# Semantik, Modul 1003

## Semantic types, $\lambda$-calculus, different kinds of predicates

Heim & Kratzer (1998), ch. 2.5-3

Leipzig University

April $18^{th}$, 2024

Fabian Heck
Slides by Imke Driemel

# Recap: functions, sets, predicate logic

- in set-notation, the concept of set-membership is central to truth
- in function-notation, predicate saturation – involving the application of an argument to the characteristic function – yields truth
- a formal system is a syntactic object, a set of expressions and rules of combination and derivation
- we use Predicate Logic as a formal system to analyze natural languages
- our formal system is always interpreted based on a model: a pair $\langle D, I \rangle$, where $D$ is the domain, a set of individuals, and $I$ is an interpretation function: an assignment of semantic values to every basic expression (constant) in the language

# Recap: composition rules

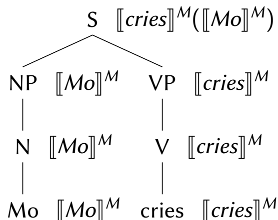
Last week we came up with composition rules to interpret syntactic trees that serve as the input for semantic computation.

(1) a. **Terminal node rule:** the denotation for a terminal node comes from the lexicon

b. **Non-branching node rule:** If $\alpha$ is a node whose only daughter is $\beta$, then $[\![\alpha]\!]^M = [\![\beta]\!]^M$

c. **Function application:** If $\alpha$ is a node whose daughter nodes are $\beta$ and $\gamma$, where $\gamma$ is a function whose domain contains $[\![\beta]\!]^M$, then $[\![\alpha]\!]^M = [\![\gamma]\!]^M([\![\beta]\!]^M)$

(2) Let $\alpha =$



# Recap: composition rules

We substitute lexical items by their denotations and get:

(3)
$$\left[\!\left[ \text{[S [NP [N Mo]] [VP [V cries]]]} \right]\!\right]^M = [\![cries]\!]^M([\![Mo]\!]^M) = \begin{bmatrix} \text{Nat} & \rightarrow & 1 \\ \text{Mo} & \rightarrow & 1 \\ \text{Dee} & \rightarrow & 1 \\ \text{Jean} & \rightarrow & 0 \end{bmatrix}(Mo) = 1$$

We can also write this by substitution with denotations:

(4)
$$\left[\!\left[ \text{[S [NP [N Mo]] [VP [V cries]]]} \right]\!\right]^M = [\![cries]\!]^M([\![Mo]\!]^M) = \begin{bmatrix} \text{f: } D_M \rightarrow \{1,0\} \text{ such that} \\ \text{for all } x \in D_M.\ f(x) = 1 \\ \text{iff } x \text{ cries} \end{bmatrix}(Mo) = 1 \text{ iff Mo cries}$$

# More on composition rules

We started applying our composition rules to transitive verbs, i.e. two-place predicates. We used right-to-left *Schönfinkelization* to derive nested one-place-predicates from two-place predicates.

(5) $[\![dislike]\!]^{M'} = \{\langle Nat, Dee\rangle, \langle Mo, Dee\rangle\}$

(6) $\left[\!\!\left[ [_S\ [_{NP}\ [_N\ \text{Mo}]]\ [_{VP}\ [_V\ \text{dislikes}]\ [_{NP}\ [_N\ \text{Dee}]]]] \right]\!\!\right]^{M'} = ([\![dislikes]\!]^{M'}([\![Dee]\!]^{M'}))([\![Mo]\!]^{M'})$

$$= \left( \begin{bmatrix} \text{Nat} \to \begin{bmatrix} \text{Dee} \to 0 \\ \text{Mo} \to 0 \\ \text{Nat} \to 0 \end{bmatrix} \\ \text{Dee} \to \begin{bmatrix} \text{Nat} \to 1 \\ \text{Mo} \to 1 \\ \text{Dee} \to 0 \end{bmatrix} \\ \text{Mo} \to \begin{bmatrix} \text{Mo} \to 0 \\ \text{Dee} \to 0 \\ \text{Nat} \to 0 \end{bmatrix} \end{bmatrix} (Dee) \right) (Mo) = 1$$

# More on composition rules

Question: How do we get to the truth value for the following sentence/tree?

(7) $[\![dislike]\!]^{M'} = \{\langle Nat, Dee\rangle, \langle Mo, Dee\rangle\}$

(8) $\left[\!\!\left[ [_S [_{NP} [_N \text{Dee}]] [_{VP} [_V \text{dislikes}] [_{NP} [_N \text{Mo}]]]] \right]\!\!\right]^{M'}$ = $([\![\quad]\!]^{M'}([\![\quad]\!]^{M'}))([\![\quad]\!]^{M'})$

$= \left( \left[ \quad \right] (\quad) \right) (\quad) =$

# More on composition rules

Question: How do we get to the truth value for the following sentence/tree?

(7) $[\![dislike]\!]^{M'} = \{\langle Nat, Dee\rangle, \langle Mo, Dee\rangle\}$

(8) $\left[\!\!\left[ \text{[S [NP [N Dee]] [VP [V dislikes] [NP [N Mo]]]]} \right]\!\!\right]^{M'}$ = $([\![dislikes]\!]^{M'}([\![Mo]\!]^{M'}))([\![Dee]\!]^{M'})$

$= \left(\left[\quad\right](\quad)\right)(\quad) =$

# More on composition rules

Question: How do we get to the truth value for the following sentence/tree?

(7) $[\![dislike]\!]^{M'} = \{\langle Nat, Dee\rangle, \langle Mo, Dee\rangle\}$

(8) $\left[\!\!\left[ \text{[S [NP [N Dee]] [VP [V dislikes] [NP [N Mo]]]]} \right]\!\!\right]^{M'}$ = $([\![dislikes]\!]^{M'}([\![Mo]\!]^{M'}))([\![Dee]\!]^{M'})$

$$= \left( \begin{bmatrix} \text{Nat} \rightarrow \begin{bmatrix} \text{Dee} \rightarrow 0 \\ \text{Mo} \rightarrow 0 \\ \text{Nat} \rightarrow 0 \end{bmatrix} \\ \text{Dee} \rightarrow \begin{bmatrix} \text{Nat} \rightarrow 1 \\ \text{Mo} \rightarrow 1 \\ \text{Dee} \rightarrow 0 \end{bmatrix} \\ \text{Mo} \rightarrow \begin{bmatrix} \text{Mo} \rightarrow 0 \\ \text{Dee} \rightarrow 0 \\ \text{Nat} \rightarrow 0 \end{bmatrix} \end{bmatrix} (\quad) \right) (\quad) =$$

# More on composition rules

Question: How do we get to the truth value for the following sentence/tree?

(7) $[\![dislike]\!]^{M'} = \{\langle Nat, Dee\rangle, \langle Mo, Dee\rangle\}$

(8) $\left[\!\!\left[ \text{[S [NP [N Dee]] [VP [V dislikes] [NP [N Mo]]]]} \right]\!\!\right]^{M'} = ([\![dislikes]\!]^{M'}([\![Mo]\!]^{M'}))([\![Dee]\!]^{M'})$

$$= \left( \begin{bmatrix} \text{Nat} \rightarrow \begin{bmatrix} \text{Dee} \rightarrow 0 \\ \text{Mo} \rightarrow 0 \\ \text{Nat} \rightarrow 0 \end{bmatrix} \\ \text{Dee} \rightarrow \begin{bmatrix} \text{Nat} \rightarrow 1 \\ \text{Mo} \rightarrow 1 \\ \text{Dee} \rightarrow 0 \end{bmatrix} \\ \text{Mo} \rightarrow \begin{bmatrix} \text{Mo} \rightarrow 0 \\ \text{Dee} \rightarrow 0 \\ \text{Nat} \rightarrow 0 \end{bmatrix} \end{bmatrix} (Mo) \right) (Dee) =$$

# More on composition rules

Question: How do we get to the truth value for the following sentence/tree?

(7) $[\![dislike]\!]^{M'} = \{\langle Nat, Dee\rangle, \langle Mo, Dee\rangle\}$

(8) $\left[\!\!\left[ \text{[}_S \text{ [}_{NP} \text{ [}_N \text{ Dee]] [}_{VP} \text{ [}_V \text{ dislikes] [}_{NP} \text{ [}_N \text{ Mo]]]]} \right]\!\!\right]^{M'}$ $= ([\![dislikes]\!]^{M'}([\![Mo]\!]^{M'}))([\![Dee]\!]^{M'})$

$$= \left( \begin{bmatrix} \text{Nat} \rightarrow \begin{bmatrix} \text{Dee} \rightarrow 0 \\ \text{Mo} \rightarrow 0 \\ \text{Nat} \rightarrow 0 \end{bmatrix} \\ \text{Dee} \rightarrow \begin{bmatrix} \text{Nat} \rightarrow 1 \\ \text{Mo} \rightarrow 1 \\ \text{Dee} \rightarrow 0 \end{bmatrix} \\ \text{Mo} \rightarrow \begin{bmatrix} \text{Mo} \rightarrow 0 \\ \text{Dee} \rightarrow 0 \\ \text{Nat} \rightarrow 0 \end{bmatrix} \end{bmatrix} (Mo) \right) (Dee) = 0$$

# More on composition rules

(9) $[\![dislike]\!]^{M'} = \{\langle Nat, Dee\rangle, \langle Mo, Dee\rangle\}$

(10) $$[\![\alpha]\!]^{M'} = ([\![dislikes]\!]^{M'}([\![Dee]\!]^{M'}))([\![Mo]\!]^{M'}) = \left(\begin{bmatrix} Nat \rightarrow \begin{bmatrix} Dee \rightarrow 0 \\ Mo \rightarrow 0 \\ Nat \rightarrow 0 \end{bmatrix} \\ Dee \rightarrow \begin{bmatrix} Nat \rightarrow 1 \\ Mo \rightarrow 1 \\ Dee \rightarrow 0 \end{bmatrix} \\ Mo \rightarrow \begin{bmatrix} Mo \rightarrow 0 \\ Dee \rightarrow 0 \\ Nat \rightarrow 0 \end{bmatrix} \end{bmatrix} (Dee)\right)(Mo) = 1$$

For transitive verbs, we can also use the more abstract function notation instead of the table notation.

(11) $$[\![\alpha]\!]^{M'} = ([\![dislikes]\!]^{M'}([\![Dee]\!]^{M'}))([\![Mo]\!]^{M'})$$

$$= \left(\begin{bmatrix} \text{f: } D_M \rightarrow \{g : D_{M'} \rightarrow \{1,0\}\} \\ \text{such that for all } x \in D_M.\ f(x) = 1 \\ \text{such that for all } y \in D_{M'}, g_x(y) = 1 \\ \text{iff } y \text{ dislikes } x \end{bmatrix} (Dee)\right)(Mo) = 1 \text{ iff Mo dislikes Dee}$$

# Semantic types

Our system now makes reference to various sorts of objects:

1. individuals
2. truth values
3. functions from individuals to truth values (one-place predicates)
4. functions from individuals to functions from individuals to truth values (two-place predicates)

It would be nice to have a simpler system for keeping track of what sorts of objects we're referring to. Semantic types give us a systematic (and less wordy!) way of describing how different kinds of linguistic objects fit together in compositional semantics.

(12) Semantic types:

a. Let $\langle e \rangle$ be the type of individuals ($e$ = "entity")

b. Let $\langle t \rangle$ be the type of truth values

c. If $\sigma$ and $\tau$ are semantic types, then $\langle \sigma, \tau \rangle$ is a semantic type

d. nothing else is a semantic type

# Semantic types

(13) Semantic types:

a. Let $\langle e \rangle$ be the type of individuals ($e$ = "entity")

b. Let $\langle t \rangle$ be the type of truth values

c. If $\sigma$ and $\tau$ are semantic types, then $\langle \sigma, \tau \rangle$ is a semantic type

d. nothing else is a semantic type

A type is a kind of implication: $\langle \sigma, \tau \rangle$ says "If you give me a $\sigma$, I will return a $\tau$." And $\langle \sigma, \langle \sigma, \tau \rangle \rangle$ says "If you gave me a $\sigma$, I will return a $\langle \sigma, \tau \rangle$." In other words: $\langle \underbrace{\sigma,}_{in} \underbrace{\langle \sigma, \tau \rangle}_{out} \rangle$

(14) Basic types:

a. $e$ is the type of individuals
$D_e := D$

b. $t$ is the type of truth values
$D_t := \{0,1\}$

(15) (Some) derived types:

a. $\langle e, t \rangle$ is the type for functions from indviduals to truth values
$D_{\langle e,t \rangle} := \{$f: f is a function from $D_e$ to $D_t\}$

b. $\langle e, \langle e, t \rangle \rangle$ is the type for functions from individuals to functions from individuals to truth values
$D_{\langle e,\langle e,t \rangle \rangle} := \{$f: f is a function from $D_e$ to $D_{\langle e,t \rangle}\}$

# Semantic types

(16) Semantic types:

a. Let $\langle e \rangle$ be the type of individuals ($e$ = "entity")

b. Let $\langle t \rangle$ be the type of truth values

c. If $\sigma$ and $\tau$ are semantic types, then $\langle \sigma, \tau \rangle$ is a semantic type

d. nothing else is a semantic type

A type is a kind of implication: $\langle \sigma, \tau \rangle$ says "If you give me a $\sigma$, I will return a $\tau$." And $\langle \sigma, \langle \sigma, \tau \rangle\rangle$ says "If you gave me a $\sigma$, I will return a $\langle \sigma, \tau \rangle$." In other words: $\langle \underbrace{\sigma,}_{in} \underbrace{\langle \sigma, \tau \rangle}_{out} \rangle$

(17) Examples:

a. type $\langle e \rangle$: *Nat, Mo, Spiderman, ...*

b. type $\langle t \rangle$: *1,0*

c. type $\langle e, t \rangle$: sleeps, happy, cried,...

d. type $\langle e, \langle e, t \rangle\rangle$: dislikes, loves, beschimpft, ...

e. type $\langle e, \langle e, \langle e, t \rangle\rangle\rangle$: give, introduce,...

# Semantic types

We can now assign types to the semantic object we have been using:

1. individuals $\langle e \rangle$
2. truth values $\langle t \rangle$
3. functions from individuals to truth values (one-place predicates) $\langle e, t \rangle$
4. functions from individuals to functions from individuals to truth values (two-place predicates) $\langle e, \langle e, t \rangle$

(18) Semantic denotation domains:

a. $D_e = D_M$

b. $D_t = \{1, 0\}$

c. For any semantic type $\sigma$ and $\tau$, $D_{\langle \sigma, \tau \rangle}$ is the set of all functions from $D_\sigma$ to $D_\tau$

# $\lambda$-calculus

The way we wrote down functions so far requires a lot of space. There is a simpler way of identifying functions without writing long definitions or big tables.

Instead of...

(19) $[\![\textit{cries}]\!]^M = f : D_M \rightarrow \{1, 0\}$ such that for all $x \in D_M . f(x) = 1$ iff $x$ cries

... we can write:

(20) $[\![\textit{cries}]\!]^M = \lambda x : x \in D_M$ [$x$ cries]

General form of a function in $\lambda$-notation: $\boxed{\lambda \alpha : \phi[\gamma]}$ In other words:

$\lambda$ argument : domain condition [value description]

- $\alpha$ is the argument of the function. The semantic type of the argument of the function is the same as the type of $\alpha$
- $\phi$ is the *domain condition* on $\alpha$. It is used to specify the domain that $\alpha$ denotes in, and is often suppressed if its identity is implicitly known
- $\gamma$ is the *value description*. It specifies the value that the function assigns to $\alpha$. The semantic type of can be inferred from the nature of its description.

# $\lambda$-calculus

General form of a function in $\lambda$-notation: $\boxed{\lambda\alpha : \phi[\gamma]}$

In other words: $\boxed{\lambda \text{ argument : domain condition [value description]}}$

- $\alpha$ is the argument of the function. The semantic type of the argument of the function is the same as the type of $\alpha$
- $\phi$ is the *domain condition* on $\alpha$. It is used to specify the domain that $\alpha$ denotes in, and is often suppressed if its identity is implicitly known
- $\gamma$ is the *value description*. It specifies the value that the function assigns to $\alpha$. The semantic type of can be inferred from the nature of its description.

There many different versions of using the $\lambda$-notation, here are a few:

(21) Alternative notations for $[\![\textit{cries}]\!]^M = \lambda x : x \in D_M$ [$x$ cries]

a. $[\![\textit{cries}]\!]^M = \lambda x : x \in D_M.x$ cries

b. $[\![\textit{cries}]\!]^M = \lambda x[x$ cries]

c. $[\![\textit{cries}]\!]^M = \lambda x.x$ cries

We still read them as functions:
*the function f from individuals x to truth values such that f(x) = 1 iff x cries, and 0 otherwise*

# $\lambda$-calculus

General form of a function in $\lambda$-notation: $\lambda\alpha : \phi[\gamma]$

In other words: $\lambda$ argument : domain condition [value description]

- $\alpha$ is the argument of the function. The semantic type of the argument of the function is the same as the type of $\alpha$
- $\phi$ is the *domain condition* on $\alpha$. It is used to specify the domain that $\alpha$ denotes in, and is often suppressed if its identity is implicitly known
- $\gamma$ is the *value description*. It specifies the value that the function assigns to $\alpha$. The semantic type of can be inferred from the nature of its description.

We can now simplify the function notation for transitive verbs:

(22) $[\![\text{dislike}]\!]^M = f : D_M \rightarrow \{g : D_M \rightarrow \{1, 0\}\}$ such that for all $x \in D_M, f(x) = g_x : D_M \rightarrow \{1,0\}$ such that for all $y \in D_M, g_x(y) = 1$ iff $y$ dislikes $x$

(23) $[\![\text{dislike}]\!]^M = \lambda x : x \in D_M.\ [\lambda y : y \in D_M.\ y \text{ dislikes } x\ ]$

(24) More notation variants:

a. $[\![\text{dislike}]\!]^M = \lambda x \lambda y . y \text{ dislikes } x$

b. $[\![\text{dislike}]\!]^M = \lambda x \lambda y [y \text{ dislikes } x]$

# $\lambda$-calculus and semantic types

- a $\lambda$ can be interpreted as a slot or place, denoting how many arguments must saturate the predicate to yield a truth-value
- a one-place predicate will have one $\lambda$, a two-place predicate will have two $\lambda$s, a three-place predicate will have three, and so on

(25) $[\![\text{dislike}]\!]^M = \lambda x : x \in D_M . [\lambda y : y \in D_M . \ y \text{ dislikes } x\ ]$

(26) $[\![\textit{cries}]\!]^M = \lambda x : x \in D_M . [x \text{ cries}]$

- so $\lambda x$... can be understood as "complete me by giving me something to substitute for $x$"
- to make clear what kind of argument the function needs, we can write the semantic type as a subscript on the variable
- we know that one-place predicates and two-place predicates want arguments which are individuals, i.e. arguments of type $\langle e \rangle$

(27) $[\![\text{dislike}]\!] = \lambda x : x \in D_{\langle e \rangle} . [\lambda y : y \in D_{\langle e \rangle} . \ y \text{ dislikes } x\ ] = \lambda x_e \lambda y_e [y \text{ dislikes } x\ ]$

(28) $[\![\textit{cries}]\!] = \lambda x : x \in D_{\langle e \rangle} . [x \text{ cries}] = \lambda x_e [x \text{ cries}]$

# $\lambda$-calculus and semantic types

How do we get from types to $\lambda$-notation?

(29) $\underbrace{[\![cries]\!]}_{\langle e,t\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,} \underbrace{[\text{ x cries }]}_{t\rangle}$

(30) $\underbrace{[\![dislike]\!]}_{\langle e,\langle e,t\rangle\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,} . [\underbrace{\lambda y : y \in D_{\langle e\rangle}}_{\langle e,} . \underbrace{\text{y dislikes x}}_{t\rangle\rangle}]$

(31) $\underbrace{[\![introduce]\!]}_{} = \underbrace{\qquad\qquad}_{} . [\underbrace{\qquad\qquad}_{} . [\underbrace{\qquad\qquad}_{} . \underbrace{\qquad\qquad\qquad}_{}]]$

(32) $\underbrace{[\![nicht]\!]}_{} = \underbrace{\qquad\qquad}_{} . [\underbrace{\qquad\qquad}_{} . \underbrace{\qquad}_{}]$ *(weil) Peter nicht weint*

(33) $\underbrace{[\![und]\!]}_{} = \underbrace{\qquad\qquad}_{} . [\underbrace{\qquad\qquad}_{} . [\underbrace{\qquad\qquad}_{} . \underbrace{\qquad\qquad}_{}]]$

*(weil) Peter lacht und weint*

# $\lambda$-calculus and semantic types

How do we get from types to $\lambda$-notation?

(29) $\underbrace{[\![\textit{cries}]\!]}_{\langle e,t\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,} \underbrace{[\text{ x cries }]}_{t\rangle}$

(30) $\underbrace{[\![\textit{dislike}]\!]}_{\langle e,\langle e,t\rangle\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,}. [\underbrace{\lambda y : y \in D_{\langle e\rangle}}_{\langle e,}. \underbrace{\text{y dislikes x}}_{t\rangle\rangle}]$

(31) $\underbrace{[\![\textit{introduce}]\!]}_{} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{}. [\underbrace{\lambda y : y \in D_{\langle e\rangle}}_{}. [\underbrace{\lambda z : z \in D_{\langle e\rangle}}_{}. \underbrace{\text{z introduces x to y}}_{}]]$

(32) $\underbrace{[\![\textit{nicht}]\!]}_{} = \underbrace{\qquad\qquad}_{}. [\underbrace{\qquad\qquad}_{}. \underbrace{\qquad}_{}]$ *(weil) Peter nicht weint*

(33) $\underbrace{[\![\textit{und}]\!]}_{} = \underbrace{\qquad\qquad}_{}. [\underbrace{\qquad\qquad}_{}. [\underbrace{\qquad\qquad}_{}. \underbrace{\qquad\qquad}_{}]]$

*(weil) Peter lacht und weint*

# $\lambda$-calculus and semantic types

How do we get from types to $\lambda$-notation?

(29) $\underbrace{[\![ cries ]\!]}_{\langle e,t \rangle} = \underbrace{\lambda x : x \in D_{\langle e \rangle}}_{\langle e,} \underbrace{[\text{ x cries }]}_{t \rangle}$

(30) $\underbrace{[\![ dislike ]\!]}_{\langle e,\langle e,t \rangle\rangle} = \underbrace{\lambda x : x \in D_{\langle e \rangle}}_{\langle e,} . [\underbrace{\lambda y : y \in D_{\langle e \rangle}}_{\langle e,} . \underbrace{\text{y dislikes x}}_{t \rangle\rangle}]$

(31) $\underbrace{[\![ introduce ]\!]}_{\langle e,\langle e,\langle e,t \rangle\rangle\rangle} = \underbrace{\lambda x : x \in D_{\langle e \rangle}}_{\langle e,} . [\underbrace{\lambda y : y \in D_{\langle e \rangle}}_{\langle e,} . [\underbrace{\lambda z : z \in D_{\langle e \rangle}}_{\langle e,} . \underbrace{\text{z introduces x to y}}_{t \rangle\rangle\rangle}]]$

(32) $\underbrace{[\![ nicht ]\!]} = \underbrace{\qquad\qquad\qquad} . [\underbrace{\qquad\qquad\qquad} . \underbrace{\qquad\quad}]$ *(weil) Peter nicht weint*

(33) $\underbrace{[\![ und ]\!]} = \underbrace{\qquad\qquad\qquad} . [\underbrace{\qquad\qquad\qquad} . [\underbrace{\qquad\qquad\qquad} . \underbrace{\qquad\qquad\qquad}]]$

*(weil) Peter lacht und weint*

# $\lambda$-calculus and semantic types

How do we get from types to $\lambda$-notation?

(29) $\underbrace{[\![cries]\!]}_{\langle e,t\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,} \underbrace{[\text{ x cries }]}_{t\rangle}$

(30) $\underbrace{[\![dislike]\!]}_{\langle e,\langle e,t\rangle\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,}. [\underbrace{\lambda y : y \in D_{\langle e\rangle}}_{\langle e,}. \underbrace{\text{y dislikes x}}_{t\rangle\rangle}]$

(31) $\underbrace{[\![introduce]\!]}_{\langle e,\langle e,\langle e,t\rangle\rangle\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,}. [\underbrace{\lambda y : y \in D_{\langle e\rangle}}_{\langle e,}. [\underbrace{\lambda z : z \in D_{\langle e\rangle}}_{\langle e,}. \underbrace{\text{z introduces x to y}}_{t\rangle\rangle\rangle}]]$

(32) $\underbrace{[\![nicht]\!]} = \underbrace{\lambda P : P \in D_{\langle e,t\rangle}}. [\underbrace{\lambda x : x \in D_{\langle e\rangle}}. \underbrace{\neg P(x)}]$ *(weil) Peter nicht weint*

(33) $\underbrace{[\![und]\!]} = \underbrace{\qquad\qquad}. [\underbrace{\qquad\qquad}. [\underbrace{\qquad\qquad}. \underbrace{\qquad\qquad}]]$

*(weil) Peter lacht und weint*

# $\lambda$-calculus and semantic types

How do we get from types to $\lambda$-notation?

(29) $\underbrace{[\![cries]\!]}_{\langle e,t\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,} \underbrace{[\text{ x cries }]}_{t\rangle}$

(30) $\underbrace{[\![dislike]\!]}_{\langle e,\langle e,t\rangle\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,}. [\underbrace{\lambda y : y \in D_{\langle e\rangle}}_{\langle e,}. \underbrace{\text{y dislikes x}}_{t\rangle\rangle}]$

(31) $\underbrace{[\![introduce]\!]}_{\langle e,\langle e,\langle e,t\rangle\rangle\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,}. [\underbrace{\lambda y : y \in D_{\langle e\rangle}}_{\langle e,}. [\underbrace{\lambda z : z \in D_{\langle e\rangle}}_{\langle e,}. \underbrace{\text{z introduces x to y}}_{t\rangle\rangle\rangle}]]$

(32) $\underbrace{[\![nicht]\!]}_{\langle\langle e,t\rangle,\langle e,t\rangle\rangle} = \underbrace{\lambda P : P \in D_{\langle e,t\rangle}}_{\langle\langle e,t\rangle,}. [\underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,}. \underbrace{\neg P(x)}_{t\rangle\rangle}]$ *(weil) Peter nicht weint*

(33) $\underbrace{[\![und]\!]}_{} = \underbrace{\qquad\qquad}_{}. [\underbrace{\qquad\qquad}_{}. [\underbrace{\qquad\qquad}_{}. \underbrace{\qquad\qquad}_{}]]$

*(weil) Peter lacht und weint*

# $\lambda$-calculus and semantic types

How do we get from types to $\lambda$-notation?

(29) $\underbrace{[\![cries]\!]}_{\langle e,t\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,} \underbrace{[\text{ x cries }]}_{t\rangle}$

(30) $\underbrace{[\![dislike]\!]}_{\langle e,\langle e,t\rangle\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,}. [\underbrace{\lambda y : y \in D_{\langle e\rangle}}_{\langle e,}. \underbrace{\text{y dislikes x}}_{t\rangle\rangle}]$

(31) $\underbrace{[\![introduce]\!]}_{\langle e,\langle e,\langle e,t\rangle\rangle\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,}. [\underbrace{\lambda y : y \in D_{\langle e\rangle}}_{\langle e,}. [\underbrace{\lambda z : z \in D_{\langle e\rangle}}_{\langle e,}. \underbrace{\text{z introduces x to y}}_{t\rangle\rangle\rangle}]]$

(32) $\underbrace{[\![nicht]\!]}_{\langle\langle e,t\rangle,\langle e,t\rangle\rangle} = \underbrace{\lambda P : P \in D_{\langle e,t\rangle}}_{\langle\langle e,t\rangle,}. [\underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,}. \underbrace{\neg P(x)}_{t\rangle\rangle}]$ *(weil) Peter nicht weint*

(33) $\underbrace{[\![und]\!]} = \underbrace{\lambda P : P \in D_{\langle e,t\rangle}}. [\underbrace{\lambda Q : Q \in D_{\langle e,t\rangle}}. [\underbrace{\lambda x : x \in D_{\langle e\rangle}}. \underbrace{P(x) \wedge Q(x)}]]$

*(weil) Peter lacht und weint*

# $\lambda$-calculus and semantic types

How do we get from types to $\lambda$-notation?

(29) $\underbrace{[\![cries]\!]}_{\langle e,t\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,} \underbrace{[\text{ x cries }]}_{t\rangle}$

(30) $\underbrace{[\![dislike]\!]}_{\langle e,\langle e,t\rangle\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,}. [\underbrace{\lambda y : y \in D_{\langle e\rangle}}_{\langle e,}. \underbrace{\text{y dislikes x}}_{t\rangle\rangle}]$

(31) $\underbrace{[\![introduce]\!]}_{\langle e,\langle e,\langle e,t\rangle\rangle\rangle} = \underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,}. [\underbrace{\lambda y : y \in D_{\langle e\rangle}}_{\langle e,}. [\underbrace{\lambda z : z \in D_{\langle e\rangle}}_{\langle e,}. \underbrace{\text{z introduces x to y}}_{t\rangle\rangle\rangle}]]$

(32) $\underbrace{[\![nicht]\!]}_{\langle\langle e,t\rangle,\langle e,t\rangle\rangle} = \underbrace{\lambda P : P \in D_{\langle e,t\rangle}}_{\langle\langle e,t\rangle,}. [\underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,}. \underbrace{\neg P(x)}_{t\rangle\rangle}]$ *(weil) Peter nicht weint*

(33) $\underbrace{[\![und]\!]}_{\langle\langle e,t\rangle,\langle\langle e,t\rangle,\langle e,t\rangle\rangle\rangle} = \underbrace{\lambda P : P \in D_{\langle e,t\rangle}}_{\langle\langle e,t\rangle,}. [\underbrace{\lambda Q : Q \in D_{\langle e,t\rangle}}_{\langle\langle e,t\rangle,}. [\underbrace{\lambda x : x \in D_{\langle e\rangle}}_{\langle e,}. \underbrace{P(x) \wedge Q(x)}_{t\rangle\rangle\rangle}]]$

*(weil) Peter lacht und weint*

# $\lambda$-conversion

- what we have just introduced is also called $\lambda$-*abstraction*: the operation of getting from a description which contains a variable to a lambda term
- in order to get to the value of a function, we apply the $\lambda$-term to an argument by substituting the $\lambda$-term with that argument, we call this operation $\lambda$-*conversion*
- thus, functional application leads to $\lambda$-*conversion*

(34) $[_S$ Mo cries$]$

(35)
a. $[\![cries]\!] = \lambda x_{\langle e \rangle}[\text{cries}(x)]$ *Lexicon*
b. $[\![Mo]\!] = \text{Mo}$ *Lexicon*
c. $[\![cries]\!]([\![Mo]\!]) = \lambda x_{\langle e \rangle}[\text{cries}(x)](\text{Mo})$ *Functional Application*
d. $[\![S]\!] = \text{cries}(\text{Mo}) = 1$ iff Mo cries $\lambda$-*conversion*

Steps for $\lambda$-*conversion*:

1. identify the leftmost $\lambda$ and the variable it introduces
2. go through the formula and replace every instance of that variable with the argument the $\lambda$-term applies to
3. delete the leftmost $\lambda$ with its variable

# $\lambda$-conversion

- the number of $\lambda$s in our formula can tell us the valency of the predicate we are dealing with
- the order of $\lambda$s tells us the order in which the function application occurs: we saturate $\lambda$s from outermost to innermost

(36) $[_S$ Dee $[_{VP}$ loves Jean$]]$

(37) a.

b.

c.

d.

e.

f.

g.

# $\lambda$-conversion

- the number of $\lambda$s in our formula can tell us the valency of the predicate we are dealing with
- the order of $\lambda$s tells us the order in which the function application occurs: we saturate $\lambda$s from outermost to innermost

(36) $[_S \text{ Dee } [_{VP} \text{ loves Jean}]]$

(37) a. $[\![ \textit{loves} ]\!] = \lambda y_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{loves}(x, y)]$ *Lexicon*

b.

c.

d.

e.

f.

g.

# $\lambda$-conversion

- the number of $\lambda$s in our formula can tell us the valency of the predicate we are dealing with
- the order of $\lambda$s tells us the order in which the function application occurs: we saturate $\lambda$s from outermost to innermost

(36) $[_S$ Dee $[_{VP}$ loves Jean$]]$

(37)
a. $[\![\textit{loves}]\!] = \lambda y_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{loves}(x, y)]$ *Lexicon*
b. $[\![\textit{Jean}]\!] =$ Jean *Lexicon*
c.
d.
e.
f.
g.

# $\lambda$-conversion

- the number of $\lambda$s in our formula can tell us the valency of the predicate we are dealing with
- the order of $\lambda$s tells us the order in which the function application occurs: we saturate $\lambda$s from outermost to innermost

(36) $[_S$ Dee $[_{VP}$ loves Jean$]]$

(37)
a. $[\![\textit{loves}]\!] = \lambda y_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{loves}(x, y)]$ *Lexicon*
b. $[\![\textit{Jean}]\!] = \text{Jean}$ *Lexicon*
c. $[\![\textit{loves}]\!]([\![\textit{Jean}]\!]) = \lambda y_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{loves}(x, y)](\text{Jean})$ *Functional Application*
d.
e.
f.
g.

# $\lambda$-conversion

- the number of $\lambda$s in our formula can tell us the valency of the predicate we are dealing with
- the order of $\lambda$s tells us the order in which the function application occurs: we saturate $\lambda$s from outermost to innermost

(36) $[_S$ Dee $[_{VP}$ loves Jean$]]$

(37) a. $[\![\textit{loves}]\!] = \lambda y_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{loves}(x, y)]$ *Lexicon*

b. $[\![\textit{Jean}]\!] = \text{Jean}$ *Lexicon*

c. $[\![\textit{loves}]\!]([\![\textit{Jean}]\!]) = \lambda y_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{loves}(x, y)](\text{Jean})$ *Functional Application*

d. $[\![VP]\!] = \lambda x_{\langle e \rangle}[\text{loves}(x, \text{Jean})]$ *$\lambda$-conversion*

e.

f.

g.

# $\lambda$-conversion

- the number of $\lambda$s in our formula can tell us the valency of the predicate we are dealing with
- the order of $\lambda$s tells us the order in which the function application occurs: we saturate $\lambda$s from outermost to innermost

(36) $[_S$ Dee $[_{VP}$ loves Jean$]]$

(37)
a. $[\![loves]\!] = \lambda y_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{loves}(x, y)]$ *Lexicon*

b. $[\![Jean]\!] = \text{Jean}$ *Lexicon*

c. $[\![loves]\!]([\![Jean]\!]) = \lambda y_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{loves}(x, y)](\text{Jean})$ *Functional Application*

d. $[\![VP]\!] = \lambda x_{\langle e \rangle}[\text{loves}(x,\text{Jean})]$ *$\lambda$-conversion*

e. $[\![Dee]\!] = \text{Dee}$ *Lexicon*

f.

g.

# $\lambda$-conversion

- the number of $\lambda$s in our formula can tell us the valency of the predicate we are dealing with
- the order of $\lambda$s tells us the order in which the function application occurs: we saturate $\lambda$s from outermost to innermost

(36) $[_S$ Dee $[_{VP}$ loves Jean$]]$

(37) a. $[\![loves]\!] = \lambda y_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{loves}(x, y)]$ *Lexicon*

b. $[\![Jean]\!] = \text{Jean}$ *Lexicon*

c. $[\![loves]\!]([\![Jean]\!]) = \lambda y_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{loves}(x, y)](\text{Jean})$ *Functional Application*

d. $[\![VP]\!] = \lambda x_{\langle e \rangle}[\text{loves}(x, \text{Jean})]$ *$\lambda$-conversion*

e. $[\![Dee]\!] = \text{Dee}$ *Lexicon*

f. $[\![VP]\!]([\![Dee]\!]) = \lambda x_{\langle e \rangle}[\text{loves}(x, \text{Jean})](\text{Dee})$ *Functional Application*

g.

# $\lambda$-conversion

- the number of $\lambda$s in our formula can tell us the valency of the predicate we are dealing with
- the order of $\lambda$s tells us the order in which the function application occurs: we saturate $\lambda$s from outermost to innermost

(36) $[_S$ Dee $[_{VP}$ loves Jean$]]$

(37)
a. $[\![ \textit{loves} ]\!] = \lambda y_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{loves}(x, y)]$ — *Lexicon*
b. $[\![ \textit{Jean} ]\!] = \text{Jean}$ — *Lexicon*
c. $[\![ \textit{loves} ]\!]([\![ \textit{Jean} ]\!]) = \lambda y_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{loves}(x, y)](\text{Jean})$ — *Functional Application*
d. $[\![ \textit{VP} ]\!] = \lambda x_{\langle e \rangle} [\text{loves}(x, \text{Jean})]$ — *$\lambda$-conversion*
e. $[\![ \textit{Dee} ]\!] = \text{Dee}$ — *Lexicon*
f. $[\![ \textit{VP} ]\!]([\![ \textit{Dee} ]\!]) = \lambda x_{\langle e \rangle} [\text{loves}(x, \text{Jean})](\text{Dee})$ — *Functional Application*
g. $[\![ \textit{S} ]\!] = \text{loves}(\text{Dee}, \text{Jean}) = 1$ iff Dee loves Jean — *$\lambda$-conversion*

# More predicates: ditransitives

Provide the $\lambda$-notation for $[\![$*show*$]\!]$!

(38) $[_S$ Jean $[_{VP}$ shows Spiderman to Mo$]]$

# More predicates: ditransitives

Provide the $\lambda$-notation for $[\![show]\!]$!

(38) $[_S$ Jean $[_{VP}$ shows Spiderman to Mo]]

(39) $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)] = 1$ iff x shows y to z

# More predicates: ditransitives

Provide the $\lambda$-notation for $[\![show]\!]$!

(38) $[_S$ Jean $[_{VP}$ shows Spiderman to Mo]]

(39) $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{show}(x, y, z)] = 1$ iff x shows y to z

Calculate the denotation of S above!

# More predicates: ditransitives

Provide the $\lambda$-notation for $[\![show]\!]$!

(38) $[_S$ Jean $[_{VP}$ shows Spiderman to Mo]]

(39) $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)] = 1$ iff x shows y to z

Calculate the denotation of S above!

(40) a. $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)]$ *Lex*

# More predicates: ditransitives

Provide the $\lambda$-notation for $[\![show]\!]$!

(38) $[_S$ Jean $[_{VP}$ shows Spiderman to Mo$]]$

(39) $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)] = 1$ iff x shows y to z

Calculate the denotation of S above!

(40) a. $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)]$ *Lex*

b. $[\![Spiderman]\!]$ = Spiderman *Lex*

# More predicates: ditransitives

Provide the $\lambda$-notation for $[\![ show ]\!]$!

(38) $[_S$ Jean $[_{VP}$ shows Spiderman to Mo$]]$

(39) $[\![ show ]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{show}(x, y, z)] = 1$ iff x shows y to z

Calculate the denotation of S above!

(40) a. $[\![ show ]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{show}(x, y, z)]$ *Lex*

b. $[\![ Spiderman ]\!] = \text{Spiderman}$ *Lex*

c. $[\![ shows ]\!]([\![ Spiderman ]\!]) = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{show}(x, y, z)](\text{Spiderman})$ *FA*

# More predicates: ditransitives

Provide the $\lambda$-notation for $[\![show]\!]$!

(38) $[_S$ Jean $[_{VP}$ shows Spiderman to Mo$]]$

(39) $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{show}(x, y, z)]$ = 1 iff x shows y to z

Calculate the denotation of S above!

(40)
a. $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{show}(x, y, z)]$ *Lex*
b. $[\![Spiderman]\!]$ = Spiderman *Lex*
c. $[\![shows]\!]([\![Spiderman]\!]) = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{show}(x, y, z)](\text{Spiderman})$ *FA*
d. $[\![shows\ Spiderman]\!] = \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{show}(x, \text{Spiderman}, z)]$ *$\lambda$-C*

# More predicates: ditransitives

Provide the $\lambda$-notation for $[\![show]\!]$!

(38) $[_S$ Jean $[_{VP}$ shows Spiderman to Mo$]]$

(39) $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)] = 1$ iff x shows y to z

Calculate the denotation of S above!

(40)
a. $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)]$ *Lex*
b. $[\![Spiderman]\!] = \text{Spiderman}$ *Lex*
c. $[\![shows]\!]([\![Spiderman]\!]) = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)](\text{Spiderman})$ *FA*
d. $[\![shows\ Spiderman]\!] = \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, \text{Spiderman}, z)]$ *$\lambda$-C*
e. $[\![Mo]\!] = \text{Mo}$; $[\![to]\!] = \lambda y_{\langle e \rangle}[y]$ *Lex*

# More predicates: ditransitives

Provide the $\lambda$-notation for $[\![show]\!]$!

(38) $[_S$ Jean $[_{VP}$ shows Spiderman to Mo]]

(39) $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)] = 1$ iff x shows y to z

Calculate the denotation of S above!

(40)
a. $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)]$ *Lex*
b. $[\![Spiderman]\!] = \text{Spiderman}$ *Lex*
c. $[\![shows]\!]([\![Spiderman]\!]) = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)](\text{Spiderman})$ *FA*
d. $[\![shows\ Spiderman]\!] = \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, \text{Spiderman}, z)]$ *$\lambda$-C*
e. $[\![Mo]\!] = \text{Mo}$; $[\![to]\!] = \lambda y_{\langle e \rangle}[y]$ *Lex*
f. $[\![to]\!]([\![Mo]\!]) = \lambda y_{\langle e \rangle}[y](Mo) = Mo$ *FA, $\lambda$-C*

# More predicates: ditransitives

Provide the $\lambda$-notation for $[\![show]\!]$!

(38) $[_S$ Jean $[_{VP}$ shows Spiderman to Mo$]]$

(39) $[\![show]\!] = \lambda y_{\langle e\rangle}\lambda z_{\langle e\rangle}\lambda x_{\langle e\rangle}[\text{show}(x, y, z)] = 1$ iff x shows y to z

Calculate the denotation of S above!

(40)
a. $[\![show]\!] = \lambda y_{\langle e\rangle}\lambda z_{\langle e\rangle}\lambda x_{\langle e\rangle}[\text{show}(x, y, z)]$ *Lex*
b. $[\![Spiderman]\!] = \text{Spiderman}$ *Lex*
c. $[\![shows]\!]([\![Spiderman]\!]) = \lambda y_{\langle e\rangle}\lambda z_{\langle e\rangle}\lambda x_{\langle e\rangle}[\text{show}(x, y, z)](\text{Spiderman})$ *FA*
d. $[\![shows\ Spiderman]\!] = \lambda z_{\langle e\rangle}\lambda x_{\langle e\rangle}[\text{show}(x,\text{Spiderman}, z)]$ *$\lambda$-C*
e. $[\![Mo]\!] = \text{Mo}$; $[\![to]\!] = \lambda y_{\langle e\rangle}[y]$ *Lex*
f. $[\![to]\!]([\![Mo]\!]) = \lambda y_{\langle e\rangle}[y](Mo) = Mo$ *FA, $\lambda$-C*
g. $[\![shows\ Spiderman]\!]([\![to\ Mo]\!]) = \lambda z_{\langle e\rangle}\lambda x_{\langle e\rangle}[\text{show}(x,\text{Spiderman}, z)](\text{Mo})$ *FA*

# More predicates: ditransitives

Provide the $\lambda$-notation for $[\![show]\!]$!

(38) $[_S$ Jean $[_{VP}$ shows Spiderman to Mo$]]$

(39) $[\![show]\!] = \lambda y_{\langle e\rangle}\lambda z_{\langle e\rangle}\lambda x_{\langle e\rangle}[\text{show}(x, y, z)] = 1$ iff x shows y to z

Calculate the denotation of S above!

(40)
a. $[\![show]\!] = \lambda y_{\langle e\rangle}\lambda z_{\langle e\rangle}\lambda x_{\langle e\rangle}[\text{show}(x, y, z)]$ *Lex*

b. $[\![Spiderman]\!] = \text{Spiderman}$ *Lex*

c. $[\![shows]\!]([\![Spiderman]\!]) = \lambda y_{\langle e\rangle}\lambda z_{\langle e\rangle}\lambda x_{\langle e\rangle}[\text{show}(x, y, z)](\text{Spiderman})$ *FA*

d. $[\![shows\ Spiderman]\!] = \lambda z_{\langle e\rangle}\lambda x_{\langle e\rangle}[\text{show}(x,\text{Spiderman}, z)]$ *$\lambda$-C*

e. $[\![Mo]\!] = \text{Mo}$; $[\![to]\!] = \lambda y_{\langle e\rangle}[y]$ *Lex*

f. $[\![to]\!]([\![Mo]\!]) = \lambda y_{\langle e\rangle}[y](Mo) = Mo$ *FA, $\lambda$-C*

g. $[\![shows\ Spiderman]\!]([\![to\ Mo]\!]) = \lambda z_{\langle e\rangle}\lambda x_{\langle e\rangle}[\text{show}(x,\text{Spiderman}, z)](\text{Mo})$ *FA*

h. $[\![shows\ Spiderman]\!]([\![to\ Mo]\!]) = \lambda x_{\langle e\rangle}[\text{show}(x,\text{Spiderman},\text{Mo})]$ *$\lambda$-C*

# More predicates: ditransitives

Provide the $\lambda$-notation for $[\![show]\!]$!

(38) $[_S$ Jean $[_{VP}$ shows Spiderman to Mo$]]$

(39) $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{show}(x, y, z)] = 1$ iff x shows y to z

Calculate the denotation of S above!

(40)

a. $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{show}(x, y, z)]$ *Lex*

b. $[\![Spiderman]\!] = \text{Spiderman}$ *Lex*

c. $[\![shows]\!]([\![Spiderman]\!]) = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{show}(x, y, z)](\text{Spiderman})$ *FA*

d. $[\![shows\ Spiderman]\!] = \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{show}(x, \text{Spiderman}, z)]$ *$\lambda$-C*

e. $[\![Mo]\!] = \text{Mo}$; $[\![to]\!] = \lambda y_{\langle e \rangle} [y]$ *Lex*

f. $[\![to]\!]([\![Mo]\!]) = \lambda y_{\langle e \rangle} [y](Mo) = Mo$ *FA, $\lambda$-C*

g. $[\![shows\ Spiderman]\!]([\![to\ Mo]\!]) = \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle} [\text{show}(x, \text{Spiderman}, z)](\text{Mo})$ *FA*

h. $[\![shows\ Spiderman]\!]([\![to\ Mo]\!]) = \lambda x_{\langle e \rangle} [\text{show}(x, \text{Spiderman}, \text{Mo})]$ *$\lambda$-C*

i. $[\![Jean]\!] = \text{Jean}$ *Lex*

# More predicates: ditransitives

Provide the $\lambda$-notation for $[\![show]\!]$!

(38) $[_S$ Jean $[_{VP}$ shows Spiderman to Mo]]

(39) $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)] = 1$ iff x shows y to z

Calculate the denotation of S above!

(40)
a. $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)]$ *Lex*
b. $[\![Spiderman]\!] = \text{Spiderman}$ *Lex*
c. $[\![shows]\!]([\![Spiderman]\!]) = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)](\text{Spiderman})$ *FA*
d. $[\![shows\ Spiderman]\!] = \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, \text{Spiderman}, z)]$ *$\lambda$-C*
e. $[\![Mo]\!] = \text{Mo}$; $[\![to]\!] = \lambda y_{\langle e \rangle}[y]$ *Lex*
f. $[\![to]\!]([\![Mo]\!]) = \lambda y_{\langle e \rangle}[y](Mo) = Mo$ *FA, $\lambda$-C*
g. $[\![shows\ Spiderman]\!]([\![to\ Mo]\!]) = \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, \text{Spiderman}, z)](\text{Mo})$ *FA*
h. $[\![shows\ Spiderman]\!]([\![to\ Mo]\!]) = \lambda x_{\langle e \rangle}[\text{show}(x, \text{Spiderman}, \text{Mo})]$ *$\lambda$-C*
i. $[\![Jean]\!] = \text{Jean}$ *Lex*
j. $[\![VP]\!]([\![Jean]\!]) = \lambda x_{\langle e \rangle}[\text{show}(x, \text{Spiderman}, \text{Mo})](\text{Jean})$ *FA*

# More predicates: ditransitives

Provide the $\lambda$-notation for $[\![show]\!]$!

(38) $[_S$ Jean $[_{VP}$ shows Spiderman to Mo$]]$

(39) $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)] = 1$ iff x shows y to z

Calculate the denotation of S above!

(40)
a. $[\![show]\!] = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)]$ — *Lex*

b. $[\![Spiderman]\!]$ = Spiderman — *Lex*

c. $[\![shows]\!]([\![Spiderman]\!]) = \lambda y_{\langle e \rangle} \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, y, z)](\text{Spiderman})$ — *FA*

d. $[\![shows\ Spiderman]\!] = \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, \text{Spiderman}, z)]$ — *$\lambda$-C*

e. $[\![Mo]\!]$ = Mo; $[\![to]\!] = \lambda y_{\langle e \rangle}[y]$ — *Lex*

f. $[\![to]\!]([\![Mo]\!]) = \lambda y_{\langle e \rangle}[y](Mo) = Mo$ — *FA, $\lambda$-C*

g. $[\![shows\ Spiderman]\!]([\![to\ Mo]\!]) = \lambda z_{\langle e \rangle} \lambda x_{\langle e \rangle}[\text{show}(x, \text{Spiderman}, z)](\text{Mo})$ — *FA*

h. $[\![shows\ Spiderman]\!]([\![to\ Mo]\!]) = \lambda x_{\langle e \rangle}[\text{show}(x, \text{Spiderman}, \text{Mo})]$ — *$\lambda$-C*

i. $[\![Jean]\!]$ = Jean — *Lex*

j. $[\![VP]\!]([\![Jean]\!]) = \lambda x_{\langle e \rangle}[\text{show}(x, \text{Spiderman}, \text{Mo})](\text{Jean})$ — *FA*

k. $[\![S]\!]$ = show(Jean,Spiderman,Mo) = 1 iff Jean shows Spiderman to Mo — *$\lambda$-C*

# Identity functions

- the denotation of the preposition *to* is an **identity function**: a function which takes an argument as an input and gives back the same argument as an output

(41) $\underbrace{[\![to]\!]}_{\langle e,e\rangle} = \underbrace{\lambda y_{\langle e\rangle}}_{\langle e,} \underbrace{[y]}_{e\rangle}$ *a function from individuals to individuals*

- identity functions are quite useful for lexical material which does not make a semantic contribution but is there for other (syntactic) reasons
- another candidate for the identity function is the copula *be*

(42) $\underbrace{[\![Mo]\!]}_{\langle e\rangle} \underbrace{[\![is]\!]}_{\langle ??\rangle} \underbrace{[\![happy]\!]}_{\langle e,t\rangle}$

(43) $\underbrace{[\![be]\!]}_{\langle\langle e,t\rangle,\langle e,t\rangle\rangle} = \underbrace{\lambda P_{\langle e,t\rangle}}_{\langle\langle e,t\rangle,} \underbrace{[P]}_{\langle e,t\rangle\rangle}$ *a function from properties to properties*

- another strategy would be that we allow for some phrasal nodes to not get a denotation, they would simply get ignored by $[\![\ ]\!]$
- Why do we think that identity functions are more promising?

# Principle of Interpretability

Why do we think that identity functions are more promising?

1. the no-meaning strategy is in conflict with *compositionality* since the meaning of a complex expression would not be composed of the meanings of all its parts
2. the no-meaning strategy would give us a hard time deriving uninterpretable structures since every time two daughters cannot be combined via functional application, we could simply say that the mother does not have to have a denotation.
   An example:

   (44) *Ann laughed John

   *laugh* is a one-place predicate, i.e. type $\langle e, t\rangle$, it can combine with *John* but *Ann* and *laughed John* cannot combine, resulting in a **type clash**
   $\rightarrow$ this type clash causes ungrammaticality

H&K thus make this semantic filter explicit by...

(45) **Principle of Interpretability**
All nodes in a phrase structure tree must be in the domain of the interpretation function $[\![\ ]\!]$.