Introduction to Formal Semantics
Lecture 4: Typed Lambda Calculus

Volha Petukhova & Nicolaie Dominik Dascalu
Spoken Language Systems Group
Saarland University
16.05.2022
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 \( \text{van}(x) \) represent ‘\( x \) is a van’
\( \text{car}(x) \) represent ‘\( x \) is a car’
\( \text{bike}(y) \) represent ‘\( y \) is a bike’
\( \text{expensive}(x,y) \) ‘\( x \) is more expensive \( y \)’
\( \text{faster}(x,y) \) ‘\( x \) is faster than \( y \)’

Translate the following formula into natural language:

1. \( \forall y \left[ \text{bike}(y) \implies \exists x \left[ \text{car}(x) \land \text{expensive}(x,y) \right] \right] \)

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. \( \forall x \forall y \left[ \left[ \text{van}(x) \land \text{bike}(y) \right] \implies \text{faster}(x,y) \right] \)

Vans are faster than bikes
All vans are faster than all bikes

3. \( \exists z \left[ \text{car}(z) \land \forall x \forall y \left[ \left[ \text{van}(x) \land \text{bike}(y) \right] \implies \text{faster}(z,x) \land \text{faster}(z,y) \land \text{exp}(z,x) \land \text{exp}(z,y) \right] \right] \)

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

$Loves(j, m)$

This expression denotes a function from an individual to truth-value
Abstraction from fully specified FOL

Example

John loves Mary

\[ \text{Loves}(j, m) \]

\[ \text{Loves}(\_\_, m) \]
Abstraction from fully specified FOL

Example

John loves Mary

$Loves(j, m)$

$Loves(\_\_, m)$

to abstract OVER the missing piece, ABSTRACTION OPERATOR $\lambda$ is used
Abstraction from fully specified FOL

Example

John loves Mary

$Loves(j, m)$

$Loves(_, m)$

to abstract OVER the missing piece, ABSTRACTION OPERATOR $\lambda$ 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 \( \lambda \) 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
The missing piece can be a predicate. We switch to HIGHER-ORDER LOGIC where variables ranging over predicates

Example

Everything is permanent.

$\forall x. \text{Permanent}(x)$

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

∀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.*

\[ \forall x.\text{Permanent}(x) \]

\[ \forall x.\neg(x) \]
The missing piece can be a predicate. We switch to HIGHER-ORDER LOGIC where variables ranging over predicates

Example

Everything is permanent.

\( \forall x. \text{Permanent}(x) \)

\( \forall x. \square(x) \)

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

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

Lambda ($\lambda$) notation: (Church, 1940)

- Allows abstraction over FOL formulas
- Supports compositionality

Form: $\lambda$ + variable + FOL expression

Example: $\lambda x. P(x)$ function taking $x$ to $P(x)$
Lambda Expression

Lambda (\( \lambda \)) notation: (Church, 1940)

- Allows abstraction over FOL formulas
Lambda (λ) notation: (Church, 1940)

- Allows abstraction over FOL formulas
- Supports compositionality
Lambda Expression

Lambda (\(\lambda\)) notation: (Church, 1940)

- Allows abstraction over FOL formulas
- Supports compositionality

Form: \(\lambda + \text{variable} + \text{FOL expression}\)
Lambda Expression

Lambda (\(\lambda\)) notation: (Church, 1940)

- Allows abstraction over FOL formulas
- Supports compositionality

Form: \(\lambda + \text{variable} + \text{FOL expression}\)

Example

\(\lambda x. P(x)\) function taking \(x\) to \(P(x)\)
Lambda Abstraction: Types

Syntactic categories of languages $L_{\text{Pred}}$ are terms, predicates and formulas
Lambda Abstraction: Types

Syntactic categories of languages $L_{Pred}$ are terms, predicates and formulas
Language $L_{\lambda}$ 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 $L_{\lambda}$ has a set of TYPES which are recursively specified (of arbitrary complexity and depth), with two BASIC TYPES:

- $e$ entities for individuals corresponding to terms in $L_{Pred}$

- Function TYPES: $\langle e, t \rangle$ denoting functions from individuals to truth values.

A set of types is defined recursively:

- $e$ is a type
- $t$ is a type
- if $\sigma$ is a type and $\tau$ is a type then $\langle \sigma, \tau \rangle$ is a type
- nothing else is a type
Lambda Abstraction: Types

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

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

- $e$ entities for individuals corresponding to terms in $L_{Pred}$
- $t$ truth values for formulas
Lambda Abstraction: Types

Syntactic categories of languages $L_{\text{Pred}}$ are terms, predicates and formulas.

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

- $e$ entities for individuals corresponding to terms in $L_{\text{Pred}}$
- $t$ truth values for formulas

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

A set of types is defined recursively:

\[
\begin{align*}
\text{• } e & \text{ is a type} \\
\text{• } t & \text{ is a type} \\
\text{• } \text{if } \sigma & \text{ is a type and } \tau \text{ is a type then } < \sigma, \tau > \text{ is a type} \\
\text{• } \text{nothing else is a type}
\end{align*}
\]
Lambda Abstraction: Types

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

Language $L_{\lambda}$ has a set of TYPES which are recursively specified (of arbitrary complexity and depth), with two BASIC TYPES:

- $e$ entities for individuals corresponding to terms in $L_{Pred}$
- $t$ truth 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 $L_{\text{Pred}}$ are terms, predicates and formulas. Language $L_\lambda$ has a set of TYPES which are recursively specified (of arbitrary complexity and depth), with two BASIC TYPES:

- $e$ entities for individuals corresponding to terms in $L_{\text{Pred}}$
- $t$ truth 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 $L_{\lambda}$ has a set of TYPES which are recursively specified (of arbitrary complexity and depth), with two BASIC TYPES:

- $e$ entities for individuals corresponding to terms in $L_{Pred}$
- $t$ truth 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 $\sigma$ is a type and $\tau$ is a type then $< \sigma, \tau >$ is a type
Lambda Abstraction: Types

Syntactic categories of languages $L_{Pred}$ are terms, predicates and formulas.
Language $L_{\lambda}$ has a set of TYPES which are recursively specified (of arbitrary complexity and depth), with two BASIC TYPES:

- $e$ entities for individuals corresponding to terms in $L_{Pred}$
- $t$ truth 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 $\sigma$ is a type and $\tau$ 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

\(< e, t >\) denotes function from individuals to individuals, e.g. semantic relations like `loverOf` denoted by \(\lambda x.\lambda y.\) `loverOf`\((x)\)\. \(\cdot\) denotes relation called CURRYING binary relation, e.g. if left-to-right then function \(f\) such as \[f(x)\] equals 1 iff \((x, y)\) \(\in\) \(R\) results of applying \(f\) first to \(x\) and then \(f(x)\) to \(y\) \(\cdot\) 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

\(< e, t >\) denote predicate modifiers, e.g. 'Pete drives fast' \(< e, t >|< e, t >\) \(< e, t >\) \(< e, t >\) prepositions \(< e, t >,\) \(< e, t >, t\) \(< e, t >, t\) determiners
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 \(\text{loverOf}\) denoted by \(\lambda x.\text{loverOf}(x)\)
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 
  \([f(x)]y = 1 \text{ iff } (x, y) \in R\) 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 \(\text{loverOf}\) denoted by \(\lambda x.\text{loverOf}(x)\)
\(< e, < e, t >>\)

- 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, \(\lambda x.\lambda y.\text{Loves}(x, y)\) 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’
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 \(\text{loverOf}\) denoted by \(\lambda x. \text{loverOf}(x)\)
\(<e, <e, t>>\)

- denotes relation called CURRYING binary relation, e.g. if left-to-right then function \(f\) such as 
  \([f(x)]y = 1 \text{ iff } (x, y) \in R\) results of applying \(f\) first to \(x\) and then \(f(x)\) to \(y\)
- binary predicate, e.g. transitive verbs, \(\lambda x. \lambda y. \text{Loves}(x, y)\) 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 \(\text{loverOf}\) denoted by \(\lambda x.\text{loverOf}(x)\)
\(<e, <e, t>>\)

- 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, \(\lambda x.\lambda y.\text{Loves}(x, y)\) 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
\(<<<e, t>>, <<e, t>>, t>>\) determiners
Syntactic Rule: Lambda Abstraction

If $\alpha$ is an expression of type $\tau$ and $u$ is a variable of type $\sigma$ then $[\lambda u. \alpha]$ is an expression of type $<\sigma, \tau>$.
Lambda Abstraction: syntax

**Syntactic Rule: Lambda Abstraction**

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

where $\sigma$ is INPUT TYPE and $\tau$ is OUTPUT TYPE
Lambda Abstraction: semantics

Semantic Rule: Lambda Abstraction

If $\alpha$ is an expression of type $\tau$ and $u$ is a variable of type $\sigma$ then $\left[\lambda u.\alpha\right]^{M,g}$ is that function $f$ from $D_\sigma$ into $D_\tau$ such that for all objects $o$ in $D_\sigma$, $f(o) = \left[\alpha\right]^{M,g}[w\rightarrow o]$. 

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

Semantic Rule: Lambda Abstraction

If $\alpha$ is an expression of type $\tau$ and $u$ is a variable of type $\sigma$ then $\llbracket \lambda u. \alpha \rrbracket_{M,g}$ is that function $f$ from $D_\sigma$ into $D_\tau$ such that for all objects $o$ in $D_\sigma$, $f(o) = \llbracket \alpha \rrbracket_{M,g[w \mapsto o]}$

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

Apply $\lambda$-expression to logical term
Lambda Reduction

Apply $\lambda$-expression to logical term

Example

$\lambda x. P(x)$
Lambda Reduction

Apply $\lambda$-expression to logical term

Example

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

Apply $\lambda$-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

\[ \lambda x. \lambda y. \text{Near} (x, y) \]

\[ \lambda y. \text{Midway} (\text{Midway}, y) \]

\[ \lambda y. \text{Midway} (\text{Midway}, \text{Chicago}) \]
Nested Lambda Reduction

Lambda expression as body of another

Example

\( \lambda x. \lambda y. \text{Near}(x, y) \)
Nested Lambda Reduction

Lambda expression as body of another

Example

\( \lambda x. \lambda y. \text{Near}(x, y) \)
\( \lambda x. \lambda y. \text{Near}(x, y)(\text{midway}) \)
Nested Lambda Reduction

Lambda expression as body of another

Example

\( \lambda x. \lambda y. \text{Near}(x, y) \)
\( \lambda x. \lambda y. \text{Near}(x, y)(\text{midway}) \)
\( \lambda y. \text{Near}(\text{midway}, y) \)
Nested Lambda Reduction

Lambda expression as body of another

Example

\[ \lambda x. \lambda y. \text{Near}(x, y) \]
\[ \lambda x. \lambda y. \text{Near}(x, y)(\text{midway}) \]
\[ \lambda y. \text{Near}(\text{midway}, y) \]
\[ \lambda y. \text{Near}(\text{midway}, y)(\text{chicago}) \]
Nested Lambda Reduction

Lambda expression as body of another

Example

\( \lambda x. \lambda y. \text{Near}(x, y) \)
\( \lambda x. \lambda y. \text{Near}(x, y)(\text{midway}) \)
\( \lambda y. \text{Near}(\text{midway}, y) \)
\( \lambda y. \text{Near}(\text{midway}, y)(\text{chicago}) \)
\( \text{Near}(\text{midway}, \text{chicago}) \)
Lambda Reduction: Types

\[
S \\
NP \quad VP
\]

\[
\text{everybody} \\
V \quad NP
\]

\[
\text{loves} \quad \text{mary}
\]
Lambda Reduction: Types

\[ S \]

\[ \text{NP} \quad \text{VP} \]

everybody

\[ \text{V} \quad \text{NP} \]

loves  e  mary
Lambda Reduction: Types

S

NP  VP

everybody

V  NP

⟨e, ⟨e, t⟩⟩  e

loves  mary
Lambda Reduction: Types

\[
S \\
NP \quad VP \\
\text{everybody} \quad \langle e, t \rangle \\
V \quad NP \\
\langle e, \langle e, t \rangle \rangle \quad e \\
\text{loves} \quad \text{mary}
\]
Lambda Reduction: Types

S

NP  VP

everybody  \langle e, t \rangle

V  NP

\langle e, \langle e, t \rangle \rangle  e

loves  mary
Lambda Reduction: Types

\[
S \\
\quad \text{NP} \quad \text{VP} \\
\quad \langle \langle e, t \rangle, t \rangle \quad \langle e, t \rangle \\
\quad \text{everybody} \\
\quad \text{V} \\
\quad \langle e, \langle e, t \rangle \rangle \\
\quad \text{VP} \\
\quad \langle e, t \rangle \\
\quad \text{NP} \\
\quad e \\
\quad \text{loves} \\
\quad \text{mary}
\]
Lambda Reduction: Types

\[
\begin{align*}
S & \\
  t & \\
\text{NP} & \\
  \langle \langle e, t \rangle, t \rangle & \\
\text{everybody} & \\
\text{VP} & \\
  \langle e, t \rangle & \\
\text{V} & \\
  \langle e, \langle e, t \rangle \rangle & \\
\text{loves} & \\
\text{NP} & \\
  e & \\
\text{mary} & 
\end{align*}
\]
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: ∀x. [Flight(x) → 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.\texttt{Flight}(x)}
Creating Attachments

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

Noun → flight \{λx.\text{Flight}(x)\}

Nom → Noun \{\text{Noun.sem} \}

{\text{o.petukhova; nddascalu}@lsv.uni-saarland.de}
Creating Attachments

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

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

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Noun</td>
<td>flight</td>
</tr>
<tr>
<td>Nom</td>
<td>Noun</td>
</tr>
<tr>
<td>Det</td>
<td>Every</td>
</tr>
<tr>
<td>NP</td>
<td>Det Nom</td>
</tr>
</tbody>
</table>

\begin{align*}
\forall x. [\text{Flight}(x) \implies \text{Arrived}(x)] & \\
\forall x. [\text{Flight}(x) \implies \lambda y. \text{Arrived}(y)(x)] & \\
\forall x. [\text{Flight}(x) \implies \text{Arrived}(x)] & \\
\end{align*}
Creating Attachments

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

Noun → flight \{λx.\text{Flight}(x)\}
Nom → Noun \{\text{Noun.sem}\}
Det → Every \{λP.λQ.∀x.[P(x) ⇒ Q(x)]\}
NP → Det Nom \{\text{Det.sem(Nom.sem)}\}
λP.λQ.∀x.[P(x) ⇒ Q(x)](λx.\text{Flight}(x))
Creating Attachments

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

\[
\begin{align*}
\text{Noun} & \rightarrow \text{flight} & \{\lambda x.\text{Flight}(x)\} \\
\text{Nom} & \rightarrow \text{Noun} & \{\ \text{Noun.sem}\ \} \\
\text{Det} & \rightarrow \text{Every} & \{\lambda P.\lambda Q.\forall x. [P(x) \implies Q(x)]\} \\
\text{NP} & \rightarrow \text{Det Nom} & \{\ \text{Det.sem(Nom.sem)}\ \}
\end{align*}
\]

\[
\begin{align*}
\lambda P.\lambda Q.\forall x. [P(x) \implies Q(x)](\lambda x.\text{Flight}(x)) \\
\lambda P.\lambda Q.\forall x. [P(x) \implies Q(x)](\lambda y.\text{Flight}(y)) \ (\text{alpha conversion})
\end{align*}
\]
Creating Attachments

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

Noun → flight \{λx.\text{Flight}(x)\}
Nom → Noun \{ \text{Noun.sem} \}
Det → Every \{λP.λQ.∀x.[P(x) \implies Q(x)]\}
NP → Det Nom \{ \text{Det.sem(Nom.sem)} \}

\lambda P.\lambda Q.∀x.[P(x) \implies Q(x)](\lambda x.\text{Flight}(x))
\lambda P.\lambda Q.∀x.[P(x) \implies Q(x)](\lambda y.\text{Flight}(y)) \text{ (alpha conversion)}
\lambda Q.∀x.[\lambda y.\text{Flight}(y)(x) \implies Q(x)]
Creating Attachments

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

Noun → flight \{ λx.\text{Flight}(x) \}
Nom → Noun \{ \text{Noun.sem} \}
Det → Every \{ λP.λQ.∀x.[P(x) \implies Q(x)] \}
NP → Det Nom \{ \text{Det.sem(Nom.sem)} \}

\[
\begin{align*}
\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda x.\text{Flight}(x)) & \\
\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda y.\text{Flight}(y)) & \text{(alpha conversion)} \\
\lambda Q.\forall x.[\lambda y.\text{Flight}(y)(x) \implies Q(x)] & \\
\lambda Q.\forall x.[\text{Flight}(x) \implies Q(x)] & \\
\end{align*}
\]
Creating Attachments

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

Noun → flight \{ \lambda x.\text{Flight}(x) \}\nNom → Noun \{ \text{Noun.sem} \}\nDet → Every \{ \lambda P.\lambda Q. \forall x. [P(x) \implies Q(x)] \}\nNP → Det Nom \{ \text{Det.sem(Nom.sem)} \}

\begin{align*}
\lambda P.\lambda Q. \forall x. [P(x) \implies Q(x)](\lambda x.\text{Flight}(x)) \\
\lambda P.\lambda Q. \forall x. [P(x) \implies Q(x)](\lambda y.\text{Flight}(y)) \text{ (alpha conversion)} \\
\lambda Q. \forall x. [\lambda y.\text{Flight}(y)(x) \implies Q(x)] \\
\lambda Q. \forall x. [\text{Flight}(x) \implies Q(x)]
\end{align*}

Verb → arrive \{ \lambda y.\text{Arrived}(y) \}
Creating Attachments

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

Noun → flight  \{\lambda x.\text{Flight}(x)\}\nNom → Noun  \{\text{Noun.sem}\}\nDet → Every  \{\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)]\}\nNP → Det Nom  \{\text{Det.sem(Nom.sem)}\}\n
\begin{align*}
\lambda P.\lambda Q.\forall x.[P(x) & \implies Q(x)](\lambda x.\text{Flight}(x)) \\
\lambda P.\lambda Q.\forall x.[P(x) & \implies Q(x)](\lambda y.\text{Flight}(y)) & \text{(alpha conversion)} \\
\lambda Q.\forall x.[\lambda y.\text{Flight}(y)(x) & \implies Q(x)] \\
\lambda Q.\forall x.[\text{Flight}(x) & \implies Q(x)]
\end{align*}

Verb → arrive  \{\lambda y.\text{Arrived}(y)\}\nVP → Verb  \{\text{Verb.sem}\}
Creating Attachments

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

Noun → flight \{ λx.\textit{Flight}(x) \}
Nom → Noun \{ \textit{Noun}.\textit{sem} \}
Det → Every \{ λP.λQ.∀x.[P(x) \implies Q(x)] \}
NP → Det Nom \{ Det.\textit{sem}(\textit{Nom}.\textit{sem}) \}

\begin{align*}
\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda x.\textit{Flight}(x)) \\
\lambda P.\lambda Q.\forall x.[P(x) \implies Q(x)](\lambda y.\textit{Flight}(y)) \text{ (alpha conversion)} \\
\lambda Q.\forall x.[\lambda y.\textit{Flight}(y)(x) \implies Q(x)] \\
\lambda Q.\forall x.\textit{Flight}(x) \implies Q(x)
\end{align*}

Verb → arrive \{ λy.\textit{Arrived}(y) \}
VP → Verb \{ \textit{Verb}.\textit{sem} \}
S → NP VP \{ \textit{NP}.\textit{sem}(\textit{VP}.\textit{sem}) \}
Creating Attachments

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

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

\[\text{Verb} \rightarrow \text{arrive} \quad \{\lambda y.\text{Arrived}(y)\}\]
\[\text{VP} \rightarrow \text{Verb} \quad \{\text{Verb.sem}\}\]
\[\text{S} \rightarrow \text{NP VP} \quad \{\text{NP.sem(VP.sem)}\}\]

\[\lambda Q.\forall x.[\text{Flight}(x) \implies Q(x)](\lambda y.\text{Arrived}(y))\]
Creating Attachments

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

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

\[\lambda P. \lambda Q. \forall x. [P(x) \implies Q(x)](\lambda x. Flight(x))\]
\[\lambda P. \lambda Q. \forall x. [P(x) \implies Q(x)](\lambda y. Flight(y))\] (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 \[\{\text{Verb.sem}\}\]
S → NP VP \[\{\text{NP.sem(VP.sem)}\}\]

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

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

Noun → flight  \{\lambda x.\text{Flight}(x)\}\nNom → Noun  \{ \text{Noun.sem} \}  
Det → Every  \{\lambda P.\lambda Q.\forall x.[P(x) \Rightarrow Q(x)]\}\nNP → Det Nom  \{ \text{Det.sem(Nom.sem)} \}  

\lambda P.\lambda Q.\forall x.[P(x) \Rightarrow Q(x)](\lambda x.\text{Flight}(x))
\lambda P.\lambda Q.\forall x.[P(x) \Rightarrow Q(x)](\lambda y.\text{Flight}(y)) (alpha conversion)
\lambda Q.\forall x.\lambda y.\text{Flight}(y)(x) \Rightarrow Q(x)
\lambda Q.\forall x.\text{Flight}(x) \Rightarrow Q(x)

Verb → arrive  \{\lambda y.\text{Arrived}(y)\}\nVP → Verb  \{ \text{Verb.sem} \}  
S → NP VP  \{ \text{NP.sem(VP.sem)} \}  

\lambda Q.\forall x.\text{Flight}(x) \Rightarrow Q(x)(\lambda y.\text{Arrived}(y))
\forall x.\text{Flight}(x) \Rightarrow \lambda y.\text{Arrived}(y)(x]
\forall x.\text{Flight}(x) \Rightarrow \text{Arrived}(x)\]
Extending Attachments

\[ \text{ProperNoun} \rightarrow \text{UA223} \quad \{ \lambda x.x(\text{UA223}) \} \]
ProperNoun \rightarrow UA223 \quad \{\lambda x.x(UA223)\}
Extending Attachments

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

UA223

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

Arrived(UA223)
ProperNoun → 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**

$\{ \lambda x. x(\text{UA223}) \}$

$\text{UA223}$

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

$\text{Arrived}(\text{UA223})$

**Determiner**

**Det → a**

$\{ \lambda P. \lambda Q. \exists x. [P(x) \land Q(x)] \}$
Extending Attachments

ProperNoun → UA223
{λx.x(UA223)}
UA223

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

Arrived(UA223)

Determiner
Det → a
{λP.λQ.∃x.[P(x) ∧ Q(x)]}
λQ.∃x.[Flight(x) ∧ Q(x)]
a flight
Extending Attachments

ProperNoun → UA223  \{λx.x(UA223)\}
UA223

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

\textit{Arrived}(UA223)

Determiner
Det → a
\{λP.λQ.∃x.[P(x) ∧ Q(x)]\}
a flight
\{λQ.∃x.[\textit{Flight}(x) ∧ Q(x)]\}

Transitive Verb
Extending Attachments

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

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

Arrived(UA223)

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

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

Transitive Verb

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

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

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

\textit{Arrived}(UA223)

Determiner
Det → a
{\lambda P. \lambda Q. \exists x. [P(x) \land Q(x)]}
\lambda Q. \exists x. [\textit{Flight}(x) \land Q(x)]

Transitive Verb
VP → Verb NP
{Verb.sem(NP.sem)}
Verb → booked
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

\[ \{ \lambda P . \lambda Q . \exists x . [ P(x) \land Q(x) ] \} \]

\[ \lambda Q . \exists x . [ Flight(x) \land Q(x) ] \]

Transitive Verb

VP → Verb NP

\{ Verb.sem(NP.sem) \}

Verb → booked

\[ \lambda W . \lambda z . W ( \lambda x . Booked(z, x) ) \]
Lambda Abstraction: Types (cont.)

\[ \forall x. [\text{Person}(x) \rightarrow \text{Loves}(x, m)] \]

\[ \lambda P. \forall x. [\text{Person}(x) \rightarrow P(x)] \]

\[ \langle \langle e, t \rangle, t \rangle \lambda P. \forall x. [\text{Person}(x) \rightarrow P(x)] \]

\[ \langle e, t \rangle \lambda P. \forall x. [\text{Person}(x) \rightarrow P(x)] \]

\[ \langle e, \langle e, t \rangle \rangle \lambda x. \lambda y. [\text{Loves}(y, x)] \]

\[ \langle e, \langle e, t \rangle \rangle \lambda x. \lambda y. [\text{Loves}(y, x)] \]

\[ \langle e, e \rangle \lambda y. [\text{Loves}(y, m)] \]

\[ \langle e, e \rangle \lambda y. [\text{Loves}(y, m)] \]

\[ \text{everybody} \]

\[ \text{loves} \]

\[ \text{mary} \]
Lambda Abstraction: Types (cont.)

\[
S \forall x. \left[ \text{Person}(x) \rightarrow \text{Loves}(x, m) \right]
\]

\[
\langle e, t \rangle \lambda P. \forall x. \left[ \text{Person}(x) \rightarrow P(x) \right]
\]

everybody

\[
\langle e, \langle e, t \rangle \rangle \lambda x. \lambda y. \left[ \text{Loves}(y, m) \right]
\]

loves mary

\[
\text{loves}
\]

\[
\text{everybody}
\]
Lambda Abstraction: Types (cont.)

\[ \forall x. \left[ \text{Person}(x) \rightarrow \text{Loves}(x, m) \right] \]

NP \langle \langle e, t \rangle, t \rangle \lambda P. \forall x. \left[ \text{Person}(x) \rightarrow P(x) \right]$

everybody

\[ \forall y. \left[ \text{Loves}(y, m) \right] \]

V \langle e, \langle e, t \rangle \rangle \lambda x. \lambda y. \left[ \text{Loves}(y, x) \right]$

loves

NP e m

mary

\{ o.petukhova; nddascalu }@lsv.uni-saarland.de

Introduction to Formal Semantics, Summer 2022
Lambda Abstraction: Types (cont.)

\[ \forall x. \left[ \text{Person}(x) \rightarrow \text{Loves}(x, m) \right] \]

\[ \text{NP} \langle \langle e, t \rangle, t \rangle \lambda P. \forall x. \left[ \text{Person}(x) \rightarrow P(x) \right] \]

\[ \text{VP} \langle e, t \rangle \lambda y. \left[ \text{Loves}(y, m) \right] \]

\[ \langle e, \langle e, t \rangle \rangle \lambda x. \lambda y. \left[ \text{Loves}(y, x) \right] \]

\[ \text{loves} \quad \text{NP} \quad \text{mary} \]
\[ \lambda x.\lambda y.[\text{Loves}(y, x)] \]

\[ \langle e, \langle e, t \rangle \rangle \quad e \]

\[ \langle e, t \rangle \]

\[ \lambda y.[\text{Loves}(y, m)] \]

\[ \langle e, t \rangle \quad t \]

\[ \lambda x.\lambda y.[\text{Loves}(y, x)] \quad m \]

\[ \langle e, \langle e, t \rangle \rangle \]

\[ \text{everybody} \]

\[ \text{loves mary} \]
Lambda Abstraction: Types (cont.)

\[ \lambda P. \forall x. [\text{Person}(x) \rightarrow P(x)] \]

\[ \lambda y. [\text{Loves}(y, m)] \]

\[ \lambda x. \lambda y. [\text{Loves}(y, x)] \]

\[ \text{everybody} \]

\[ \text{loves} \]

\[ \text{mary} \]
Lambda Abstraction: Types (cont.)

\[ \forall x. [\text{Person}(x) \rightarrow \text{Loves}(x, m)] \]

\[
\begin{array}{c}
\text{S} \\
\text{t} \\
\forall x. [\text{Person}(x) \rightarrow \text{Loves}(x, m)] \\
\text{NP} \\
\langle \langle e, t \rangle, t \rangle \\
\lambda P. \forall x. [\text{Person}(x) \rightarrow P(x)] \\
\text{everybody} \\
\text{VP} \\
\langle e, t \rangle \\
\lambda y. [\text{Loves}(y, m)] \\
\text{NP} \\
\langle e, \langle e, t \rangle \rangle \\
\lambda x. \lambda y. [\text{Loves}(y, x)] \\
\text{loves} \\
\text{NP} \\
\langle e, t \rangle \\
\lambda y. [\text{Loves}(y, m)] \\
\text{mary} \\
\end{array}
\]
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 → john

{\lambda x.x(john)}
One more example

John booked a flight

ProperNoun $\rightarrow$ john

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

(john)
One more example

John booked a flight

ProperNoun → john

a flight

\{\lambda x. x(john)\}

(john)

\lambda Q. \exists y. [Flight(y) \land Q(y)]
One more example

John booked a flight

ProperNoun → john

a flight
Verb → booked

\{\lambda x. x(john)\}

(john)

\lambda Q. \exists y. [\text{Flight}(y) \land Q(y)]
One more example

John booked a flight

ProperNoun → john

a flight

Verb → booked

\[ \lambda x. x (john) \]

\( (john) \)

\[ \lambda Q. \exists y. [\text{Flight}(y) \land Q(y)] \]

\[ \lambda W. \lambda z. W (\lambda x. \text{Booked}(z, x)) \]
One more example

John booked a flight

ProperNoun → john

a flight
Verb → booked
VP → Verb NP

{\lambda x . x(john)}
(john)
\lambda Q. \exists y. [\text{Flight}(y) \land Q(y)]
\lambda W . \lambda z . W(\lambda x. \text{Booked}(z, x))
{\text{Verb.sem}(\text{NP.sem})}
One more example

John booked a flight

ProperNoun → john
(a flight)
Verb → booked
VP → Verb NP

\[
\lambda W. \lambda z. W(\lambda x. Booked(z, x))(\lambda Q. \exists y. [Flight(y) \land Q(y)])
\]
One more example

John booked a flight

ProperNoun → john
a flight
Verb → booked
VP → Verb NP

\[ \lambda W. \lambda z. W(\lambda x. Booked(z, x))(\lambda Q. \exists y. [Flight(y) \land Q(y)]) \]
\[ \lambda z. \lambda Q. \exists y. [Flight(y) \land Q(y)](\lambda x. Booked(z, x)) \]
One more example

John booked a flight

ProperNoun → john

a flight

Verb → booked

VP → Verb NP

\[\lambda W. \lambda z. W(\lambda x. \text{Booked}(z, x))(\lambda Q. \exists y. [\text{Flight}(y) \land Q(y)])\]

\[\lambda z. \lambda Q. \exists y. [\text{Flight}(y) \land Q(y)](\lambda x. \text{Booked}(z, x))\]

\[\lambda z. \exists y. [\text{Flight}(y) \land \lambda x. \text{Booked}(z, x)(y)]\]
One more example

John booked a flight

ProperNoun → john

a flight

Verb → booked

VP → Verb NP

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

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

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

\[ \lambda z. \exists y. [Flight(y) \land Booked(z, y)] \]
One more example

John booked a flight

ProperNoun → john

a flight

Verb → booked

VP → Verb NP

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

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

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

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

S → NP VP

\(\{NP.sem(VP.sem)\}\)
One more example

John booked a flight

ProperNoun → john

a flight

Verb → booked

VP → Verb NP

λW.λz.W(λx.Booked(z, x))(λQ.∃y.[Flight(y) ∧ Q(y)])

λz.λQ.∃y.[Flight(y) ∧ Q(y)](λx.Booked(z, x))

λz.∃y.[Flight(y) ∧ λx.Booked(z, x)(y)]

S → NP VP

λz.∃y.[Flight(y) ∧ Booked(z, y)](john)
One more example

John booked a flight

ProperNoun → john

a flight

Verb → booked

VP → Verb NP

S → NP VP

\[
\begin{align*}
\lambda W.\lambda z. W(\lambda x. Booked(z, x))(\lambda Q.\exists y.[Flight(y) \land Q(y)]) \\
\lambda z.\lambda Q.\exists y.[Flight(y) \land Q(y)](\lambda x. Booked(z, x)) \\
\lambda z.\exists y.[Flight(y) \land \lambda x. Booked(z, x)(y)] \\
\lambda z.\exists y.[Flight(y) \land Booked(z, y)] \\
\end{align*}
\]

\[
\begin{align*}
\lambda z.\exists y.[Flight(y) \land Booked(z, y)](john) \\
\exists y.[Flight(y) \land Booked(john, y)] \\
\end{align*}
\]
Quizz for Today

TBA