PART 6. SEMANTIC PARSING OF THE VERB CLUSTER IN DUTCH AND GERMAN

6.1 Basics of functional type theory, semantics of serial verbs via function composition.

Basics of functional type theory (ignoring intensionality)
This is rigorously done in Advanced Semantics (warning : the text below is horribly sloppy on blurring object language-meta language)

Types
e is the type of individuals, t is the type of truth-values
<a,b> is the type of functions from a-entities into b-entities.

<e,t> is the type of functions from individuals into truth values = one-place properties.

Functional application
If α is of type <a,b> and β is of type a, then (α(β)) is of type b. (<a,b> + a → b)

PURR	is of type <e,t>	one place property
Ronya	is of type e	individual
(PURR(Ronya))	is of type t	truth value

Curried functions
EAT[Pat, Pap] two-place relation between individuals
curried:
EAT is of type <e,<et>>
function from individuals into one place properties
functional application:
(EAT(pap)) is of type <e,t>, one place property
the property that you have if you eat pap
Functional application:
((EAT(pap)) (Pat) is of type t
Pat has the property that you have if you eat pap.

Relational notation: if R is of type <b,<a,c>> and β of type b and α of type a, then R(α,β) is relational notation for ((R(β))(α))

EAT(Pat, pap) is relational notation for ((EAT(pap)) (Pat))

Functional abstraction
If x is a variable of type a and β is of type b, then λx.β is of type <a,b>

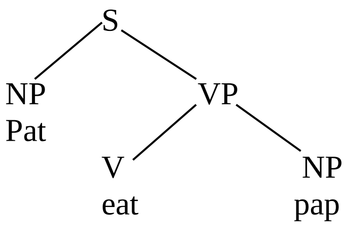
λx.β is the function that maps all a-entities d onto the interpretation of β[x:d], the interpretation of β that sets the interpretation of variable x to d.

lambda abstraction allows us to define complex functions.

$$\lambda x_n \ldots x_1.\beta(x_1,\ldots,x_n)$$

The lambda prefix $\lambda x_n \ldots x_1$.indicates the arguments that go into the function in the order $x_n \ldots x_1$.
$\beta(x_1,\ldots,x_n)$ is the description of the function, it tells you what the function does. In particular,
the order in $\beta(x_1,\ldots,x_n)$ indicates the argument (or thematic) structure (who does what to whom).

Example:



eat first combines with the object *pap*, and the result *eat pap* combines with the subject *Pat*:

eat → $\lambda y \lambda x.\mathrm{EAT}(x,y)$ the relation that holds between x and y if x eats y.
$(\lambda y \lambda x.\mathrm{EAT}(x,y)\ (\mathrm{pap}))$
$((\lambda y \lambda x.\mathrm{EAT}(x,y)\ (\mathrm{pap}))\ (\mathrm{Pat}))$

λ-conversion:
$(\lambda x.\beta\ (\alpha)) = \beta[\alpha/x]$ the result of replacing every variable x free in β by α, if no variable is bound in $\beta[\alpha/x]$ that was free in β

$(\boldsymbol{\lambda}\mathbf{y}\lambda x.\mathrm{EAT}(x,\mathbf{y})\ \mathbf{(pap)}) = \lambda x.\mathrm{EAT}(x,\mathbf{pap})$

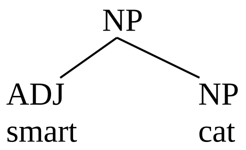
$((\lambda y \lambda x.\mathrm{EAT}(x,y)\ (\mathrm{pap}))\ (\mathrm{Pat})) = (\lambda x.\mathrm{EAT}(x,\mathrm{pap})\ (\mathrm{Pat}))$

$(\boldsymbol{\lambda}\mathbf{x.}\mathrm{EAT}(\mathbf{x},\mathrm{pap})\ (\mathbf{Pat})) = \mathrm{EAT}(\mathbf{Pat}, \mathrm{pap})$

Higher order abstraction
Example: attributive adjectives:

smart $\rightarrow \lambda P\lambda x.P(x) \wedge \text{SMART}(x)$ of type <<e,t>,<e,t>>,

P is a variable of type <e,t>, a variable over properties.
smart denotes a function from one place properties into one place properties
maps property Q onto the property that you have if you have Q and you are smart.

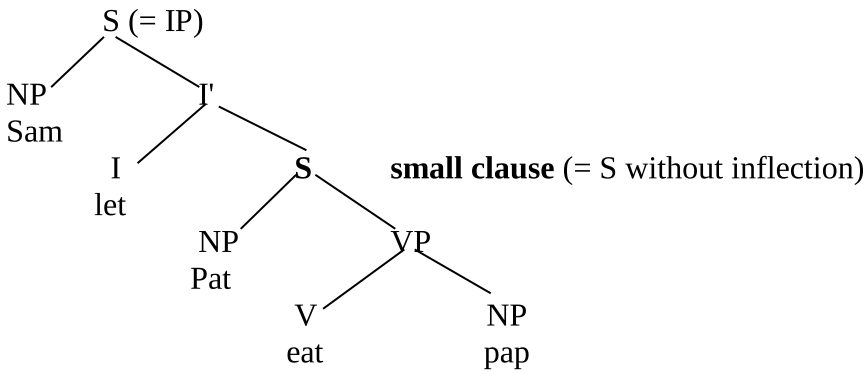


smart cat = $(\lambda P\lambda x.P(x) \wedge \text{SMART}(x)\ (\text{CAT}))$

λ-conversion:

$$(\boldsymbol{\lambda P}\lambda x.\mathbf{P}(x) \wedge \text{SMART}(x)\ (\mathbf{CAT})) = \lambda x.\mathbf{CAT}(x) \wedge \text{SMART}(x)$$

The property that you have if you are a cat and you are smart.
Small clause analysis for help/let/see… auxiliary verbs:



Semantics matching the syntax exactly (simplifying by ignoring intensionality)

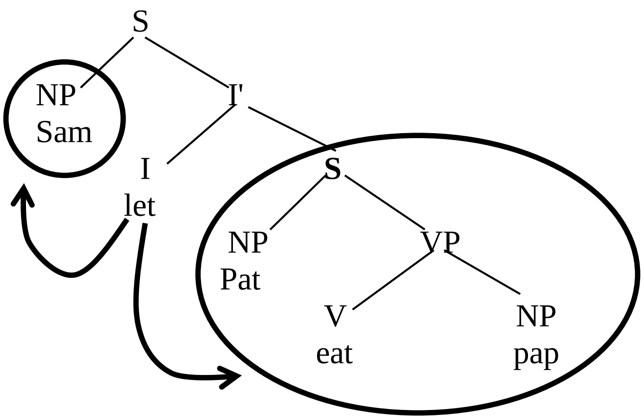
let $\rightarrow$ $\lambda p\lambda x.\text{LET}(x,p)$

p a variable over propositions, for simplicity here identified with sentences (type t, this is incorrect, but will do for pour purposes here).

Semantics of the small clause: ((EAT(pap)) (Pat))
= EAT(Pat, pap)
Semantics of the sentence: (LET(EAT(Pat, pap)) (Sam))
= LET(Sam, EAT(Pat, pap))

Sam stands in the LET relation to the proposition that Pat eats pap.
Lexical semantics of LET: LET(x,p) x allows p to happen (where p is under x's control, etc.)

Semantics:



two place relation between an individual and a proposition

Non-matching semantics

let → λPλyλx.LET(x,P(y))

Syntactically *eat* takes a small clause complement and a higher subject. Semantically *eat* accesses the subject of the small class, the predicate of the small clause and the higher subject, i.e. is a three-place relation, it applies to the the small clause predicate, the small clause subject and the higher subject.

So:

Small clause predicate:
eat pap → (EAT(pap))

let eat pap → (**λP**λyλx.LET(x,**P**(y)) (**EAT(pap)**)

λ-conversion:
λyλx.LET(x,(**EAT(pap)**) (y)))

Relational notation:
λyλx.LET(x, EAT(y,pap))

let Pat eat pap → (**λy**λx.LET(x, EAT(**y**,pap)) (**Pat**))

λ-conversion: λx.LET(x, EAT(**Pat**,pap))

Sam let Pat eat pap → (**λx**.LET(**x**, EAT(Pat,pap)) (**Sam**))

λ-conversion: LET(**Sam**, EAT(Pat,pap))

lexical semantics: LET(x, P(y)): x allows it to happen that y has P.

So: Sam lets it happen that Pat eats her pap.

-The small clause syntax is for a variety of syntactic reasons better than a syntax mirroring the semantics.
-The 3-place relational semanrics is for a variety of semantic reasons better than a semantics mirroring the syntax.
Hence: mismatches between syntax and semantics.

(Other examples: Landman 2000, 2004 on numericals in nominal versus argument or predicate position, 2016 on measure versus classifier readings).

Semantic argument: the semantics given here allows a very elegant semantics for the verb cluster in terms of function composition.

function composition:

If f is a function of type <a,b> and g a function of type <b,c>
then $g \circ f$ is a function of type <a,c> $(<a,b> + <b,c> \rightarrow <a,c>)$

$g \circ f = \lambda x.g(f(x))$

Apply f to variable x of type a:	$f(x)$	is of type b
Apply g to the result:	$g(f(x))$	is of type c
Abstract over x:	$\lambda x.g(f(x))$	is of type <a,b>

$\lambda x.g(f(x))$ The function that maps every x of type a onto the result of applying f to x and
then g to the result.

generalized function composition:
Resolve a type mimatch through function composition:
You want to compose function g with function f.
So apply f to a variable x. But f(x) isn't of the right type to be fed into g.
Solution: continue to apply f to variables (i.e. apply f(x) to a variable y, etc. until the types match.
Apply g to the result and abstract over all variables you have applies f to:

$$\mathrm{COMP}[g,f] = \lambda x_n \ldots \lambda x_1.g(f(x_1,\ldots,x_n))$$

(where the types of $x_1 \ldots x_n$ can be anything and $f(x_1,\ldots x_n)$ should be understood curried:
$(\ldots((f(x_n))(x_{n-1}))\ldots(x_1)))$

Fact:
Let R^n be an n-place relation between individuals (of type <e,…,<e,t>> with n e's).
Let **LET** = $\lambda P \lambda y \lambda x.LET(x,P(y))$

Then: COMP[**LET**, R^n] is an n+1 place relation between individuals

Proof:
We calculate COMP[**LET**, R^n]
Step 1: apply R^n to variables $x_2,\ldots,x_n$ to bring the type down to <e,t>, the type of variable P:

$R^n(x_2,\ldots,x_n)$ is a one place predicate.

Step 2: apply **LET** to $R^n(x_2,\ldots,x_n)$:

$(\boldsymbol{\lambda P}\lambda y \lambda x.LET(x,\mathbf{P}(y)))\ (\mathbf{R^n(x_2,\ldots,x_n)})$

λ-conversion: $\lambda y \lambda x.LET(x, \mathbf{R^n(x_2,\ldots,x_n)}\ (y))$

Relational notation: $\lambda y \lambda x.LET(x, R^n(y,x_2,\ldots,x_n))$

Step 2: abstract over variables $x_2,\ldots,x_n$:

$\lambda x_n \ldots \lambda x_2 \lambda y \lambda x.LET(x, R^n(y,x_2,\ldots,x_n))$

Step 3: Rename variables for clarity:

$\lambda x_{n+1} \ldots \lambda x_2 \lambda x_1.LET(x_1, R^n(x_2,\ldots,x_{n+1}))$ an n+1 place relation

With this we understand what happens in the verb cluster:

COMP[**LET** ,EAT]

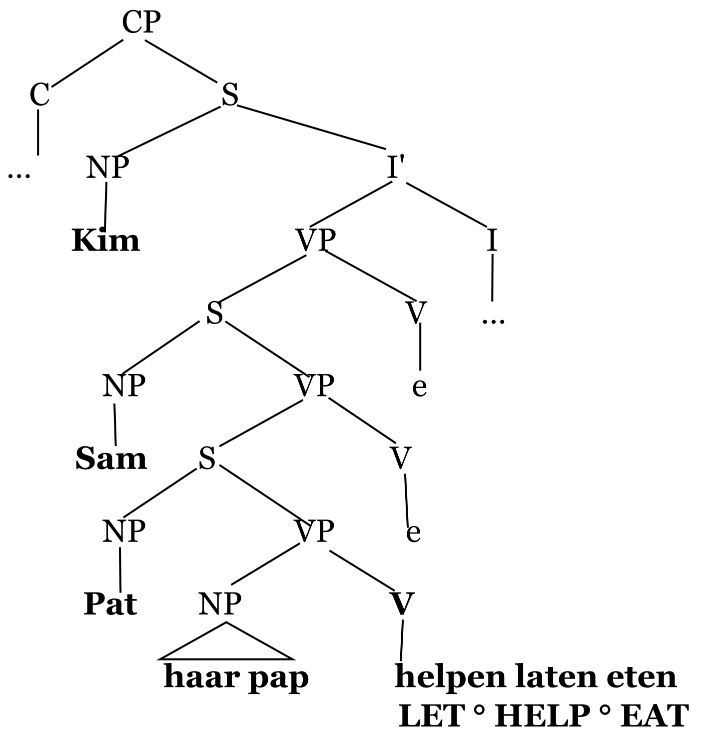
	$\lambda u \lambda v.EAT(v,u)$	2-place relation
Apply to a variable z:	$\boldsymbol{\lambda u}\lambda v.EAT(v,\mathbf{u})\ (\mathbf{z})$	
λ-conversion:	$\lambda v.EAT(v,\mathbf{z})$	1-place property
Apply **LET**:	$(\boldsymbol{\lambda P}\lambda y \lambda x.LET(x,\mathbf{P}(y)))\ (\boldsymbol{\lambda v.EAT(v,z)})$	
λ-conversion:	$\lambda y \lambda x.LET(x, \boldsymbol{\lambda v.EAT(v,z)}\ (y))$	
	$\lambda y \lambda x.LET(x, \boldsymbol{\lambda v}.EAT(v,z)\ (\mathbf{y}))$	
λ-conversion:	$\lambda y \lambda x.LET(x, EAT(\mathbf{y},z))$	
abstract over z:	$\lambda z \lambda y \lambda x.LET(x, EAT(y,z))$	3-place relation

So given the semantics for *let*, function composition in the verb cluster of *let* and two place relation *eat* forms three-place relation *let eat*.

-In **phrasal domains** (IP (S), VP), the basic **meaning composition** operation is function-argument application.
-In **lexical domains** (like V), the basic **meaning composition** operation is function composition. (Hoeksema 19??)

Claim: The mismatch assumption and the composition assumption are the only assumptions that need to be made to get the semantics of the serial verb cluster come out right.

Example: (ignoring the modal *zal*)



Predicted Semantics:

$\mathbf{LET} = \lambda P \lambda y \lambda x.\ \mathbf{LET}(x, P(y))$ $\qquad$ $\mathbf{EAT} = \lambda z \lambda y.\ \mathbf{EAT}(y,z)$

$\mathbf{LET} \circ \mathbf{EAT} =$ $\qquad$ $\lambda P \lambda y \lambda x.\ \mathbf{LET}(x, P(y)) \circ \lambda z \lambda y.\ \mathbf{EAT}(y,z) =$

$\lambda z\ [\lambda P \lambda y \lambda x.\ \mathbf{LET}(x, P(y))]\ (\ \boldsymbol{\lambda z} \lambda xy.\ \mathbf{EAT}(y,\mathbf{z})\ (z)\ ) =$
$\lambda z\ [\boldsymbol{\lambda P} \lambda y \lambda x.\ \mathbf{LET}(x, \mathbf{P}(y))]\ (\lambda y.\ \mathbf{EAT}(y,z)\ ) =$
$\lambda z\ [\ \lambda y \lambda x.\ \mathbf{LET}(x, \boldsymbol{\lambda y}.\ \mathbf{EAT}(\mathbf{y},z)\ (y))] =$

$\lambda z \lambda y \lambda x.\ \mathbf{LET}(x, \mathbf{EAT}(y,z))$ $\qquad$ *x lets*: *y eat z*

Conclusion:

2 place verb	**+ LET**	$\rightarrow$	**3-place verb**
$\lambda z \lambda y.\ \mathbf{EAT}(y,z)$	**LET**		$\lambda z \lambda y \lambda x.\ \mathbf{LET}(x, \mathbf{EAT}(y,z))$
1 2			**1 2 3**

Next:
HELP ° (LET ° EAT) =

$\lambda P \lambda x \lambda u.\ \mathbf{HELP}(u, P(x)) \circ \lambda z \lambda y \lambda x.\ \mathbf{LET}(x, \mathbf{EAT}(y,z)) =$

$\lambda z \lambda y\ [\lambda P \lambda x \lambda u.\ \mathbf{HELP}(u, P(x))]\ (\boldsymbol{\lambda z \lambda y} \lambda x.\ \mathbf{LET}(x, \mathbf{EAT}\mathbf{(y,z)})\ (y,z)) =$
$\lambda z \lambda y\ [\boldsymbol{\lambda P} \lambda x \lambda u.\ \mathbf{HELP}(u, \mathbf{P}(x))]\ (\lambda x.\ \mathbf{LET}(x, \mathbf{EAT}(y,z))\ ) =$
$\lambda z \lambda y \lambda x \lambda u.\ \mathbf{HELP}(u, \mathbf{LET}(x, \mathbf{EAT}(y,z)))$

3 place verb **+ HELP→** **4-place verb**

λzλyλx. **LET**(x, **EAT**(y,z)) λzλyλxλu. **HELP**(u, **LET**(x, **EAT**(y,z)))

1 **2 3** **1** **2** **3 4**

So: **helpen laten eten →** λzλyλxλu. **HELP**(u, **LET**(x, **EAT**(y,z)))

u helps x; x lets y, y eats z.

-The basic composition operation in phrasal domains is **function-argument application:**

Type theory:

The left-rightorder in the λ-prefix represents the order of application.

-The meaning of the V ***helpen laten eten*** applies to the meaning of ***haar pap*** (λz)
-the result applies to the meaning of ***Pat,*** (λy),
-the result to the meaning of ***Sam*** (λx),
-the result to the meaning of ***Kim*** (λu),
giving, the correct meaning for the sentence:

APPLY[, Kim]
APPLY[, Sam]
APPLY[, Pat]
APPLY[HELP ° (LET ° EAT), Her Porridge]

=

HELP(**Kim**, **LET**(**Sam**, **EAT**(**Pat**,**Her Porridge**)))

The semantics proposed gets the meanings right within a framework of standard assumptions about the semantic composition operations applicable in different domains (composition and application).

Given the meanings of the verbs that enter into the serial verb, composition has the effect of:

n-PLACE SERIAL VERB FORMATION:

Let α be one of LET, HELP, SEE, HEAR ,…
Let β be an n−1 place relation, then α ° β is an n-place relation.

6.2. Semantic Parsing

Bach et. al. 1986 performed a cross-linguistic experiment.
The idea of the experiment was the following: Dutch and German are similar enough to be able to compare the speed of processing of the Dutch verb cluster by native speakers of Dutch with that of the German verb cluster by native speakers of German. What Bach et. al. measured for Dutch and for German was the following.

Let's use the numbers 0,1,2,3,... for 0 embeddings, 1 embedding, 2 embeddings,... as indicated below:

German:

0 Jan weiβt daβ wir --- das Haus haben wollen malen ---
1 Hans lassen
2 Hans Peter helfen lassen
3 Hans Peter Marie sehen helfen lassen
NP_1 NP_2 NP_3 V_3 V_2 V_1

Center embedded dependencies.

Dutch:

0 Jan weet dat we --- het huis hebben willen --- schilderen
1 Hans laten
2 Hans Peter laten helpen
3 Hans Peter Marie laten helpen zien
NP_1 NP_2 NP_3 V_1 V_2 V_3

Cross serial dependencies.

Bach et. al. measured, in terms of processing time, in each language, how much longer type 1 sentences take to process than type 0, how much longer type 2 than type 1, etc. And then they compared the figures they got for Dutch and for German. They found that indeed type 0 in Dutch and in German (and in English) take about the same time (which forms the basis for comparison). Interestingly enough they found the following:

> **Systematically Dutch speakers process n Dutch embeddings FASTER than German speakers process n German embeddings.**

I want to suggest a possible explanation of this result in terms of **semantic parsing**. (For an alternative explanation, see Joshi 1989.)

Let's first explain the idea of semantic parsing.

We have an input string which is read symbol by symbol from left to right:

> *dat Jan Marie kust* [that Jan kisses Marie]

In semantic parsing the task is to come up with a semantic interpretation of the sentence. And we use the **types** of the input expressions and the type assignment of the grammar to do that online. Basically the idea is to find the values for the variables introduced in the parsing process, or equivalently, eliminate the variables. The parse is done when all variables are eliminated.
We indicate in boldface where we are in the parse:

Step 1 ***dat*** *Jan Marie kust*
Semantics: φ [$\varphi \in VAR_t$, a sentential variable]
Task: find the value of φ.

Step 2 *dat* ***Jan*** *Marie kust*
Semantics: $\varphi = P^1(j)$ [$P^1 \in VAR_{<e,t>}$, a one-place predicate variable]
Task: find the value of P^1.

Step 3 *dat Jan* ***Marie*** *kust*
Semantics: $\varphi = P^1(j)$
$P^1 = P^2(m)$ [$P^2 \in VAR_{<e,<e,t>>}$, a two place predicate variable]

This means that we can eliminate P^1:
Semantics: $\varphi = P^2(j,m)$

Task: find the value of P^2.

Step 4 *dat Jan Marie* ***kust***
Semantics: $\varphi = P^2(j,m)$
$P^2 = KISS$

We eliminate P^2:
Semantics: $\varphi = KISS(j,m)$
We eliminate φ:
Semantics: KISS(j,m)

Done.

Applying the very same strategy in the verb cluster, we get for Dutch (and for German) the following partial parse: (LF stands for: 'look for the value of')

dat	Kim	Sam	Pat	**haar pap** zal helpen laten eten.
SEM: φ	SEM: $P^1(f)$	SEM: $P^2(f,s)$	SEM: $P^3(f,s,d)$	SEM: $P^4(k,s,p,pap)$
LF: φ	LF: P^1	LF: P^2	LF: P^3	LF: P^4

So at the point where we reach the verb cluster, the parser is looking for a **four-place relation**.

We are at the following stage in Dutch:

dat Kim Sam Pat haar pap **zal** helpen laten eten.
  SEMANTICS: P^4(k,s,p,pap)
  LOOK FOR: P^4

In German we are, similarly at the following stage:

daβ Kim Sam Pat ihr Brei **essen** lassen helfen wird.
  SEMANTICS: P^4(k,s,p,pap)
  LOOK FOR: P^4

In both cases, we are looking for a **four-place relation** P^4 and we rely on function composition to find it.

Let's argue the German case first.
We are looking for a four place relation. But *essen* is a two-place relation.
So we are stuck.
At this point we **start a store in which we build a four place relation**:

daβ Kim Sam Pat ihr Brei **essen** lassen helfen wird.
  SEMANTICS: $\varphi = P^4$(k,s,p,pap)
  LF: P^4
  STORE: **EAT** (a two place relation)

We continue and at the next step we apply function composition in the store:

daβ Kim Sam Pat ihr Brei essen **lassen** helfen wird.
  SEMANTICS: $\varphi = P^4$(k,s,p,pap)
  LF: P^4
  STORE: **LET** o **EAT** (a three-place relation)

  LET o **EAT =**
  λzλyλx.LET(x,EAT(y,z))

We continue to *helfen* and again do function composition on the store:

daβ Kim Sam Pat ihr Brei essen lassen **helfen** wird.
  SEMANTICS: $\varphi = P^4$(k,s,p,pap)
  LF: P^4
  STORE: **HELP** o (**LET** o **EAT**) (a four-place relation)

  HELP o (**LET** o **EAT**) =
  λuλzλyλx.HELP(x,LET(y,EAT(z,u)))

We have now a four-place relation in store, but the parse continues inside V, so we continue with function composition:

daβ Kim Sam Pat ihr Brei essen lassen helfen **wird.**

SEMANTICS: $\varphi = P^4(k,s,p,pap)$

LF: P^4

STORE: WILL o (HELP o (**LET** o **EAT))** (a four-place relation)

WILL o (HELP o (**LET** o **EAT)) =**

λuλzλyλx.WILL(HELP(x,LET(y,EAT(z,u))))

This completes the parse in the V domain, we match the store and P^4:

daβ Kim Sam Pat ihr Brei essen lassen helfen wird.

SEMANTICS: $\varphi = P^4(k,s,p,pap)$

LF: P^4 = **WILL o (HELP** o (**LET** o **EAT))**

We eliminate variables and get the correct parse:

daβ Kim Sam Pat ihr Brei essen lassen helfen wird.

SEMANTICS: WILL(HELP(k,LET(s,EAT(p,pap))))

DONE

In Dutch we have exactly the same option as in German, we can create a store:

dat Kim Sam Pat haar pap **zal** helpen laten eten.

SEMANTICS: $P^4(k,s,p,pap)$

LOOK FOR: P^4

STORE: WILL

We continue in the store with function composition:

dat Kim Sam Pat haar pap zal **helpen** laten eten.

SEMANTICS: $P^4(k,s,p,pap)$

LOOK FOR: P^4

STORE: WILL o HELP

WILL o HELP =

$\lambda P^1\lambda y\lambda x.WILL(HELP(x,P^1(y)))$

We continue with *laten*:

dat Kim Sam Pat haar pap zal helpen **laten** eten.

SEMANTICS: $P^4(k,s,p,pap)$

LOOK FOR: P^4

STORE: (WILL o HELP) o LET

(WILL o HELP) o LET =

$\lambda P^1\lambda z\lambda y\lambda x.WILL(HELP(x,LET(y,P^1(z))))$

We compose with *eten*:

dat Kim Sam Pat haar pap zal helpen laten **eten**.

SEMANTICS: P^4(k,s,p,pap)

LOOK FOR: P^4

STORE: ((WILL o HELP) o LET) o EAT

((WILL o HELP) o LET) o EAT =

λuλzλyλx.WILL(HELP(x,LET(y,EAT(z,u))))

We are done in the V-domain, we have the same relation in store as in German, so we match, eliminate variables and are done:

dat Kim Sam Pat haar pap zal helpen laten **eten**.

SEMANTICS: WILL(HELP(k,LET(s,EAT(p,pap))))

DONE

Thus far, there is no difference between the Dutch and the German case. The difference comes in with the following observation:

Dutch allows a straightforward alternative parsing strategy that does not involve a store at all.

We go back to the point where we switched from functional application to function composition:

dat Kim Sam Pat haar pap **zal** helpen laten eten.

SEMANTICS: P^4(k,s,p,pap)

LOOK FOR: P^4

We continue the parse by introducing search variable Q^4 and compose as follows:

dat Kim Sam Pat haar pap zal **helpen** laten eten.

SEMANTICS: P^4(k,s,p,pap)

P^4 = **WILL** o Q^4

LOOK FOR: P^4

So we get:

dat Kim Sam Pat haar pap zal **helpen** laten eten.

SEMANTICS: [**WILL** o Q^4] (k,s,p,pap)

LOOK FOR: Q^4

And we continue by introducing a search variable P^3 and compose as follows:

dat Kim Sam Pat haar pap zal helpen **laten** eten.

SEMANTICS: [**WILL** o (**HELP** o P^3)] (k,s,p,pap)

LOOK FOR: P^3

We continue by introducing search variable P^2 and compose similarly:

dat Kim Sam Pat haar pap zal helpen laten **eten**.

SEMANTICS: [**WIIL o (HELP o (LET o** P^2))] (k,s,p,pap)
LOOK FOR: P^2

At this point we reach the end of the V and we resolve:

dat Kim Sam Pat haar pap zal helpen laten **eten**.
SEMANTICS: [**WIIL o (HELP o (LET o EAT**))] (k,s,p,pap)

The result is the same:

dat Kim Sam Pat haar pap zal helpen laten **eten**.
SEMANTICS: WILL(HELP(k,LET(s,EAT(p,pap))))
DONE

Thus, on this strategy, we just continue to compose on.
In this parse, we do not use a store, and we only need to introduce **six search variables all in all**: φ, P^1,P^2,P^3,P^4, Q^4 **of five different types.**

Now, composition is a powerful mechanism, so it shouldn't come as a surprise that also for German we can find a direct parse that doesn't rely on the store.

The parse in German can continue directly as follows:

P^4(k,s,p,pap)
LF: P^4

(R^3 o **EAT**)(k,s,p,pap) where R^3 is a variable of type <<e,t>,<e,<e,<e,t>>>>>
LF: R^3

(R^2 o (**LET** o **EAT**))(k,s,p,pap) where R^2 is a variable of type <<e,t>,<e,<e,t>>>
LF: R^2

(R^1 o (**HELP** o (**LET** o **EAT**)))(k,s,p,pap) where R^1 is a variable of type <<e,t>,<e,t>>
LF: R^1

(**WILL** o (**HELP** o (**LET** o **EAT**)))(k,s,p,pap)

which is:

WILL(HELP(k,LET(s,EAT(p,pap))))

This parse takes as many steps as the Dutch parse, and doesn't use a store either.
It differs from the Dutch parse, though, in that **it introduces more search variables than the Dutch parse**: **eight search variables all in all:** φ, P^1,P^2,P^3,P^4,R^3,R^2,R^1 **of eight different types**.

If we make the plausible hypotheses that **using a store is costly**, and that **using more search variables of more different type is costly** it follows that both parsing strategies potentially available in German are **more costly** than the fastest strategy available in Dutch. And this is what Bach et. al. 1986 found.