Semantic Theory

Week 3: Typed Lambda Calculus

Noortje Venhuizen & Harm Brouwer — Universität des Saarlandes — Summer 2022

Principle of compositionality

> "The meaning of a complex expression is a function of the meanings of its parts and of the syntactic rules by which they are combined"
>
> (Barbara Partee, 1993)

Compositional semantic construction:

1. Define meaning representations for sub-expressions
2. Combine them in a principled manner to obtain a meaning representation for a complex expression.

$$[\![A]\!] = \mathbf{fa}([\![B]\!], [\![C]\!])$$

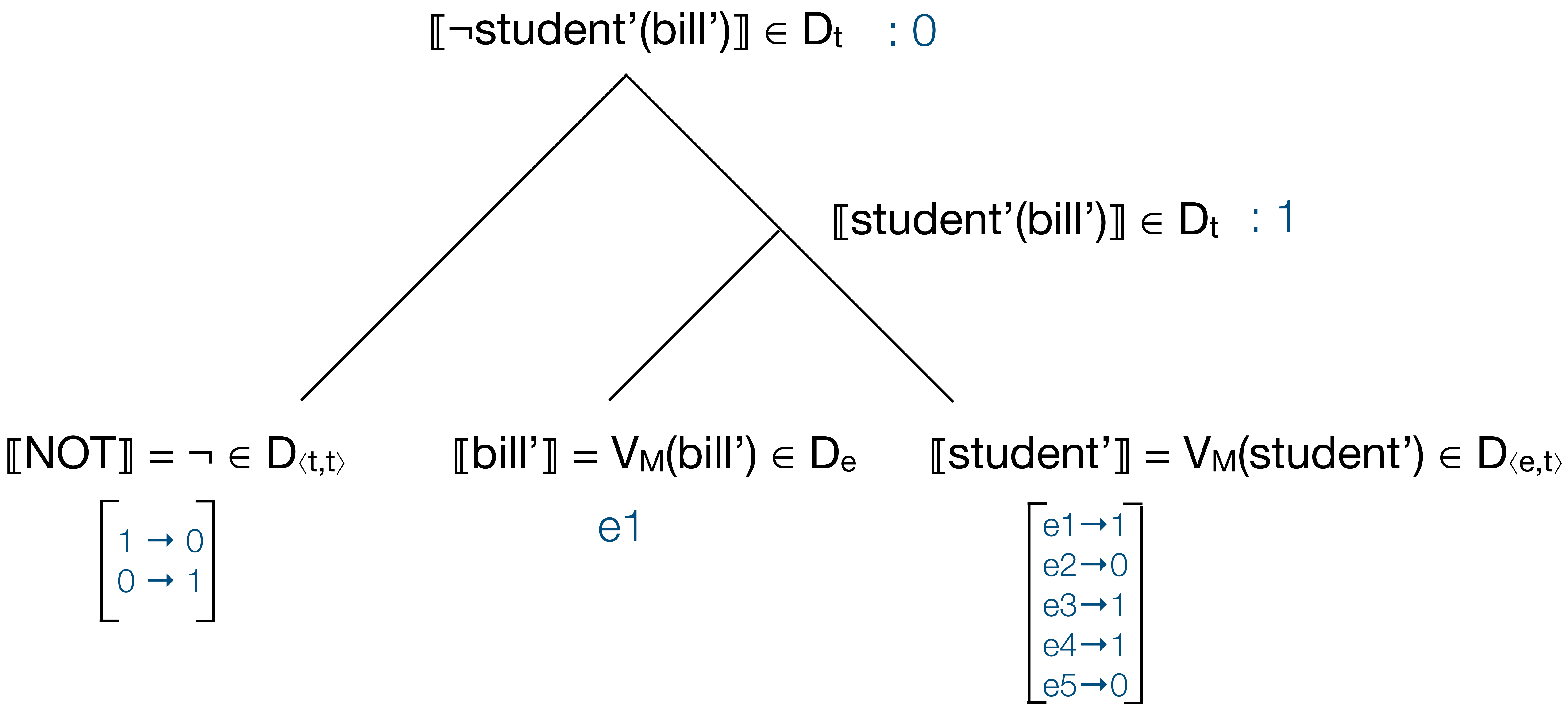
$$[\![B]\!] \qquad [\![C]\!] = \mathbf{fa}([\![D]\!], [\![E]\!])$$

$$[\![D]\!] \qquad [\![E]\!]$$

Compositionality: First try

Types and denotations

Bill is not a student => [NOT [[bill]$_{NP}$ [student]$_{VP}$]$_{S}$]$_{S}$



Compositionality: First try

Explicating Functions and Arguments

Bill is not a student => $[\text{NOT } [[\text{bill}]_{NP} \ [\text{student}]_{VP}]_S \ ]_S$

$[\![\lambda S.\neg S \ (\text{student}'(\text{bill}'))]\!] = [\![\neg \text{student}'(\text{bill}')]\!] \in D_t$

: 0

$[\![\lambda x.\text{student}'(x) \ (\text{bill}')]\!] = [\![\text{student}'(\text{bill}')]\!] \in D_t$

: 1

$[\![\text{NOT}]\!] = [\![\lambda S.\neg S]\!] \in D_{\langle t,t \rangle}$

$\begin{bmatrix} 1 \rightarrow 0 \\ 0 \rightarrow 1 \end{bmatrix}$

$[\![\text{bill}']\!] = V_M(\text{bill}') \in D_e$

e1

$[\![\text{student}']\!] = [\![\lambda x.\text{student}'(x)]\!] \in D_{\langle e,t \rangle}$

$\begin{bmatrix} e1 \rightarrow 1 \\ e2 \rightarrow 0 \\ e3 \rightarrow 1 \\ e4 \rightarrow 1 \\ e5 \rightarrow 0 \end{bmatrix}$

I'm looking for an entity … … it goes here

UNIVERSITÄT DES SAARLANDES

Lambda expressions

Expressiveness of Functions and Arguments

- Lambda expressions are functions that consist of a set of lambda variables and a body
- The body of Lambda expressions can contain logical operators

$[\text{Mary}_e\ [\text{sings and dances}]_{\langle e,t\rangle}]$ $[\![\lambda x(\text{sing}'(x) \wedge \text{dance}'(x))(\text{mary}')]\!] \in D_t$

- Lambda expressions can themselves serve as arguments for functions

$[\ [\text{Not smoking}_{\langle e,t\rangle}]\ [\text{is healthy}]_{\langle\langle e,t\rangle,t\rangle}]$ $[\![\text{healthy}'(\lambda y.\neg(\text{smoking}(y)))]\!] \in D_t$

Lambda abstraction

Formal definition

Winter: EFS Ch3
Page 65

If α is in WE_σ, and x is in VAR_π then λx(α) is in $WE_{\langle\pi, \sigma\rangle}$

λ-abstraction is the operation that transforms expressions of any type σ into a function ⟨π,σ⟩, where π is the type of the λ-variable.

- The scope of the λ-operator is the smallest WE to its right. Wider scope must be indicated by brackets.
- We often use the "dot notation" λx.ϕ indicating that the λ-operator takes wide scope over ϕ.

Interpretation of Lambda-expressions

If $\alpha \in WE_\sigma$ and $v \in VAR_\pi$, then $[\![\lambda v\alpha]\!]^{M,g}$ is that function $f : D_\pi \rightarrow D_\sigma$ such that for all $d \in D_\pi$, $f(d) = [\![\alpha]\!]^{M,g[v/d]}$

If the λ-expression is applied to an argument, we can simplify the interpretation:

- $[\![\lambda v\alpha]\!]^{M,g}([\![x]\!]^{M,g}) = [\![\alpha]\!]^{M,g[v/[\![x]\!]^{M,g}]}$

Example: "Bill is a student"

$[\![\lambda x(S(x))(b')]\!]^{M,g} = 1$ *iff* $[\![\lambda x(S(x))]\!]^{M,g(x/[\![b']\!]^{M,g})} = 1$ *iff* $[\![S(x)]\!]^{M,g'} = 1$ (*where* $g'=g[x/[\![b']\!]^{M,g}]$)

iff $[\![S]\!]^{M,g'}([\![x]\!]^{M,g'}) = 1$ *iff* $V_M(S)(V_M(b')) = 1$

➡ $[\![\lambda x(S(x))(b')]\!]^{M,g} = [\![S(b')]\!]^{M,g}$ *Function Application!*

β-Reduction

Winter: EFS Ch3
Page 67

Function application in Lambda Calculus

$$[\![\lambda v(\alpha)(\beta)]\!]^{M,g} = [\![\alpha]\!]^{M,g[v/[\![\beta]\!]^{M,g}]}$$

⇒ all (free) occurrences of the λ-variable in α get the interpretation of β as value.

This operation is called β-reduction

- $\lambda v(\alpha)(\beta) \Leftrightarrow \alpha[v/\beta]$
- where: $\alpha[v/\beta]$ is the result of replacing all free occurrences of v in α with β

Achtung: This equivalence is not unconditionally valid …

Variable capturing

Q: Are $\lambda v(\alpha)(\beta)$ and $\alpha[\beta/v]$ always equivalent?

- $\lambda x(\text{sing'}(x) \wedge \text{dance'}(x))(j') \Leftrightarrow \text{sing'}(j') \wedge \text{dance'}(j')$
- $\lambda x(\text{sing'}(x) \wedge \text{dance'}(x))(y) \Leftrightarrow \text{sing'}(y) \wedge \text{dance'}(y)$
- $\lambda x(\forall y\ \text{know'}(x)(y))(j') \Leftrightarrow \forall y\ \text{know}(j')(y)$
- $\lambda x(\forall y\ \text{know'}(x)(y))(y) \not\Leftrightarrow \forall y\ \text{know}(y)(y)$ $\Rightarrow$ **Problem: y is not "free for x"**

Definition: Let v, v' be variables of the same type, and let α be a WE of any type.

- v is free for v' in α iff no free occurrence of v' in α is in the scope of a quantifier or a λ-operator that binds v.

Conversion rules

Equivalence transformations in Lambda Calculus

- **β-conversion:** $\lambda v(\alpha)(\beta) \Leftrightarrow \alpha[v/\beta]$ *(α with all instances of v replaced by β)*
  (assuming all free variables in β are free for v in α)
- **α-conversion:** $\lambda v.\alpha \Leftrightarrow \lambda w.\alpha[v/w]$ *(α with all instances of v replaced by w)*
  (assuming w is free for v in α)
- **η-conversion:** $\lambda v.\alpha(v) \Leftrightarrow \alpha$

Quantifiers as lambda-expressions

- a student works ➔ $\exists x(\text{student}'(x) \wedge \text{work}'(x))$ :: t
  - a student ➔ $\lambda P \exists x(\text{student}'(x) \wedge P(x))$ :: $\langle\langle e,t\rangle,t\rangle$
  - a, some ➔ $\lambda Q \lambda P \exists x(Q(x) \wedge P(x))$ :: $\langle\langle e,t\rangle,\langle\langle e,t\rangle,t\rangle\rangle$
- every student ➔ $\lambda P \forall x(\text{student}'(x) \rightarrow P(x))$ :: $\langle\langle e,t\rangle,t\rangle$
  - every ➔ $\lambda Q \lambda P \forall x(Q(x) \rightarrow P(x))$ :: $\langle\langle e,t\rangle,\langle\langle e,t\rangle,t\rangle\rangle$
- no student ➔ $\lambda P \neg \exists x(\text{student}(x) \wedge P(x))$ :: $\langle\langle e,t\rangle,t\rangle$
  - no ➔ $\lambda Q \lambda P \neg \exists x(Q(x) \wedge P(x))$ :: $\langle\langle e,t\rangle,\langle\langle e,t\rangle,t\rangle\rangle$
- someone ➔ $\lambda F \exists x F(x)$ :: $\langle\langle e,t\rangle,t\rangle$

Quantifiers as lambda-expressions

Interpretation of expressions of type ⟨⟨e,t⟩,t⟩

- someone' $\in$ $CON_{\langle\langle e,t\rangle,t\rangle}$, so V_M(someone') $\in$ $D_{\langle\langle e,t\rangle,t\rangle}$
- $D_{\langle\langle e,t\rangle,t\rangle}$ is the set of functions from $D_{\langle e,t\rangle}$ to D_t
  i.e., the set of functions from $\mathcal{P}(U_M)$ (the powerset of U_M) to {0,1},
  which in turn is equivalent to $\mathcal{P}(\mathcal{P}(U_M))$

From V_M(someone') $\in$ $\mathcal{P}(\mathcal{P}(U_M))$ it follows that V_M(someone') $\subseteq$ $\mathcal{P}(U_M)$
More specifically:

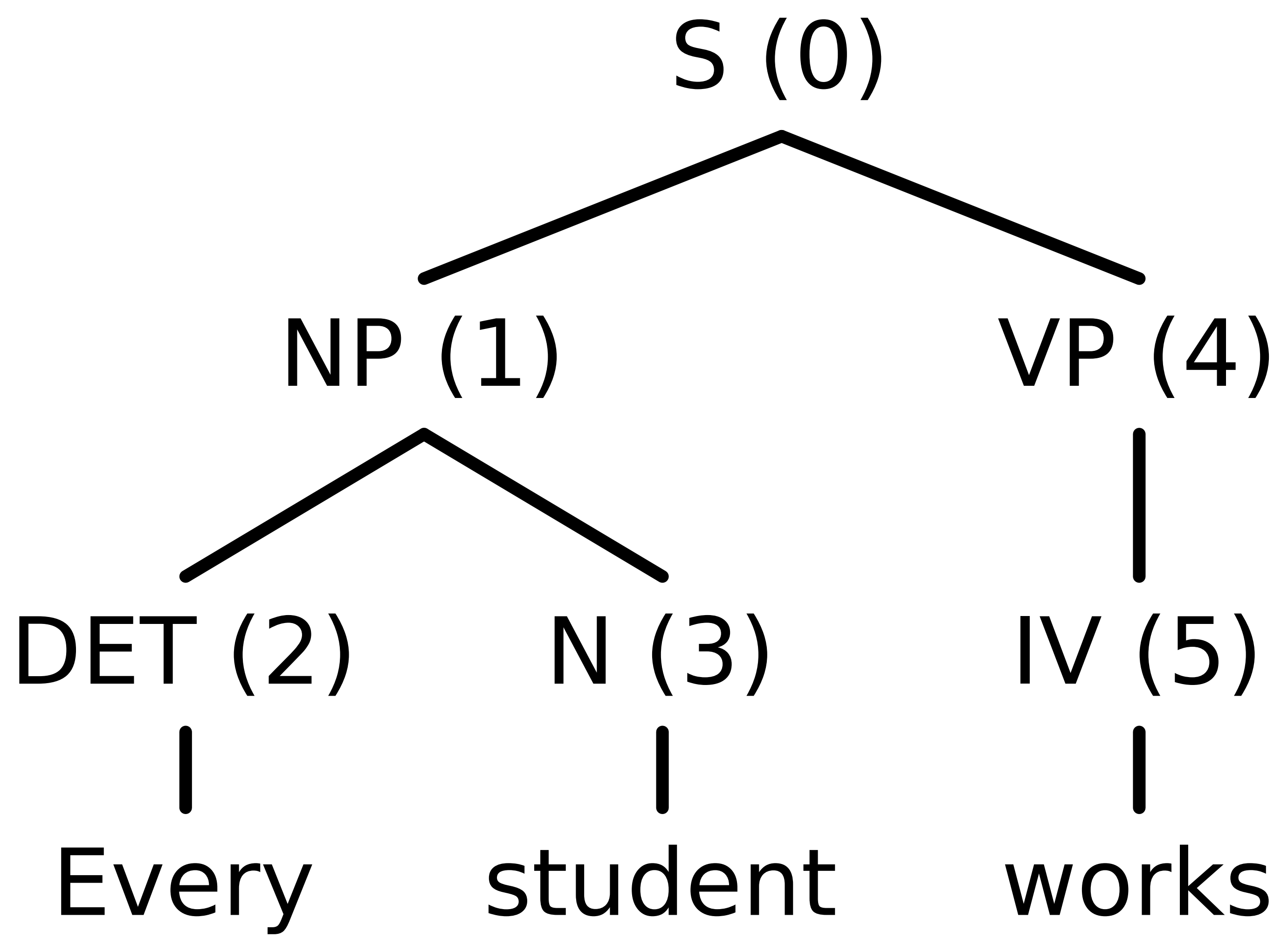
- V_M(someone') = $\{S \subseteq U_M \mid S \neq \varnothing\}$, if U_M is a domain of individuals

⇒ **More on quantified expressions in natural language in two weeks!**

Compositional construction

Example with quantified expression

Every student works.



(2) $\lambda P \lambda Q \forall x(P(x) \rightarrow Q(x)) :: \langle\langle e, t\rangle, \langle\langle e, t\rangle, t\rangle\rangle$

(3) $\lambda y.student'(y) \Leftrightarrow^{\eta} student' :: \langle e, t\rangle$

(1) $\lambda P \lambda Q \forall x(P(x) \rightarrow Q(x))(student')$
$\Leftrightarrow^{\beta} \lambda Q \forall x(student'(x) \rightarrow Q(x)) :: \langle\langle e, t\rangle, t\rangle$

(4)/(5) $\lambda z.work'(z) \Leftrightarrow^{\eta} work' :: \langle e, t\rangle$

(0) $\lambda Q \forall x(student'(x) \rightarrow Q(x))(work') \Leftrightarrow^{\beta} \forall x(student'(x) \rightarrow work'(x)) :: t$

Compositional construction

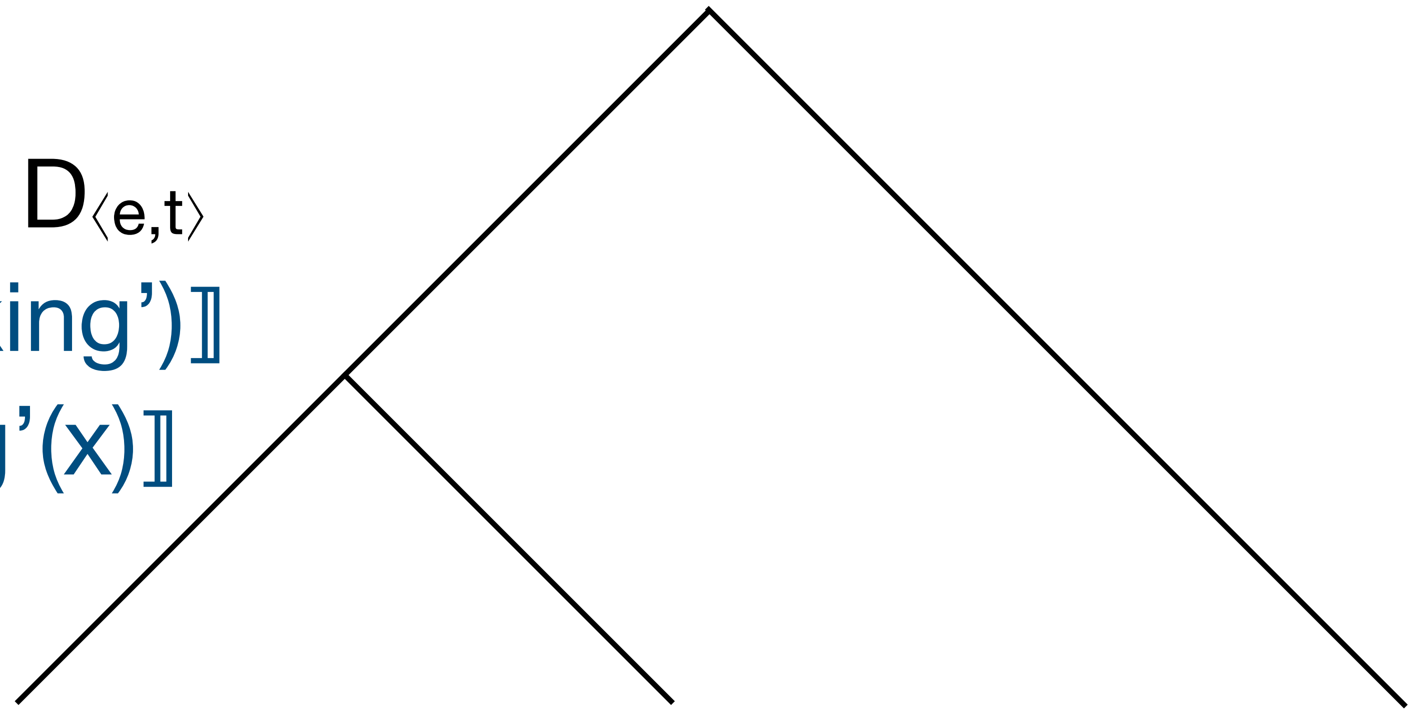
Example with higher-order expression

Not smoking is healthy => [[Not smoking] [is healthy]]

⟦healthy(NOT smoking)⟧ $\in D_t$

⟦λP.healthy’(P)(λx(¬smoking’(x)))⟧

$\Leftrightarrow^{\beta}$ ⟦healthy’(λx(¬smoking’(x)))⟧



⟦NOT smoking⟧ $\in D_{\langle e,t\rangle}$

⟦λQλx.¬Q(x)(smoking’)⟧

$\Leftrightarrow^{\beta}$ ⟦λx.¬smoking’(x)⟧

⟦NOT⟧ $\in D_{\langle\langle e,t\rangle,\langle e,t\rangle\rangle}$

⟦λPλx.¬P(x)⟧ $\Leftrightarrow^{\alpha}$ ⟦λQλx.¬Q(x)⟧

⟦smoking⟧ $\in D_{\langle e,t\rangle}$

⟦λx.smoking’(x)⟧ $\Leftrightarrow^{\alpha}$ ⟦smoking’⟧

⟦healthy⟧ $\in D_{\langle\langle e,t\rangle,t\rangle}$

⟦healthy’⟧ $\Leftrightarrow^{\alpha}$ ⟦λP.healthy’(P)⟧

Type Clash

When arguments and functions do not match

- **Problem:** In natural language, quantified expressions occur with transitive verbs in both subject and object position.
- **Example:** Someone reads a book

$$\dfrac{\text{someone} :: \langle\langle e, t\rangle,t\rangle \qquad \dfrac{\text{read} :: \langle e,\langle e, t\rangle\rangle \qquad \text{a book} :: \langle\langle e, t\rangle,t\rangle}{?? :: ??}}{?? :: t}$$

- **Solution**: reverse functor-argument relation (again!)
  - Logical form: *someone(read(a book))*
  - Use type raising to adjust the type of the transitive verb: $\text{read}_{\langle\langle\langle e, t\rangle,t\rangle,\langle e, t\rangle\rangle}$

Type Raising

Interpretation of type-raised expressions

What if we only change the type of the transitive verb?

- read ➔ read’ $\in CON_{\langle\langle\langle e,t\rangle, t\rangle, \langle e, t\rangle\rangle}$

$$[\![\text{someone reads a book}]\!] =$$
$$[\![\lambda F\exists xF(x)(read'(\lambda P\exists y(book'(y) \wedge P(y))))]\!]$$
$$\Leftrightarrow^{\beta} [\![\exists x(read'(\lambda P\exists y(book'(y) \wedge P(y)))(x)]\!]$$

… No further reduction steps possible.

Problem: this does not support the following entailment:

someone reads a book ⊨ there exists a book

Hence, we need a more explicit λ-term:

- read ➔ $\lambda Q\lambda z.Q(\lambda x(read^*(x)(z))) \in WE_{\langle\langle\langle e,t\rangle, t\rangle, \langle e, t\rangle\rangle}$

where: read* $\in WE_{\langle e, \langle e, t\rangle\rangle}$ is the “underlying” first-order relation

Compositionality with Transitive Verbs

Using type raised expresssions: Example

someone reads a book: someone(reads(a book))

$\lambda F \exists x F(x)(\lambda Q \lambda z(Q(\lambda x(read^*(x)(z))))(\lambda R \lambda P(\exists y(R(y) \wedge P(y)))(book')))$

$\Leftrightarrow^{\beta} \lambda F \exists x F(x)(\lambda Q \lambda z(Q(\lambda x(read^*(x)(z))))(\lambda P(\exists y(book'(y) \wedge P(y)))))$

$\Leftrightarrow^{\beta} \lambda F \exists x F(x)(\lambda z(\lambda P(\exists y(book'(y) \wedge P(y)))(\lambda x(read^*(x)(z)))))$

$\Leftrightarrow^{\beta} \lambda F \exists x F(x)(\lambda z(\exists y(book'(y) \wedge \lambda x(read^*(x)(z))(y))))$

$\Leftrightarrow^{\beta} \lambda F \exists x F(x)(\lambda z(\exists y(book'(y) \wedge read^*(y)(z))))$

$\Leftrightarrow^{\beta} \exists x(\lambda z(\exists y(book'(y) \wedge read^*(y)(z)))(x))$

$\Leftrightarrow^{\beta} \exists x \exists y(book'(y) \wedge read^*(y)(x))$

Reading material

Recommended reading

- Winter: Elements of Formal Semantics (Chapter 3, Part III)
  http://www.phil.uu.nl/~yoad/efs/main.html