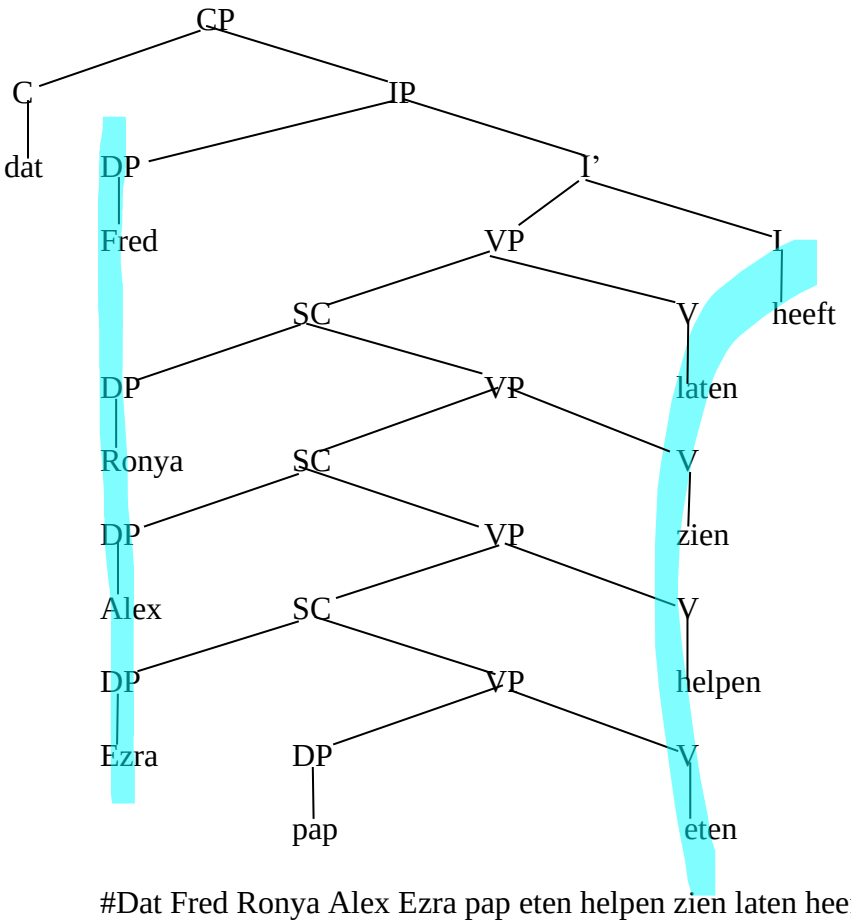
Thus we derive the narrow scope interpretation by giving a higher interpretation to the gap (with a 'big lambda'). This technique has also been applied to cases involving internal interpretations of external material in relative clauses involving intensional contexts (e.g. Sharvit 20??).

6.4.3. Naked infinitives and serial verbs in Dutch

that Fred has let Ronya see Alex help Ezra eat porridge
dat Fred Ronya Alex Ezra pap laten zien helpen eten heeft



In Dutch verbal heads are on the right, this would give the following structure:



#Dat Fred Ronya Alex Ezra pap eten helpen zien laten heeft.

This is not what we find, what we find is:

Dat Fred Ronya Alex Ezra pap ***laten zien helpen etnen*** heeft
Dat Fred Ronya Alex Ezra pap heeft ***laten zien helpen eten***

I will here only deal with the structure with the auxiliary at the end.

Constituent structure tests show that the verb sequence ***laten zien helpen eten*** is a constituent and that it is a constituent sitting in the lowest V position. This is because all of the following are also constituents:

laten zien helpen eten
pap laten zien helpen eten
Ezra pap laten zien helpen eten
Alex Ezra pap laten zien helpen eten

This can be shown with verb second in Dutch. In verb second the auxiliary *heeft* occurs in the C position, and in that case the specifier position of CP must be filled with a constituent. Everything that is a constituent in the above structure can occur felicitously in first position (unless putting it there violates syntax independently), nothing else can.

– **heeft** Fred Ronya Alex Ezra pap laten zien helpen eten

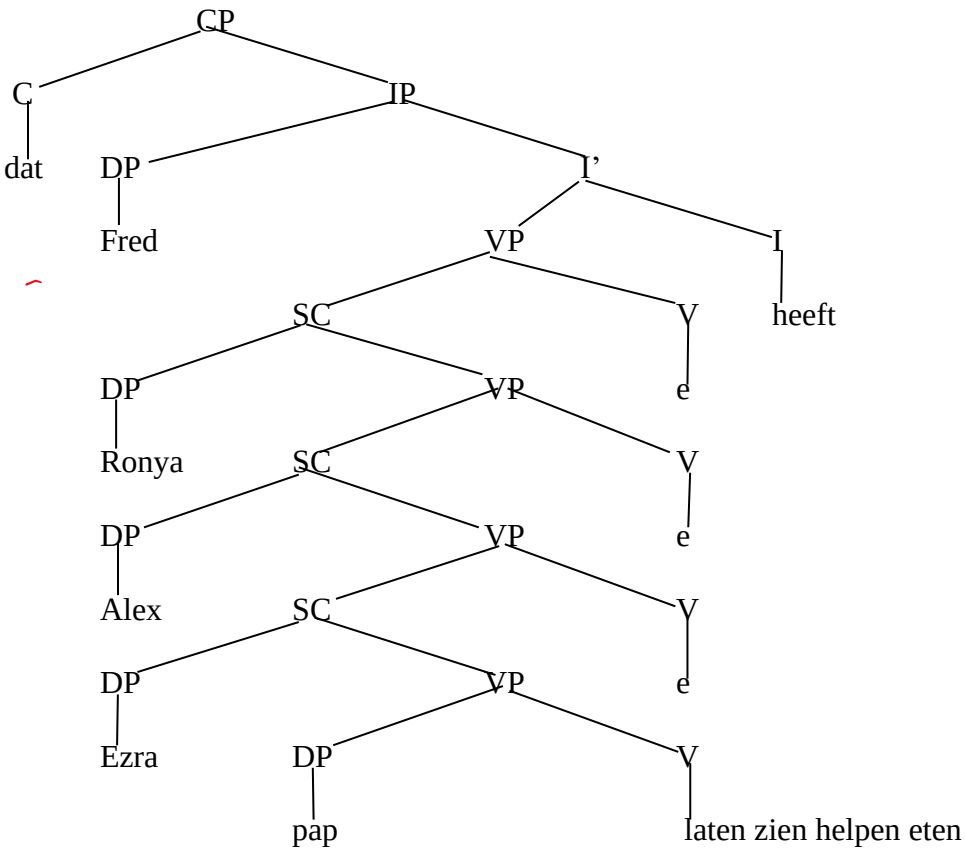
Fred **heeft** Ronya Alex Ezra pap laten zien helpen eten
Ronya **heeft** Fred Alex Ezra pap laten zien helpen eten
Alex **heeft** Fred Ronya Ezra pap laten zien helpen eten
Ezra **heeft** Fred Ronya Alex pap laten zien helpen eten
Pap **heeft** Fred Ronya Alex Ezra laten zien helpen eten

laten zien helpen eten **heeft** Fred Ronya Alex Ezra pap
pap laten zien helpen eten **heeft** Fred Ronya Alex Ezra
Ezra pap laten zien helpen eten **heeft** Fred Ronya Alex
Alex Ezra pap laten zien helpen eten **heeft** Fred Ronya
Ronya Alex Ezra pap laten zien helpen eten **heeft** Fred

Fred Ronya Alex Ezra pap laten zien helpen eten **heeft**
#[Fred e_n Ronya Alex Ezra pap laten zien helpen eten]$_k$ [**heeft**$_n$ [e_k]]]

The latter is a constituent but moves the trace of the verb over the landing site of the verb in C, which violates syntax in however way you may want to formulate verb second.

This argues for the following structure:



Syntactic – semantic mismatch:
The naked infinitive verbs *let, see, help* take small clause complements,
so you would think they are best analyzed as two place relations between individuals and propositions.

But semantically they are better analyzed as three place relations between two individuals and a property, the higher subject, the lower small clause subject and the small clause property.
The argument comes from the analysis given for Dutch below.

So:

have → $\mathbf{P}_w = \lambda P \lambda x.PERF_w(P(x))$

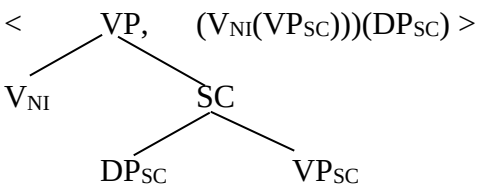
help → $\mathbf{H}_w = \lambda P \lambda y \lambda x.HELP_w(x,P(y))$ x helps y in w having property P.

let → $\mathbf{L}_w = \lambda P \lambda y \lambda x.LET_w(x,P(y))$ x lets y in w have property P.

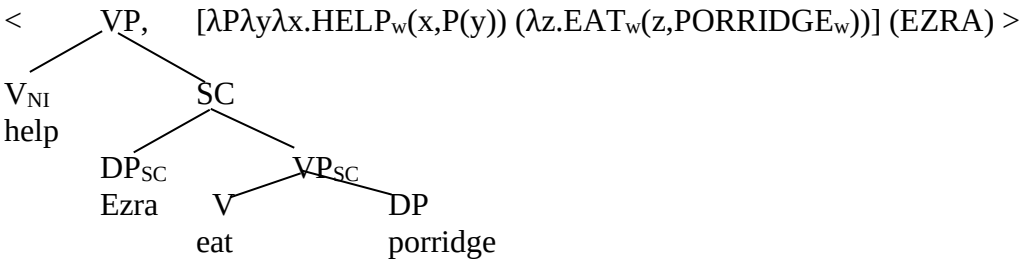
see → $\mathbf{S}_w = \lambda P \lambda y \lambda x.SEE_w(x,P(y))$ x sees y in w having property P.

I will ignore issues of the proper semantics of Naked Infinives (there is an event based proposal in Landman 2000), but concentrate at getting right who does what.

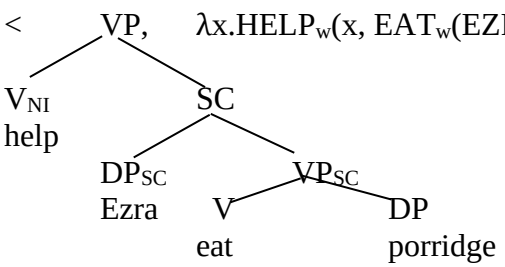
English grammar:



So:



$(\lambda P \lambda y \lambda x.HELP_w(x,P(y)) (\lambda z.EAT_w(z,PORRIDGE_w))(EZRA) > =$
$\lambda y \lambda x.HELP_w(x, \lambda z.EAT_w(z,PORRIDGE_w)(y)) (EZRA) =$
$\lambda y \lambda x.HELP_w(x, EAT_w(y,PORRIDGE_w)) (EZRA) =$
$\lambda x.HELP_w(x, EAT_w(EZRA,PORRIDGE_w)) =$



With this we derive:

that Fred has let Ronya see Alex help Ezra eat porridge

λw.PERF$_w$(LET$_w$(FRED, SEE$_w$(RONYA, HELP$_w$(ALEX, EAT$_w$(EZRA, PORRIDGE)))))

Dutch:

[$_V$ laten zien helpen eten]

In lexical domains function composition is a normal operation.
But we need to generalize function composition:

Generalized Function Composition:
$\alpha \circ \beta = \lambda x_n ... \lambda x_1 . \alpha(\beta(x_1,...x_n))$

The idea is: you want to apply α to β, but the types don't fit. Apply β to variables $x_1...,x_n$ to bring it to the input type of α (in Curried functional type theory, in order x_n then x_{n-1}... then x_1). Apply α, and abstract over the variables x_1 ...x_n.

So, I propose:

laten zien helpen eten → (**L**$_w$ ∘ (**S**$_w$ ∘ (**H**$_w$ ∘ **EAT**$_w$)))

The fact that this works is the direct motivation for the assumption that Naked Infinitive verbs are three place relations.
The analysis doesn't work if the semantics can be read off the syntactic structure, because then you would not allow the naked infinitive access inside the small clause, i.e. it would combine with a proposition,
not with the subject and the VP property seperately.

But the analysis with function compostion **does** work and is very insightful.

Here is a situation where it is useful to choose your variables in a fixed way and systematically apply: $[\lambda x_n.\varphi(x_n)](x_n)$ and get $\varphi(x_n)$.
The unnaturalness of variables chosen in the beginning, has mnemonic advantages in the end.

$\mathbf{H}_w \circ \mathbf{EAT}_w =$

$\lambda x_5 \lambda x_4.EAT(x_4,x_5)$ of type $<e^2,t>$ (= $<e,<e,t>>$)

$\lambda P \lambda x_4 \lambda x_3.HELP_w(x_3,P(x_4)) \circ \lambda x_5 \lambda x_4.EAT(x_4,x_5)$ =

$\mathbf{\lambda x_5}.[\lambda P \lambda x_4 \lambda x_3.HELP_w(x_3,P(x_4))\ \lambda x_5 \lambda x_4.EAT(x_4,x_5)\mathbf{(x_5)}\ )]$ =

$\mathbf{\lambda x_5}.[\lambda P \lambda x_4 \lambda x_3.HELP_w(x_3,P(x_4))\ (\lambda x_4.EAT_w(x_4,\mathbf{x_5})\ )]$ =

$\lambda x_5.[\lambda P \lambda x_4 \lambda x_3.HELP_w(x_3,P(x_4))\ (\lambda x_4.EAT_w(x_4,x_5)\ )]$ =

$\lambda x_5 \lambda x_4 \lambda x_3.HELP_w(x_3,(\lambda x_4.EAT_w(x_4,x_5)(x_4))$ =

$\lambda x_5 \lambda x_4 \lambda x_3.HELP_w(x_3, EAT_w(x_4,x_5))$ of type $<e^3,t>$

$\mathbf{S}_w \circ (\mathbf{H}_w \circ \mathbf{EAT}_w)) =$

$\lambda P \lambda x_3 \lambda x_2.SEE_w(x_2,P(x_3)) \circ \lambda x_5 \lambda x_4 \lambda x_3.HELP_w(x_3, EAT_w(x_4,x_5))$ =

Here we need to apply $\lambda x_5 \lambda x_4 \lambda x_3.HELP_w(x_3, EAT_w(x_4,x_5))$ to two variables to bring it down to <e,t>:

$\mathbf{\lambda x_5 \lambda x_4}[\ \lambda P \lambda x_3 \lambda x_2.SEE_w(x_2,P(x_3))\ (\lambda x_5 \lambda x_4 \lambda x_3.HELP_w(x_3, EAT_w(x_4,x_5))\mathbf{(x_4,x_5)}]$ =

$\mathbf{\lambda x_5 \lambda x_4}[\ \lambda P \lambda x_3 \lambda x_2.SEE_w(x_2,P(x_3))\ (\lambda x_3.HELP_w(x_3, EAT_w\mathbf{(x_4,x_5)})]$ =

$\lambda x_5 \lambda x_4[\ \lambda P \lambda x_3 \lambda x_2.SEE_w(x_2,P(x_3))\ (\lambda x_3.HELP_w(x_3, EAT_w(x_4,x_5))]$ =

$\lambda x_5 \lambda x_4 \lambda x_3 \lambda x_2.SEE_w(x_2,\lambda x_3.HELP_w(x_3, EAT_w(x_4,x_5))(x_3))$ =

$\lambda x_5 \lambda x_4 \lambda x_3 \lambda x_2.SEE_w(x_2, HELP_w(x_3, EAT_w(x_4,x_5)))$ of type $<e^4,t>$

$(\mathbf{L}_w \circ (\mathbf{S}_w \circ (\mathbf{H}_w \circ \mathbf{EAT}_w))) =$

$\lambda P \lambda x_2 \lambda x_1.LET_w(x_1,P(x_2)) \circ \lambda x_5 \lambda x_4 \lambda x_3 \lambda x_2.SEE_w(x_2, HELP_w(x_3, EAT_w(x_4,x_5)))$
=

Here we need to let $\lambda x_5\lambda x_4\lambda x_3\lambda x_2.SEE_w(x_2, HELP_w(x_3, EAT_w(x_4,x_5)))$ apply to three variables to bring it down to <e,t>

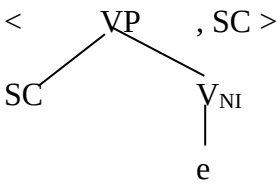
$\mathbf{\lambda x_5\lambda x_4\lambda x_3}[\lambda P\lambda x_2\lambda x_1.LET_w(x_1,P(x_2))$
$(\lambda x_5\lambda x_4\lambda x_3\lambda x_2.SEE_w(x_2, HELP_w(x_3, EAT_w(x_4,x_5)))\mathbf{(x_3,x_4,x_5)}]$
=

$\mathbf{\lambda x_5\lambda x_4\lambda x_3}[\lambda P\lambda x_2\lambda x_1.LET_w(x_1,P(x_2))\ (\lambda x_2.SEE_w(x_2, HELP_w(\mathbf{x_3}, EAT_w(\mathbf{x_4,x_5})))$
=
$\lambda x_5\lambda x_4\lambda x_3[\lambda P\lambda x_2\lambda x_1.LET_w(x_1,P(x_2))\ (\lambda x_2.SEE_w(x_2, HELP_w(x_3, EAT_w(x_4,x_5))))$
=
$\lambda x_5\lambda x_4\lambda x_3\lambda x_2\lambda x_1.LET_w(x_1,\lambda x_2.SEE_w(x_2, HELP_w(x_3, EAT_w(x_4,x_5))))(x_2))$
=
$\lambda x_5\lambda x_4\lambda x_3\lambda x_2\lambda x_1.LET_w(x_1,SEE_w(x_2,HELP_w(x_3,EAT_w(x_4,x_5))))$ of type $<e^5,t>$

So:

later zien helpen eten → $\lambda x_5\lambda x_4\lambda x_3\lambda x_2\lambda x_1.LET_w(x_1,SEE_w(x_2,HELP_w(x_3,EAT_w(x_4,x_5))))$
of type $<e^5,t>$

Dutch grammar:



The n-place relation interpretation of SC inherits up to the VP

$[_{VP}$ *pap laten zien helpen eten*] →
$\lambda x_4\lambda x_3\lambda x_2\lambda x_1.LET_w(x_1,SEE_w(x_2,HELP_w(x_3,EAT_w(x_4,PORRIDGE_w))))$ of type $<e^4,t>$

$[_{SC}$ *Ezra pap laten zien helpen eten*] →
$\lambda x_3\lambda x_2\lambda x_1.LET_w(x_1,SEE_w(x_2,HELP_w(x_3,EAT_w(EZRA,PORRIDGE_w))))$ of type $<e^3,t>$

$[_{VP}$ *Ezra pap laten zien helpen eten* e_V] →
$\lambda x_3\lambda x_2\lambda x_1.LET_w(x_1,SEE_w(x_2,HELP_w(x_3,EAT_w(EZRA,PORRIDGE_w))))$ of type $<e^3,t>$

$[_{SC}$ *Alex Ezra pap laten zien helpen eten* e_V] →
$\lambda x_2\lambda x_1.LET_w(x_1,SEE_w(x_2,HELP_w(ALEX,EAT_w(EZRA,PORRIDGE_w))))$ of type $<e^2,t>$

$[_{VP}$ *Alex Ezra pap laten zien helpen eten* e_V e_V] →
$\lambda x_2\lambda x_1.LET_w(x_1,SEE_w(x_2,HELP_w(ALEX,EAT_w(EZRA,PORRIDGE_w))))$ of type $<e^2,t>$

$[_{SC}$ *Ronya Alex Ezra pap laten zien helpen eten* $e_V\ e_V] \rightarrow$
$\lambda x_1.LET_w(x_1,SEE_w(RONYA,HELP_w(ALEX,EAT_w(EZRA,PORRIDGE_w))))$ of type <e,t>

$[_{VP}$ *Ronya Alex Ezra pap laten zien helpen eten* $e_V\ e_V\ e_V] \rightarrow$
$\lambda x_1.LET_w(x_1,SEE_w(RONYA,HELP_w(ALEX,EAT_w(EZRA,PORRIDGE_w))))$ of type <e,t>

$[_{I'}$ *Ronya Alex Ezra pap laten zien helpen eten* $e_V\ e_V\ e_V$ *heeft*$] \rightarrow$
$\lambda x_1.PERF_w(\ LET_w(x_1,SEE_w(RONYA,HELP_w(ALEX,EAT_w(EZRA,PORRIDGE_w)))))$
of type <e,t>
$[_{IP'}$ *Fred Ronya Alex Ezra pap laten zien helpen eten* $e_V\ e_V\ e_V$ *heeft*$] \rightarrow$
$PERF_w(\ LET_w(FRED,SEE_w(RONYA,HELP_w(ALEX,EAT_w(EZRA,PORRIDGE_w)))))$
of type t
$[_{CP}$ *dat Fred Ronya Alex Ezra pap laten zien helpen eten* $e_V\ e_V\ e_V$ *heeft*$] \rightarrow$
$\lambda w.PERF_w(\ LET_w(FRED,SEE_w(RONYA,HELP_w(ALEX,EAT_w(EZRA,PORRIDGE_w)))))$
of type <s,t>
This is, of course, the same interpretation as the one we derived for English.

German works the same as Dutch, except that the verbs are in the inverse order. But the constituent structure tests work the same, so: the only diffence is the order inside the complex V:

laten zien helpen eten
essen helfen sehen lassen

The Formal Languages discusses computational aspects of this structure in Dutch and German.

Natural languages have plenty lexical verbs that are grammatically one place relations, plently lexical verbs that are two place relations, some lexical verbs that are three place relations, and very few above that.
Serial verb constructions are constructions that build complex verbs (not VPs) that are n-place relations.