Introduction to Formal Semantics
Lecture 4: Typed Lambda Calculus

Volha Petukhova & Nicolaie Dominik Dascalu

Spoken Language Systems Group
Saarland Univeristy

16.05.2022

UNIVERSITÄT
DES
SAARLANDES

Overview for today

- Recap: Predicate Logic
- Lambda abstraction
- Types
- Syntax and Semantics
- Application and beta reduction

Reading:

- Coppock, E., and Champollion, L. (2021). Invitation to formal semantics. Manuscript, Boston University and New York University (Ch.5)

Quizz (last week)

Let	van(x)	represent	'x is a van'
	car(x)	represent	'x is a car'
	bike(y)	represent	'y is a bike'
	expensive(x,y)		'x is more expensive y'
	faster(x,y)		'x is faster than y'

Translate the following formula into natural language:

1. $\forall y\ [\text{bike}(y) \implies \exists x\ [\text{car}(x) \wedge \text{expensive}(x,y)]]$
   Some cars are more expensive than any/every bike
   A car is more expensive than any/every bike
   Bikes are cheaper than some cars
2. $\forall x\ \forall y\ [[\text{van}(x) \wedge \text{bike}(y)] \implies \text{faster}(x,y)]$
   Vans are faster than bikes
   All vans are faster than all bikes
3. $\exists z\ [\text{car}(z) \wedge \forall x\ \forall y[[\text{van}(x)\ \&\ \text{bike}(y)] \implies \text{faster}(z,x) \wedge \text{faster}(z,y) \wedge \text{exp}(z,x) \wedge \text{exp}(z,y)]]]$
   A (particular) car is faster and more expensive than any van and any bike

Abstraction from fully specified FOL

Example

John loves Mary

Abstraction from fully specified FOL

Example

John loves Mary

$Loves(j, m)$

Abstraction from fully specified FOL

Example

John loves Mary

$Loves(j, m)$

$Loves(\_\_, m)$

Abstraction from fully specified FOL

Example

John loves Mary

$Loves(j, m)$

$Loves(\_\_, m)$

to abstract OVER the missing piece, ABSTRACTION OPERATOR λ is used

Abstraction from fully specified FOL

Example

John loves Mary

$Loves(j, m)$

$Loves(\_\_, m)$

to abstract OVER the missing piece, ABSTRACTION OPERATOR λ is used

Example

John loves Mary

$Loves(j, m)$

$Loves(\_\_, m)$

$\lambda x.Loves(x, m)$

Abstraction from fully specified FOL

Example

John loves Mary

$Loves(j, m)$

$Loves(\_\_, m)$

to abstract OVER the missing piece, ABSTRACTION OPERATOR λ is used

Example

John loves Mary

$Loves(j, m)$

$Loves(\_\_, m)$

$\lambda x.Loves(x, m)$

This expression denotes a function from an individual to truth-value

Abstraction from fully specified FOL (cont.)

The missing piece can be a predicate. We switch to HIGHER-ORDER LOGIC where variables ranging over predicates

Example

Abstraction from fully specified FOL (cont.)

The missing piece can be a predicate. We switch to HIGHER-ORDER LOGIC where variables ranging over predicates

Example

Everything is permanent.

Abstraction from fully specified FOL (cont.)

The missing piece can be a predicate. We switch to HIGHER-ORDER LOGIC where variables ranging over predicates

Example

Everything is permanent.

$\forall x.Permanent(x)$

Abstraction from fully specified FOL (cont.)

The missing piece can be a predicate. We switch to HIGHER-ORDER LOGIC where variables ranging over predicates

Example

Everything is permanent.

$\forall x.Permanent(x)$

$\forall x.\_\_(x)$

Abstraction from fully specified FOL (cont.)

The missing piece can be a predicate. We switch to HIGHER-ORDER LOGIC where variables ranging over predicates

Example

Everything is permanent.

$\forall x.Permanent(x)$

$\forall x.\_\_(x)$

$\lambda P.\forall x.P(x)$

The expression denotes a function from a predicate to a truth value

Lambda Expression

Lambda (λ) notation: (Church, 1940)

Lambda Expression

Lambda (λ) notation: (Church, 1940)

- Allows abstraction over FOL formulas

Lambda Expression

Lambda (λ) notation: (Church, 1940)

- Allows abstraction over FOL formulas
- Supports compositionality

Lambda Expression

Lambda (λ) notation: (Church, 1940)

- Allows abstraction over FOL formulas
- Supports compositionality

Form: λ + variable + FOL expression

Lambda Expression

Lambda (λ) notation: (Church, 1940)

- Allows abstraction over FOL formulas
- Supports compositionality

Form: λ + variable + FOL expression

Example

$\lambda x.P(x)$	function taking x to $P(x)$

Lambda Abstraction: Types

Syntactic categories of languages L_{Pred} are terms, predicates and formulas

Lambda Abstraction: Types

Syntactic categories of languages L_{Pred} are terms, predicates and formulas

Language L_λ has a set of TYPES which are recursively specified (of arbitrary complexity and depth), with two BASIC TYPES:

Lambda Abstraction: Types

Syntactic categories of languages L_{Pred} are terms, predicates and formulas

Language L_λ has a set of TYPES which are recursively specified (of arbitrary complexity and depth), with two BASIC TYPES:

- e **e**ntities for individuals corresponding to terms in L_{Pred}

Lambda Abstraction: Types

Syntactic categories of languages L_{Pred} are terms, predicates and formulas

Language L_λ has a set of TYPES which are recursively specified (of arbitrary complexity and depth), with two BASIC TYPES:

- e **e**ntities for individuals corresponding to terms in L_{Pred}
- t **t**ruth values for formulas

Lambda Abstraction: Types

Syntactic categories of languages L_{Pred} are terms, predicates and formulas

Language L_λ has a set of TYPES which are recursively specified (of arbitrary complexity and depth), with two BASIC TYPES:

- e **e**ntities for individuals corresponding to terms in L_{Pred}
- t **t**ruth values for formulas

and FUNCTION TYPES: $< e, t >$ denoting functions from individuals to truth values.

A set of types is defined recursively:

Lambda Abstraction: Types

Syntactic categories of languages L_{Pred} are terms, predicates and formulas

Language L_λ has a set of TYPES which are recursively specified (of arbitrary complexity and depth), with two BASIC TYPES:

- e **e**ntities for individuals corresponding to terms in L_{Pred}
- t **t**ruth values for formulas

and FUNCTION TYPES: $< e, t >$ denoting functions from individuals to truth values.
A set of types is defined recursively:

- e is a type

Lambda Abstraction: Types

Syntactic categories of languages L_{Pred} are terms, predicates and formulas

Language L_λ has a set of TYPES which are recursively specified (of arbitrary complexity and depth), with two BASIC TYPES:

- e **e**ntities for individuals corresponding to terms in L_{Pred}
- t **t**ruth values for formulas

and FUNCTION TYPES: $< e, t >$ denoting functions from individuals to truth values.

A set of types is defined recursively:

- e is a type
- t is a type

Lambda Abstraction: Types

Syntactic categories of languages L_{Pred} are terms, predicates and formulas

Language L_λ has a set of TYPES which are recursively specified (of arbitrary complexity and depth), with two BASIC TYPES:

- e **e**ntities for individuals corresponding to terms in L_{Pred}
- t **t**ruth values for formulas

and FUNCTION TYPES: $< e, t >$ denoting functions from individuals to truth values.
A set of types is defined recursively:

- e is a type
- t is a type
- if σ is a type and τ is a type then $< \sigma, \tau >$ is a type

Lambda Abstraction: Types

Syntactic categories of languages L_{Pred} are terms, predicates and formulas

Language L_λ has a set of TYPES which are recursively specified (of arbitrary complexity and depth), with two BASIC TYPES:

- e **e**ntities for individuals corresponding to terms in L_{Pred}
- t **t**ruth values for formulas

and FUNCTION TYPES: $< e, t >$ denoting functions from individuals to truth values.
A set of types is defined recursively:

- e is a type
- t is a type
- if σ is a type and τ is a type then $< \sigma, \tau >$ is a type
- nothing else is a type

Lambda Abstraction: Types (cont.)

Example

$< e, t >$ denotes function from individuals to truth values, e.g. standard predicate

Lambda Abstraction: Types (cont.)

Example

$< e, t >$ denotes function from individuals to truth values, e.g. standard predicate
$< e, e >$ denotes function from individuals to individuals, e.g. semantic relations like *loverOf* denoted by $\lambda x.loverOf(x)$

Lambda Abstraction: Types (cont.)

Example

$< e, t >$ denotes function from individuals to truth values, e.g. standard predicate
$< e, e >$ denotes function from individuals to individuals, e.g. semantic relations like *loverOf* denoted by $\lambda x.loverOf(x)$
$< e, < e, t >>$

- denotes relation called CURRYING binary relation, e.g. if left-to-right then function f such as $[f(x)]y = 1$ iff $(x, y) \in R$ results of applying f first to x and then $f(x)$ to y
- binary predicate, e.g. transitive verbs, $\lambda x.\lambda y.Loves(x, y)$ denotes the result of right-to-left currying the binary relation denoted by the binary predicate

Lambda Abstraction: Types (cont.)

Example

$< e, t >$ denotes function from individuals to truth values, e.g. standard predicate
$< e, e >$ denotes function from individuals to individuals, e.g. semantic relations like *loverOf* denoted by $\lambda x.loverOf(x)$
$< e, < e, t >>$

- denotes relation called CURRYING binary relation, e.g. if left-to-right then function f such as $[f(x)]y = 1$ iff $(x, y) \in R$ results of applying f first to x and then $f(x)$ to y
- binary predicate, e.g. transitive verbs, $\lambda x.\lambda y.Loves(x, y)$ denotes the result of right-to-left currying the binary relation denoted by the binary predicate

$<< e, t >, < e, t >>$ denote predicate modifiers, e.g. 'Pete drives <u>fast</u>'

Lambda Abstraction: Types (cont.)

Example

$< e, t >$ denotes function from individuals to truth values, e.g. standard predicate
$< e, e >$ denotes function from individuals to individuals, e.g. semantic relations like *loverOf* denoted by $\lambda x.loverOf(x)$
$< e, < e, t >>$

- denotes relation called CURRYING binary relation, e.g. if left-to-right then function f such as $[f(x)]y = 1$ iff $(x, y) \in R$ results of applying f first to x and then $f(x)$ to y
- binary predicate, e.g. transitive verbs, $\lambda x.\lambda y.Loves(x, y)$ denotes the result of right-to-left currying the binary relation denoted by the binary predicate

$<< e, t >, < e, t >>$ denote predicate modifiers, e.g. 'Pete drives fast'
$< e, < t, t >> | < e, << e, t >< e, t >>$ prepositions

Lambda Abstraction: Types (cont.)

Example

$< e, t >$ denotes function from individuals to truth values, e.g. standard predicate
$< e, e >$ denotes function from individuals to individuals, e.g. semantic relations like *loverOf* denoted by $\lambda x.loverOf(x)$
$< e, < e, t >>$

- denotes relation called CURRYING binary relation, e.g. if left-to-right then function f such as $[f(x)]y = 1$ iff $(x, y) \in R$ results of applying f first to x and then $f(x)$ to y
- binary predicate, e.g. transitive verbs, $\lambda x.\lambda y.Loves(x, y)$ denotes the result of right-to-left currying the binary relation denoted by the binary predicate

$<< e, t >, < e, t >>$ denote predicate modifiers, e.g. 'Pete drives <u>fast</u>'
$< e, < t, t >> \mid < e, << e, t >< e, t >>$ prepositions
$<< e, t >, << e, t >, t >>$ determiners

Lambda Abstraction: syntax

Syntactic Rule: Lambda Abstraction

If α is an expression of type τ and u is a variable of type σ then $[\lambda u.\alpha]$ is an expression of type $< \sigma, \tau >$

Lambda Abstraction: syntax

Syntactic Rule: Lambda Abstraction

If α is an expression of type τ and u is a variable of type σ then $[\lambda u.\alpha]$ is an expression of type $< \sigma, \tau >$

where σ is INPUT TYPE and τ is OUTPUT TYPE

Lambda Abstraction: semantics

Semantic Rule: Lambda Abstraction

If α is an expression of type τ and u is a variable of type σ then $[\![\lambda u.\alpha]\!]^{M,g}$ is that function f from D_σ into D_τ such that for all objects o in D_σ, $f(o) = [\![\alpha]\!]^{M,g[w \rightarrow o]}$

Lambda Abstraction: semantics

Semantic Rule: Lambda Abstraction

If α is an expression of type τ and u is a variable of type σ then $[\![\lambda u.\alpha]\!]^{M,g}$ is that function f from D_σ into D_τ such that for all objects o in D_σ, $f(o) = [\![\alpha]\!]^{M,g[w \to o]}$

$\lambda x.Happy(x)$ is of the form $\lambda u.\alpha$ and of $< e, t >$ type, so denotes function equal to $[\![Happy(x)]\!]^{M,g[x \to o]}$ and applying to all objects will return 1 (true) and 0 (false)

Lambda Reduction

Apply λ-expression to logical term

Lambda Reduction

Apply λ-expression to logical term

Example

$\lambda x.P(x)$

Lambda Reduction

Apply λ-expression to logical term

Example

$\lambda x.P(x)$

$\lambda x.P(x)(A)$

Lambda Reduction

Apply λ-expression to logical term

Example

$\lambda x.P(x)$

$\lambda x.P(x)(A)$

$P(A)$

Nested Lambda Reduction

Lambda expression as body of another

Nested Lambda Reduction

Lambda expression as body of another

Example

$\lambda x.\lambda y.Near(x, y)$

Nested Lambda Reduction

Lambda expression as body of another

Example

$\lambda x.\lambda y.Near(x, y)$
$\lambda x.\lambda y.Near(x, y)(midway)$

Nested Lambda Reduction

Lambda expression as body of another

Example

$\lambda x.\lambda y.Near(x, y)$
$\lambda x.\lambda y.Near(x, y)(midway)$
$\lambda y.Near(midway, y)$

Nested Lambda Reduction

Lambda expression as body of another

Example

$\lambda x.\lambda y.Near(x,y)$
$\lambda x.\lambda y.Near(x,y)(midway)$
$\lambda y.Near(midway,y)$
$\lambda y.Near(midway,y)(chicago)$

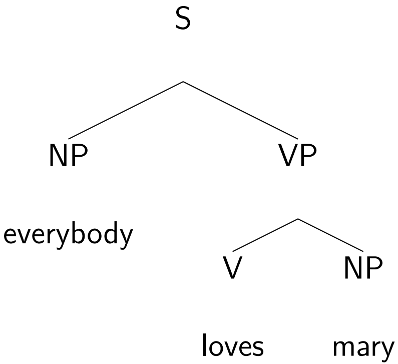
Nested Lambda Reduction

Lambda expression as body of another

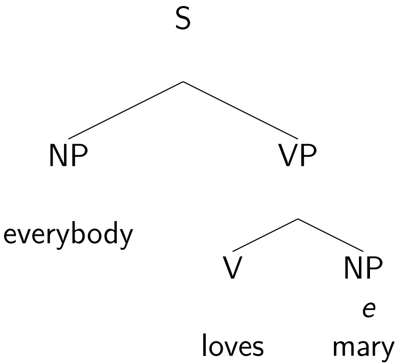
Example

$\lambda x.\lambda y.Near(x, y)$
$\lambda x.\lambda y.Near(x, y)(midway)$
$\lambda y.Near(midway, y)$
$\lambda y.Near(midway, y)(chicago)$
$Near(midway, chicago)$

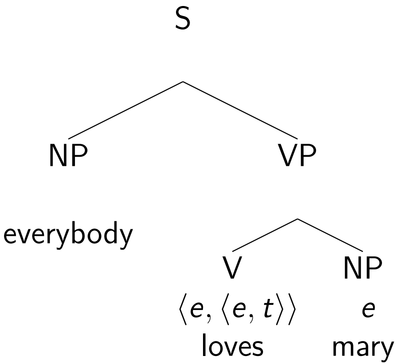
Lambda Reduction: Types



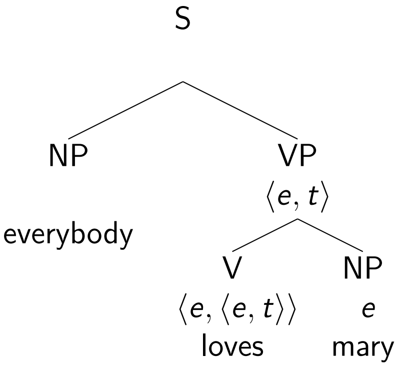
Lambda Reduction: Types



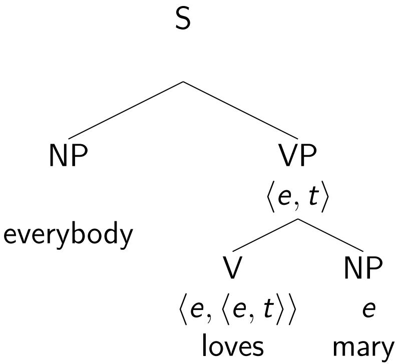
Lambda Reduction: Types



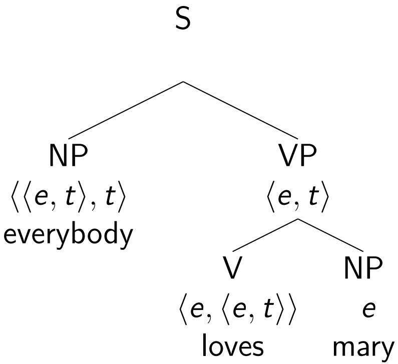
Lambda Reduction: Types



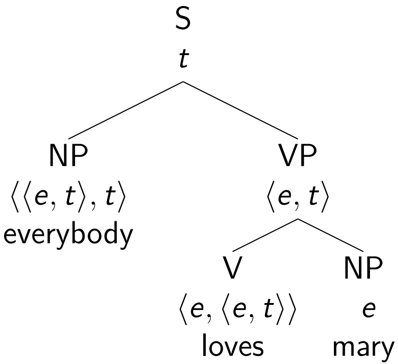
Lambda Reduction: Types



Lambda Reduction: Types



Lambda Reduction: Types



Syntax-driven Semantic Analysis

Supports compositionality: meaning of sentence constructed from meanings of parts, e.g. groupings and relations from syntax

Syntax-driven Semantic Analysis

Supports compositionality: meaning of sentence constructed from meanings of parts, e.g. groupings and relations from syntax

- Tie semantics to finite components of grammar, e.g. rules & lexicon
- Augment grammar rules with semantic info, aka “attachments”

Syntax-driven Semantic Analysis

Supports compositionality: meaning of sentence constructed from meanings of parts, e.g. groupings and relations from syntax

- Tie semantics to finite components of grammar, e.g. rules & lexicon
- Augment grammar rules with semantic info, aka “attachments”

Example

Every flight arrived.

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Syntax-driven Semantic Analysis

Supports compositionality: meaning of sentence constructed from meanings of parts, e.g. groupings and relations from syntax

- Tie semantics to finite components of grammar, e.g. rules & lexicon
- Augment grammar rules with semantic info, aka "attachments"

Example

Every flight arrived.

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Target representation: $\forall x.[\mathit{Flight}(x) \implies \mathit{Arrived}(x)]$

Creating Attachments

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Creating Attachments

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Noun → flight $\{\lambda x.\mathit{Flight}(x)\}$

Creating Attachments

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Noun $\rightarrow$ flight $\{\lambda x.Flight(x)\}$
Nom $\rightarrow$ Noun { Noun.sem }

Creating Attachments

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Noun $\rightarrow$ flight	$\{\lambda x.Flight(x)\}$
Nom $\rightarrow$ Noun	{ Noun.sem }
Det $\rightarrow$ Every	$\{\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)]\}$

Creating Attachments

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Noun $\rightarrow$ flight	$\{\lambda x.Flight(x)\}$
Nom $\rightarrow$ Noun	{ Noun.sem }
Det $\rightarrow$ Every	$\{\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)]\}$
NP $\rightarrow$ Det Nom	{ Det.sem(Nom.sem) }

Creating Attachments

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Noun → flight	$\{\lambda x.Flight(x)\}$
Nom → Noun	{ Noun.sem }
Det → Every	$\{\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)]\}$
NP → Det Nom	{ Det.sem(Nom.sem) }
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda x.Flight(x))$

Creating Attachments

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Noun $\rightarrow$ flight	$\{\lambda x.Flight(x)\}$
Nom $\rightarrow$ Noun	{ Noun.sem }
Det $\rightarrow$ Every	$\{\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)]\}$
NP $\rightarrow$ Det Nom	{ Det.sem(Nom.sem) }
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda x.Flight(x))$
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda y.Flight(y))$ (alpha conversion)

Creating Attachments

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Noun → flight	$\{\lambda x.Flight(x)\}$
Nom → Noun	{ Noun.sem }
Det → Every	$\{\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)]\}$
NP → Det Nom	{ Det.sem(Nom.sem) }
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda x.Flight(x))$
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda y.Flight(y))$ (alpha conversion)
	$\lambda Q.\forall x.[\lambda y.Flight(y)(x) \implies Q(x)]$

Creating Attachments

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Noun → flight	$\{\lambda x.Flight(x)\}$
Nom → Noun	{ Noun.sem }
Det → Every	$\{\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)]\}$
NP → Det Nom	{ Det.sem(Nom.sem) }
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda x.Flight(x))$
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda y.Flight(y))$ (alpha conversion)
	$\lambda Q.\forall x.[\lambda y.Flight(y)(x) \implies Q(x)]$
	$\lambda Q.\forall x.[Flight(x) \implies Q(x)]$

Creating Attachments

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Noun $\rightarrow$ flight	$\{\lambda x.Flight(x)\}$
Nom $\rightarrow$ Noun	{ Noun.sem }
Det $\rightarrow$ Every	$\{\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)]\}$
NP $\rightarrow$ Det Nom	{ Det.sem(Nom.sem) }
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda x.Flight(x))$
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda y.Flight(y))$ (alpha conversion)
	$\lambda Q.\forall x.[\lambda y.Flight(y)(x) \implies Q(x)]$
	$\lambda Q.\forall x.[Flight(x) \implies Q(x)]$
Verb $\rightarrow$ arrive	$\{\lambda y.Arrived(y)\}$

Creating Attachments

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Noun $\rightarrow$ flight	$\{\lambda x.Flight(x)\}$
Nom $\rightarrow$ Noun	{ Noun.sem }
Det $\rightarrow$ Every	$\{\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)]\}$
NP $\rightarrow$ Det Nom	{ Det.sem(Nom.sem) }
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda x.Flight(x))$
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda y.Flight(y))$ (alpha conversion)
	$\lambda Q.\forall x.[\lambda y.Flight(y)(x) \implies Q(x)]$
	$\lambda Q.\forall x.[Flight(x) \implies Q(x)]$
Verb $\rightarrow$ arrive	$\{\lambda y.Arrived(y)\}$
VP $\rightarrow$ Verb	{ Verb.sem }

Creating Attachments

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Noun → flight	$\{\lambda x.Flight(x)\}$
Nom → Noun	{ Noun.sem }
Det → Every	$\{\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)]\}$
NP → Det Nom	{ Det.sem(Nom.sem) }
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda x.Flight(x))$
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda y.Flight(y))$ (alpha conversion)
	$\lambda Q.\forall x.[\lambda y.Flight(y)(x) \implies Q(x)]$
	$\lambda Q.\forall x.[Flight(x) \implies Q(x)]$
Verb → arrive	$\{\lambda y.Arrived(y)\}$
VP → Verb	{ Verb.sem }
S → NP VP	{ NP.sem(VP.sem) }

Creating Attachments

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Noun $\rightarrow$ flight	$\{\lambda x.Flight(x)\}$
Nom $\rightarrow$ Noun	{ Noun.sem }
Det $\rightarrow$ Every	$\{\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)]\}$
NP $\rightarrow$ Det Nom	{ Det.sem(Nom.sem) }
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda x.Flight(x))$
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda y.Flight(y))$ (alpha conversion)
	$\lambda Q.\forall x.[\lambda y.Flight(y)(x) \implies Q(x)]$
	$\lambda Q.\forall x.[Flight(x) \implies Q(x)]$
Verb $\rightarrow$ arrive	$\{\lambda y.Arrived(y)\}$
VP $\rightarrow$ Verb	{ Verb.sem }
S $\rightarrow$ NP VP	{ NP.sem(VP.sem) }

$\lambda Q.\forall x.[Flight(x) \implies Q(x)](\lambda y.Arrived(y))$

Creating Attachments

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Noun $\rightarrow$ flight	$\{\lambda x.Flight(x)\}$
Nom $\rightarrow$ Noun	{ Noun.sem }
Det $\rightarrow$ Every	$\{\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)]\}$
NP $\rightarrow$ Det Nom	{ Det.sem(Nom.sem) }
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda x.Flight(x))$
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda y.Flight(y))$ (alpha conversion)
	$\lambda Q.\forall x.[\lambda y.Flight(y)(x) \implies Q(x)]$
	$\lambda Q.\forall x.[Flight(x) \implies Q(x)]$
Verb $\rightarrow$ arrive	$\{\lambda y.Arrived(y)\}$
VP $\rightarrow$ Verb	{ Verb.sem }
S $\rightarrow$ NP VP	{ NP.sem(VP.sem) }

$\lambda Q.\forall x.[Flight(x) \implies Q(x)](\lambda y.Arrived(y))$

$\forall x.[Flight(x) \implies \lambda y.Arrived(y)(x)]$

Creating Attachments

(S (NP (Det every) (Nom (Noun flight))) (VP (V arrived)))

Noun $\rightarrow$ flight	$\{\lambda x.Flight(x)\}$
Nom $\rightarrow$ Noun	{ Noun.sem }
Det $\rightarrow$ Every	$\{\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)]\}$
NP $\rightarrow$ Det Nom	{ Det.sem(Nom.sem) }
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda x.Flight(x))$
	$\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda y.Flight(y))$ (alpha conversion)
	$\lambda Q.\forall x.[\lambda y.Flight(y)(x) \implies Q(x)]$
	$\lambda Q.\forall x.[Flight(x) \implies Q(x)]$
Verb $\rightarrow$ arrive	$\{\lambda y.Arrived(y)\}$
VP $\rightarrow$ Verb	{ Verb.sem }
S $\rightarrow$ NP VP	{ NP.sem(VP.sem) }

$\lambda Q.\forall x.[Flight(x) \implies Q(x)](\lambda y.Arrived(y))$

$\forall x.[Flight(x) \implies \lambda y.Arrived(y)(x)]$

$\forall x.[Flight(x) \implies Arrived(x)]$

Extending Attachments

ProperNoun $\rightarrow$ UA223 $\{\lambda x.x(UA223)\}$

Extending Attachments

ProperNoun $\rightarrow$ UA223 $\{\lambda x.x(UA223)\}$

UA223

Extending Attachments

ProperNoun $\rightarrow$ UA223 $\{\lambda x.x(UA223)\}$

UA223

should produce correct form when applied to VP.sem as in "UA223 arrived"

$$Arrived(UA223)$$

Extending Attachments

ProperNoun $\rightarrow$ UA223 $\{\lambda x.x(UA223)\}$

UA223

should produce correct form when applied to VP.sem as in "UA223 arrived"

$$Arrived(UA223)$$

Determiner

Extending Attachments

ProperNoun → UA223 $\{\lambda x.x(UA223)\}$

UA223

should produce correct form when applied to VP.sem as in "UA223 arrived"

$$Arrived(UA223)$$

Determiner

Det → a $\{\lambda P.\lambda Q.\exists x.[P(x) \wedge Q(x)]\}$

Extending Attachments

ProperNoun $\rightarrow$ UA223 $\{\lambda x.x(UA223)\}$

UA223

should produce correct form when applied to VP.sem as in "UA223 arrived"

$Arrived(UA223)$

Determiner

Det $\rightarrow$ a $\{\lambda P.\lambda Q.\exists x.[P(x) \wedge Q(x)]\}$

a flight $\lambda Q.\exists x.[Flight(x) \wedge Q(x)]$

Extending Attachments

ProperNoun $\rightarrow$ UA223 $\{\lambda x.x(UA223)\}$

UA223

should produce correct form when applied to VP.sem as in "UA223 arrived"

$Arrived(UA223)$

Determiner

Det $\rightarrow$ a $\{\lambda P.\lambda Q.\exists x.[P(x) \wedge Q(x)]\}$

a flight $\lambda Q.\exists x.[Flight(x) \wedge Q(x)]$

Transitive Verb

Extending Attachments

ProperNoun → UA223 $\{\lambda x.x(UA223)\}$

UA223

should produce correct form when applied to VP.sem as in "UA223 arrived"

$Arrived(UA223)$

Determiner

Det → a $\{\lambda P.\lambda Q.\exists x.[P(x) \wedge Q(x)]\}$

a flight $\lambda Q.\exists x.[Flight(x) \wedge Q(x)]$

Transitive Verb

VP → Verb NP { Verb.sem(NP.sem) }

Extending Attachments

ProperNoun $\rightarrow$ UA223 $\{\lambda x.x(UA223)\}$

UA223

should produce correct form when applied to VP.sem as in "UA223 arrived"

$Arrived(UA223)$

Determiner

Det $\rightarrow$ a $\{\lambda P.\lambda Q.\exists x.[P(x) \wedge Q(x)]\}$

a flight $\lambda Q.\exists x.[Flight(x) \wedge Q(x)]$

Transitive Verb

VP $\rightarrow$ Verb NP { Verb.sem(NP.sem) }

Verb $\rightarrow$ booked

Extending Attachments

ProperNoun $\rightarrow$ UA223 $\{\lambda x.x(UA223)\}$

UA223

should produce correct form when applied to VP.sem as in "UA223 arrived"

$Arrived(UA223)$

Determiner

Det $\rightarrow$ a $\{\lambda P.\lambda Q.\exists x.[P(x) \wedge Q(x)]\}$

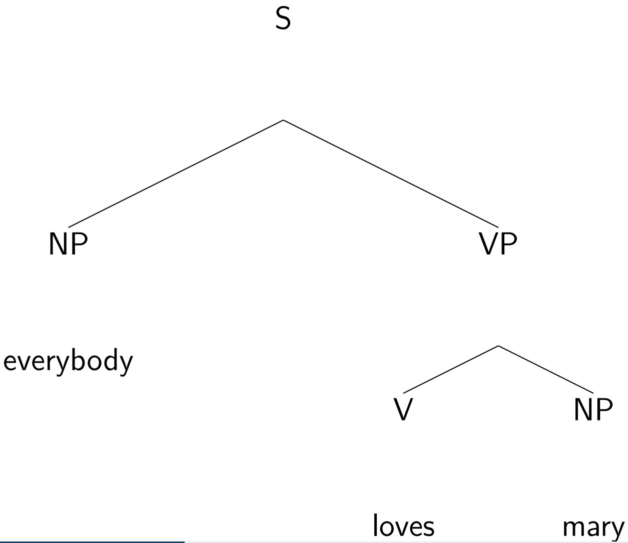
a flight $\lambda Q.\exists x.[Flight(x) \wedge Q(x)]$

Transitive Verb

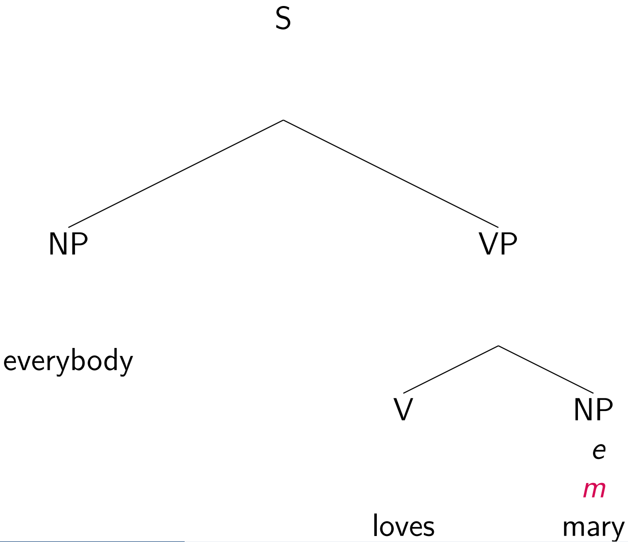
VP $\rightarrow$ Verb NP $\{$ Verb.sem(NP.sem) $\}$

Verb $\rightarrow$ booked $\lambda W.\lambda z.W(\lambda x.Booked(z,x))$

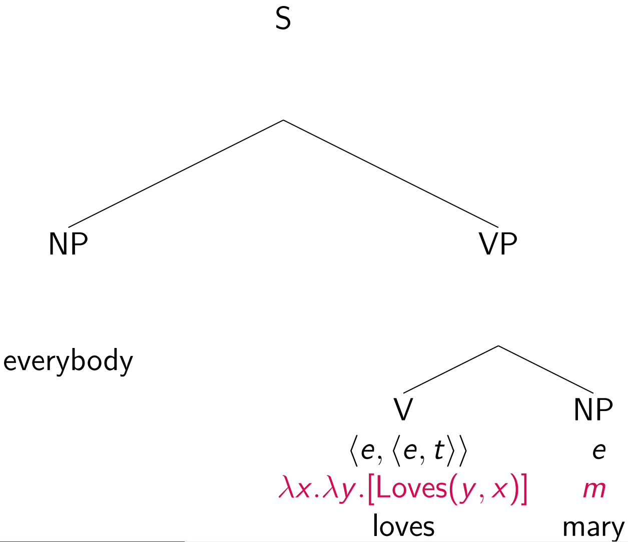
Lambda Abstraction: Types (cont.)



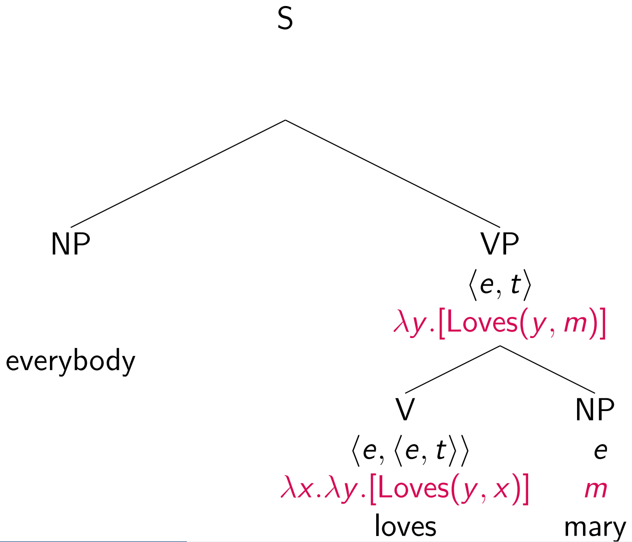
Lambda Abstraction: Types (cont.)



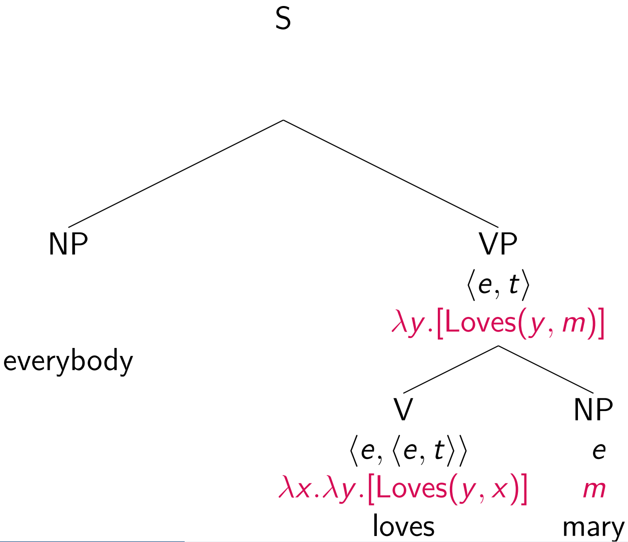
Lambda Abstraction: Types (cont.)



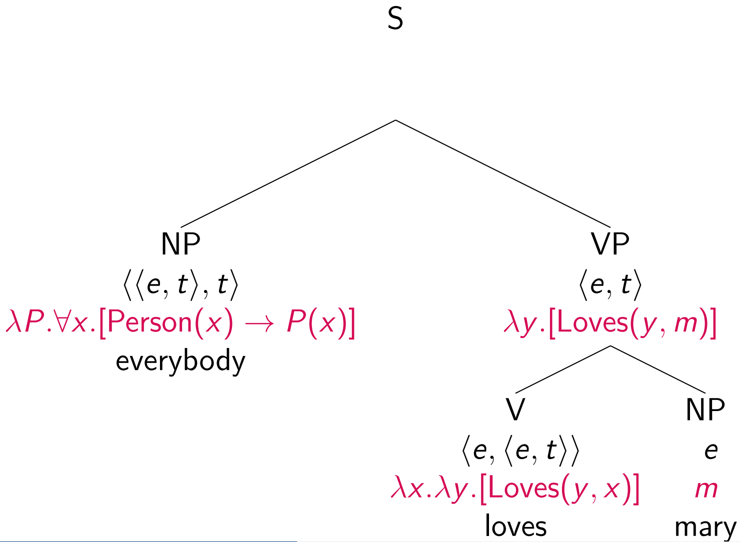
Lambda Abstraction: Types (cont.)



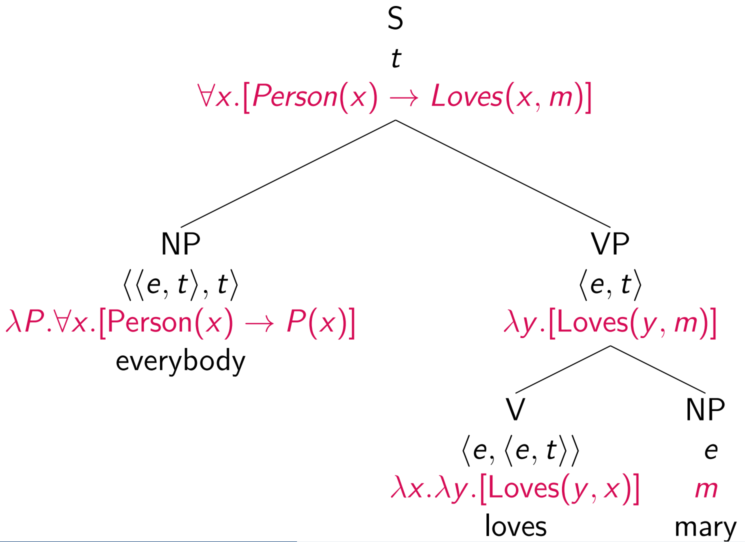
Lambda Abstraction: Types (cont.)



Lambda Abstraction: Types (cont.)



Lambda Abstraction: Types (cont.)



Strategy for Semantic Attachments

General approach:

- Create complex, lambda expressions with lexical items
  - Introduce quantifiers, predicates, terms
- Percolate up semantics from child if non-branching
- Apply semantics of one child to other through lambda
  - Combine elements, but don't introduce new

One more example

John booked a flight

One more example

John booked a flight

ProperNoun $\rightarrow$ john $\{\lambda x.x(john)\}$

One more example

John booked a flight

ProperNoun $\rightarrow$ john $\{\lambda x.x(\mathit{john})\}$

(john)

One more example

John booked a flight

ProperNoun $\rightarrow$ john	$\{\lambda x.x(john)\}$ (john)
a flight	$\lambda Q.\exists y.[Flight(y) \wedge Q(y)]$

One more example

John booked a flight

ProperNoun $\rightarrow$ john	$\{\lambda x.x(john)\}$ (john)
a flight	$\lambda Q.\exists y.[Flight(y) \wedge Q(y)]$
Verb $\rightarrow$ booked	

One more example

John booked a flight

ProperNoun $\rightarrow$ john	$\{\lambda x.x(john)\}$
	(john)
a flight	$\lambda Q.\exists y.[Flight(y) \wedge Q(y)]$
Verb $\rightarrow$ booked	$\lambda W.\lambda z.W(\lambda x.Booked(z,x))$

One more example

John booked a flight

ProperNoun $\rightarrow$ john	$\{\lambda x.x(john)\}$
	(john)
a flight	$\lambda Q.\exists y.[Flight(y) \wedge Q(y)]$
Verb $\rightarrow$ booked	$\lambda W.\lambda z.W(\lambda x.Booked(z,x))$
VP $\rightarrow$ Verb NP	{Verb.sem(NP.sem)}

One more example

John booked a flight

ProperNoun → john $\{\lambda x.x(john)\}$

(john)

a flight $\lambda Q.\exists y.[Flight(y) \wedge Q(y)]$

Verb → booked $\lambda W.\lambda z.W(\lambda x.Booked(z,x))$

VP → Verb NP $\{\text{Verb.sem(NP.sem)}\}$

$$\lambda W.\lambda z.W(\lambda x.Booked(z,x))(\lambda Q.\exists y.[Flight(y) \wedge Q(y))]$$

One more example

John booked a flight

ProperNoun → john	$\{\lambda x.x(john)\}$ (john)
a flight	$\lambda Q.\exists y.[Flight(y) \wedge Q(y)]$
Verb → booked	$\lambda W.\lambda z.W(\lambda x.Booked(z,x))$
VP → Verb NP	{Verb.sem(NP.sem)}

$\lambda W.\lambda z.W(\lambda x.Booked(z,x))(\lambda Q.\exists y.[Flight(y) \wedge Q(y))]$
$\lambda z.\lambda Q.\exists y.[Flight(y) \wedge Q(y)](\lambda x.Booked(z,x))$

One more example

John booked a flight

ProperNoun → john	$\{\lambda x.x(john)\}$ (john)
a flight	$\lambda Q.\exists y.[Flight(y) \wedge Q(y)]$
Verb → booked	$\lambda W.\lambda z.W(\lambda x.Booked(z,x))$
VP → Verb NP	{Verb.sem(NP.sem)}

$\lambda W.\lambda z.W(\lambda x.Booked(z,x))(\lambda Q.\exists y.[Flight(y) \wedge Q(y))]$
$\lambda z.\lambda Q.\exists y.[Flight(y) \wedge Q(y)](\lambda x.Booked(z,x))$
$\lambda z.\exists y.[Flight(y) \wedge \lambda x.Booked(z,x)(y)]$

One more example

John booked a flight

ProperNoun → john	$\{\lambda x.x(john)\}$ (john)
a flight	$\lambda Q.\exists y.[Flight(y) \wedge Q(y)]$
Verb → booked	$\lambda W.\lambda z.W(\lambda x.Booked(z,x))$
VP → Verb NP	{Verb.sem(NP.sem)}

$\lambda W.\lambda z.W(\lambda x.Booked(z,x))(\lambda Q.\exists y.[Flight(y) \wedge Q(y))]$
$\lambda z.\lambda Q.\exists y.[Flight(y) \wedge Q(y)](\lambda x.Booked(z,x))$
$\lambda z.\exists y.[Flight(y) \wedge \lambda x.Booked(z,x)(y)]$
$\lambda z.\exists y.[Flight(y) \wedge Booked(z,y)]$

One more example

John booked a flight

ProperNoun → john $\{\lambda x.x(john)\}$

(john)

a flight $\lambda Q.\exists y.[Flight(y) \wedge Q(y)]$

Verb → booked $\lambda W.\lambda z.W(\lambda x.Booked(z,x))$

VP → Verb NP {Verb.sem(NP.sem)}

$\lambda W.\lambda z.W(\lambda x.Booked(z,x))(\lambda Q.\exists y.[Flight(y) \wedge Q(y))]$

$\lambda z.\lambda Q.\exists y.[Flight(y) \wedge Q(y)](\lambda x.Booked(z,x))$

$\lambda z.\exists y.[Flight(y) \wedge \lambda x.Booked(z,x)(y)]$

$\lambda z.\exists y.[Flight(y) \wedge Booked(z,y)]$

S → NP VP {NP.sem(VP.sem)}

One more example

John booked a flight

ProperNoun → john	$\{\lambda x.x(john)\}$ (john)
a flight	$\lambda Q.\exists y.[Flight(y) \wedge Q(y)]$
Verb → booked	$\lambda W.\lambda z.W(\lambda x.Booked(z,x))$
VP → Verb NP	{Verb.sem(NP.sem)}

$\lambda W.\lambda z.W(\lambda x.Booked(z,x))(\lambda Q.\exists y.[Flight(y) \wedge Q(y))]$
$\lambda z.\lambda Q.\exists y.[Flight(y) \wedge Q(y)](\lambda x.Booked(z,x))$
$\lambda z.\exists y.[Flight(y) \wedge \lambda x.Booked(z,x)(y)]$
$\lambda z.\exists y.[Flight(y) \wedge Booked(z,y)]$

S → NP VP	{NP.sem(VP.sem)}

$\lambda z.\exists y.[Flight(y) \wedge Booked(z,y)](john)$

One more example

John booked a flight

ProperNoun → john $\{\lambda x.x(john)\}$

(john)

a flight $\lambda Q.\exists y.[Flight(y) \wedge Q(y)]$

Verb → booked $\lambda W.\lambda z.W(\lambda x.Booked(z,x))$

VP → Verb NP {Verb.sem(NP.sem)}

$\lambda W.\lambda z.W(\lambda x.Booked(z,x))(\lambda Q.\exists y.[Flight(y) \wedge Q(y))]$

$\lambda z.\lambda Q.\exists y.[Flight(y) \wedge Q(y)](\lambda x.Booked(z,x))$

$\lambda z.\exists y.[Flight(y) \wedge \lambda x.Booked(z,x)(y)]$

$\lambda z.\exists y.[Flight(y) \wedge Booked(z,y)]$

S → NP VP {NP.sem(VP.sem)}

$\lambda z.\exists y.[Flight(y) \wedge Booked(z,y)](john)$

$\exists y.[Flight(y) \wedge Booked(john,y)]$

Quizz for Today

TBA