9.19 Computational Psycholinguistics

Basic semantics

November 14, 2023

Table of contents

Introduction, and some technical background

From syntax to semantics

- In previous classes we have seen how to build syntax trees from strings of words.
    - The trees aimed to capture notions such as **constituency** (e.g. the fact that a transitive verb forms a "chunk" with its object, but not with its subject), and **thematic roles** assigned by a verb to its arguments (e.g. the OBJECT, THEME, GOAL...).
    - We also saw that some trees were **structurally ambiguous** (*John saw the girl with binoculars*).
- Now, we'd like to define a way to systematically compute the logical meaning of a given sentence, given its syntax tree. In other words, we'd like to define a **mapping between trees and first-order logic**.
- One goal of semanticists is to capture various kinds of **semantic ambiguities**, sometimes called "readings" of a sentence.

From syntax to semantics

- In previous classes we have seen how to build syntax trees from strings of words.
  - The trees aimed to capture notions such as **constituency** (e.g. the fact that a transitive verb forms a "chunk" with its object, but not with its subject), and **thematic roles** assigned by a verb to its arguments (e.g. the OBJECT, THEME, GOAL...).
  - We also saw that some trees were **structurally ambiguous** (*John saw the girl with binoculars*).
- Now, we'd like to define a way to systematically compute the logical meaning of a given sentence, given its syntax tree. In other words, we'd like to define a **mapping between trees and first-order logic**.
- One goal of semanticists is to capture various kinds of **semantic ambiguities**, sometimes called "readings" of a sentence.

From syntax to semantics

- In previous classes we have seen how to build syntax trees from strings of words.
  - The trees aimed to capture notions such as **constituency** (e.g. the fact that a transitive verb forms a "chunk" with its object, but not with its subject), and **thematic roles** assigned by a verb to its arguments (e.g. the OBJECT, THEME, GOAL...).
  - We also saw that some trees were **structurally ambiguous** (*John saw the girl with binoculars*).
- Now, we'd like to define a way to systematically compute the logical meaning of a given sentence, given its syntax tree. In other words, we'd like to define a **mapping between trees and first-order logic**.
- One goal of semanticists is to capture various kinds of **semantic ambiguities**, sometimes called "readings" of a sentence.

From syntax to semantics

- In previous classes we have seen how to build syntax trees from strings of words.
  - The trees aimed to capture notions such as **constituency** (e.g. the fact that a transitive verb forms a "chunk" with its object, but not with its subject), and **thematic roles** assigned by a verb to its arguments (e.g. the OBJECT, THEME, GOAL...).
  - We also saw that some trees were **structurally ambiguous** (*John saw the girl with binoculars*).
- Now, we'd like to define a way to systematically compute the logical meaning of a given sentence, given its syntax tree. In other words, we'd like to define a **mapping between trees and first-order logic**.
- One goal of semanticists is to capture various kinds of **semantic ambiguities**, sometimes called "readings" of a sentence.

From syntax to semantics

- In previous classes we have seen how to build syntax trees from strings of words.
  - The trees aimed to capture notions such as **constituency** (e.g. the fact that a transitive verb forms a "chunk" with its object, but not with its subject), and **thematic roles** assigned by a verb to its arguments (e.g. the OBJECT, THEME, GOAL...).
  - We also saw that some trees were **structurally ambiguous** (*John saw the girl with binoculars*).
- Now, we'd like to define a way to systematically compute the logical meaning of a given sentence, given its syntax tree. In other words, we'd like to define a **mapping between trees and first-order logic**.
- One goal of semanticists is to capture various kinds of **semantic ambiguities**, sometimes called "readings" of a sentence.

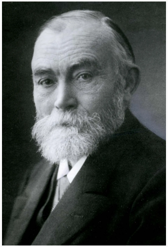
The Principle of Compositionality

- To devise a consistent mapping between syntax and semantics, we exploit the following idea which dates back (at least) from **Gottlob Frege** (1884):

Principle of Compositionality

the meaning (=denotation) of a complex expression is determined by its **structure** and **the meanings of its constituents**.

- This idea was revived in 1960's by **Richard Montague**.
- Montague's thesis was that natural languages and formal languages (in particular programming languages) can be treated in the same way.



That's Frege

And that's Montague

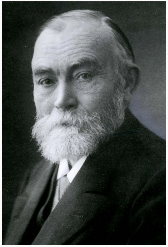
The Principle of Compositionality

- To devise a consistent mapping between syntax and semantics, we exploit the following idea which dates back (at least) from **Gottlob Frege** (1884):

Principle of Compositionality

the meaning (=denotation) of a complex expression is determined by its **structure** and **the meanings of its constituents**.

- This idea was revived in 1960's by **Richard Montague**.
- Montague's thesis was that natural languages and formal languages (in particular programming languages) can be treated in the same way.



That's Frege

And that's Montague

Is language really compositional?

- Can you think of any counterexamples to compositionality?

Is language really compositional?

- Can you think of any counterexamples to compositionality?
- **Idiom chunks** (e.g. *kick the bucket*), **logical metonymy** (e.g. *John began the book*) [Pustejovsky, 1995]
- Context-sensitive elements, such as **indexicals** (*I*, *you*, *here*, *now*...), **gradable adjectives** (e.g. *tall*), **subjective predicates** (e.g. *yummy*).
- Proper names vs. definite descriptions under **belief verbs**:

(2) *Context: Ralph saw Ortcutt at the beach and believes the man he saw is a spy. But Ralph did not realize the man was actually Ortcutt.*
Ralph **believes** the man at the beach is a spy.
$\not\rightsquigarrow$ Ralph believes Ortcutt is a spy [Quine, 1956].

Is language really compositional?

- Can you think of any counterexamples to compositionality?
- **Idiom chunks** (e.g. *kick the bucket*), **logical metonymy** (e.g. *John began the book*) [Pustejovsky, 1995]
- Context-sensitive elements, such as **indexicals** (*I*, *you*, *here*, *now*...), **gradable adjectives** (e.g. *tall*), **subjective predicates** (e.g. *yummy*).
- Proper names vs. definite descriptions under **belief verbs**:

(3) *Context: Ralph saw Ortcutt at the beach and believes the man he saw is a spy. But Ralph did not realize the man was actually Ortcutt.*
Ralph **believes** the man at the beach is a spy.
$\not\rightsquigarrow$ Ralph believes Ortcutt is a spy [Quine, 1956].

Is language really compositional?

- Can you think of any counterexamples to compositionality?
- **Idiom chunks** (e.g. *kick the bucket*), **logical metonymy** (e.g. *John began the book*) [Pustejovsky, 1995]
- Context-sensitive elements, such as **indexicals** (*I*, *you*, *here*, *now*...), **gradable adjectives** (e.g. *tall*), **subjective predicates** (e.g. *yummy*).
- Proper names vs. definite descriptions under **belief verbs**:

(4) *Context: Ralph saw Ortcutt at the beach and believes the man he saw is a spy. But Ralph did not realize the man was actually Ortcutt.*
Ralph **believes** the man at the beach is a spy.
$\not\rightsquigarrow$ Ralph believes Ortcutt is a spy [Quine, 1956].

Is language really compositional? continued

- **Bracketing paradoxes**: *unhappier* is parsed [un-[happi-er]] (-er cannot attach to a 2-syllable adjective!), yet means *more unhappy* [Allen, 1978].
- **Weakened/strengthened** modals/logical operators:

(5) Minimal Sufficiency readings [von Fintel and Iatridou, 2007]:
To get good cheese you **only have to** go to the North End.
$\rightsquigarrow$ You **don't have to** go to the North End (but it's the easisest option).

(6) "Free choice" inferences [Kamp, 1973]:
You may have cake **or** ice-cream.
$\rightsquigarrow$ You may have cake **and** you may have ice-cream.

Is language really compositional? continued

- **Bracketing paradoxes**: *unhappier* is parsed [un-[happi-er]] (-er cannot attach to a 2-syllable adjective!), yet means *more unhappy* [Allen, 1978].
- **Weakened/strengthened** modals/logical operators:

(7) Minimal Sufficiency readings [von Fintel and Iatridou, 2007]:
To get good cheese you **only have to** go to the North End.
$\rightsquigarrow$ You **don't have to** go to the North End (but it's the easisest option).

(8) "Free choice" inferences [Kamp, 1973]:
You may have cake **or** ice-cream.
$\rightsquigarrow$ You may have cake **and** you may have ice-cream.

A detour through λ-calculus and types

- To compose meanings together, we will need functions.
- λ-calculus can be seen as a compact way of writing and applying functions. λ-terms can take 3 forms (inductive definition):
  - a **variable** x;
  - a **function** $(\lambda x.\ M)$ where x is a bound variable and M is a term;
  - an **application** $M(N)$ where both M and N are terms.
- If x has type α (written "x : α") and M type β, then the term $(\lambda x.\ M)$ has type $\alpha \to \beta$. It's a function which, given an $x : \alpha$, returns a term $M : \beta$ that usually depends on x. Note that both x and M can be functions themselves.
- Lambda-terms terms can be "reduced" using the following operation (assuming the types are right):

$$(\lambda x.\ M)(y) = M[y/x]$$

- Meaning: applying the function $(\lambda x.\ M)$ to an input y amounts to substituting any occurrence of the bound variable x in M by the input y.

A detour through λ-calculus and types

- To compose meanings together, we will need functions.
- λ-calculus can be seen as a compact way of writing and applying functions. λ-terms can take 3 forms (inductive definition):
    - a **variable** x;
    - a **function** $(\lambda x.\ M)$ where x is a bound variable and M is a term;
    - an **application** $M(N)$ where both M and N are terms.
- If x has type α (written "x : α") and M type β, then the term $(\lambda x.\ M)$ has type $\alpha \to \beta$. It's a function which, given an $x : \alpha$, returns a term $M : \beta$ that usually depends on x. Note that both x and M can be functions themselves.
- Lambda-terms terms can be "reduced" using the following operation (assuming the types are right):

$$(\lambda x.\ M)(y) = M[y/x]$$

- Meaning: applying the function $(\lambda x.\ M)$ to an input y amounts to substituting any occurrence of the bound variable x in M by the input y.

A detour through λ-calculus and types

- To compose meanings together, we will need functions.
- λ-calculus can be seen as a compact way of writing and applying functions. λ-terms can take 3 forms (inductive definition):
  - a **variable** x;
  - a **function** $(\lambda x.\ M)$ where x is a bound variable and M is a term;
  - an **application** $M(N)$ where both M and N are terms.
- If x has type α (written "x : α") and M type β, then the term $(\lambda x.\ M)$ has type $\alpha \to \beta$. It's a function which, given an $x : \alpha$, returns a term $M : \beta$ that usually depends on x. Note that both x and M can be functions themselves.
- Lambda-terms terms can be "reduced" using the following operation (assuming the types are right):

$$(\lambda x.\ M)(y) = M[y/x]$$

- Meaning: applying the function $(\lambda x.\ M)$ to an input y amounts to substituting any occurrence of the bound variable x in M by the input y.

A detour through λ-calculus and types

- To compose meanings together, we will need functions.
- λ-calculus can be seen as a compact way of writing and applying functions. λ-terms can take 3 forms (inductive definition):
  - a **variable** x;
  - a **function** $(\lambda x.\ M)$ where x is a bound variable and M is a term;
  - an **application** $M(N)$ where both M and N are terms.
- If x has type α (written "x : α") and M type β, then the term $(\lambda x.\ M)$ has type $\alpha \to \beta$. It's a function which, given an $x : \alpha$, returns a term $M : \beta$ that usually depends on x. Note that both x and M can be functions themselves.
- Lambda-terms terms can be "reduced" using the following operation (assuming the types are right):

$$(\lambda x.\ M)(y) = M[y/x]$$

- Meaning: applying the function $(\lambda x.\ M)$ to an input y amounts to substituting any occurrence of the bound variable x in M by the input y.

A detour through λ-calculus and types

- To compose meanings together, we will need functions.
- λ-calculus can be seen as a compact way of writing and applying functions. λ-terms can take 3 forms (inductive definition):
  - a **variable** x;
  - a **function** $(\lambda x.\ M)$ where x is a bound variable and M is a term;
  - an **application** $M(N)$ where both M and N are terms.
- If x has type α (written "x : α") and M type β, then the term $(\lambda x.\ M)$ has type $\alpha \to \beta$. It's a function which, given an $x : \alpha$, returns a term $M : \beta$ that usually depends on x. Note that both x and M can be functions themselves.
- Lambda-terms terms can be "reduced" using the following operation (assuming the types are right):

$$(\lambda x.\ M)(y) = M[y/x]$$

- Meaning: applying the function $(\lambda x.\ M)$ to an input y amounts to substituting any occurrence of the bound variable x in M by the input y.

A detour through λ-calculus and types

- To compose meanings together, we will need functions.
- λ-calculus can be seen as a compact way of writing and applying functions. λ-terms can take 3 forms (inductive definition):
  - a **variable** x;
  - a **function** $(\lambda x.\ M)$ where x is a bound variable and M is a term;
  - an **application** $M(N)$ where both M and N are terms.
- If x has type α (written "x : α") and M type β, then the term $(\lambda x.\ M)$ has type $\alpha \to \beta$. It's a function which, given an $x : \alpha$, returns a term $M : \beta$ that usually depends on x. Note that both x and M can be functions themselves.
- Lambda-terms terms can be "reduced" using the following operation (assuming the types are right):

$$(\lambda x.\ M)(y) = M[y/x]$$

- Meaning: applying the function $(\lambda x.\ M)$ to an input y amounts to substituting any occurrence of the bound variable x in M by the input y.

A detour through λ-calculus and types

- To compose meanings together, we will need functions.
- λ-calculus can be seen as a compact way of writing and applying functions. λ-terms can take 3 forms (inductive definition):
  - a **variable** x;
  - a **function** $(\lambda x.\ M)$ where x is a bound variable and M is a term;
  - an **application** $M(N)$ where both M and N are terms.
- If x has type α (written "x : α") and M type β, then the term $(\lambda x.\ M)$ has type $\alpha \to \beta$. It's a function which, given an $x : \alpha$, returns a term $M : \beta$ that usually depends on x. Note that both x and M can be functions themselves.
- Lambda-terms terms can be "reduced" using the following operation (assuming the types are right):

$$(\lambda x.\ M)(y) = M[y/x]$$

- Meaning: applying the function $(\lambda x.\ M)$ to an input y amounts to substituting any occurrence of the bound variable x in M by the input y.

A few examples of λ-terms

- The "add 10" function (=partial application of the "sum" function):

$$(\lambda x.\ \lambda y.\ x + y)(10) = (\lambda y.\ x + y)[10/x] = (\lambda y.\ 10 + y)$$

- Adding 10 to 5 (=total application of the sum function):

$$(\lambda y.\ 10 + y)(5) = (10 + y)[5/y] = 10 + 5 = 15$$

- Testing if 10 is prime (a Boolean function):

$$(\lambda x.\ \mathbf{isprime}(x))(10) = (\mathbf{isprime}(x))[10/x] = \mathbf{isprime}(10) = \bot$$

- Negating the "prime" function (notice that we renamed the bound variable in the input term into "y" to avoid variable capture):

$$\begin{aligned}(\lambda P.\ \lambda x.\ \neg P(x))(\lambda x.\ \mathbf{isprime}(x)) &= (\lambda x.\ \neg P(x))[(\lambda y.\ \mathbf{isprime}(y))/P] \\ &= (\lambda x.\ \neg(\lambda y.\ \mathbf{isprime}(y))(x)) \\ &= (\lambda x.\ \neg\mathbf{isprime}(y)[x/y])\end{aligned}$$

A few examples of λ-terms

- The "add 10" function (=partial application of the "sum" function):

$$(\lambda x.\ \lambda y.\ x + y)(10) = (\lambda y.\ x + y)[10/x] = (\lambda y.\ 10 + y)$$

- Adding 10 to 5 (=total application of the sum function):

$$(\lambda y.\ 10 + y)(5) = (10 + y)[5/y] = 10 + 5 = 15$$

- Testing if 10 is prime (a Boolean function):

$$(\lambda x.\ \mathbf{isprime}(x))(10) = (\mathbf{isprime}(x))[10/x] = \mathbf{isprime}(10) = \bot$$

- Negating the "prime" function (notice that we renamed the bound variable in the input term into "y" to avoid variable capture):

$$\begin{aligned}(\lambda P.\ \lambda x.\ \neg P(x))(\lambda x.\ \mathbf{isprime}(x)) &= (\lambda x.\ \neg P(x))[(\lambda y.\ \mathbf{isprime}(y))/P] \\ &= (\lambda x.\ \neg(\lambda y.\ \mathbf{isprime}(y))(x)) \\ &= (\lambda x.\ \neg\mathbf{isprime}(y)[x/y])\end{aligned}$$

A few examples of λ-terms

- The "add 10" function (=partial application of the "sum" function):

$$(\lambda x.\ \lambda y.\ x + y)(10) = (\lambda y.\ x + y)[10/x] = (\lambda y.\ 10 + y)$$

- Adding 10 to 5 (=total application of the sum function):

$$(\lambda y.\ 10 + y)(5) = (10 + y)[5/y] = 10 + 5 = 15$$

- Testing if 10 is prime (a Boolean function):

$$(\lambda x.\ \mathbf{isprime}(x))(10) = (\mathbf{isprime}(x))[10/x] = \mathbf{isprime}(10) = \bot$$

- Negating the "prime" function (notice that we renamed the bound variable in the input term into "y" to avoid variable capture):

$$\begin{aligned}(\lambda P.\ \lambda x.\ \neg P(x))(\lambda x.\ \mathbf{isprime}(x)) &= (\lambda x.\ \neg P(x))[(\lambda y.\ \mathbf{isprime}(y))/P] \\ &= (\lambda x.\ \neg(\lambda y.\ \mathbf{isprime}(y))(x)) \\ &= (\lambda x.\ \neg\mathbf{isprime}(y)[x/y])\end{aligned}$$

A few examples of λ-terms

- The "add 10" function (=partial application of the "sum" function):

$$(\lambda x.\ \lambda y.\ x + y)(10) = (\lambda y.\ x + y)[10/x] = (\lambda y.\ 10 + y)$$

- Adding 10 to 5 (=total application of the sum function):

$$(\lambda y.\ 10 + y)(5) = (10 + y)[5/y] = 10 + 5 = 15$$

- Testing if 10 is prime (a Boolean function):

$$(\lambda x.\ \mathbf{isprime}(x))(10) = (\mathbf{isprime}(x))[10/x] = \mathbf{isprime}(10) = \bot$$

- Negating the "prime" function (notice that we renamed the bound variable in the input term into "y" to avoid variable capture):

$$\begin{aligned}(\lambda P.\ \lambda x.\ \neg P(x))(\lambda x.\ \mathbf{isprime}(x)) &= (\lambda x.\ \neg P(x))[(\lambda y.\ \mathbf{isprime}(y))/P] \\ &= (\lambda x.\ \neg(\lambda y.\ \mathbf{isprime}(y))(x)) \\ &= (\lambda x.\ \neg \mathbf{isprime}(y)[x/y])\end{aligned}$$

Deriving the meaning of simple sentences

Denotation of sentences

- We assume that whole sentences are defined by the conditions under which they are true (**truth conditions**).
- Note that this is slightly different from a simple Boolean value (0 or 1). For instance, the meaning of *a cat is on the mat* is not always 0 or 1; rather, it will evaluate to 1 **iff** there exists something that's a cat that is located on the unique salient mat; and 0 otherwise.
- We call the type of sentences (i.e. elements with truth conditions) $\mathtt{t}$. We use the double-bracket notation ($[\![.]\!]$) to indicate the meaning (=denotation) of a given string.

$$[\![\textit{a cat is on the}^1 \textit{ mat}]\!] = \begin{cases} 1 & \text{if } \exists x.\ \mathbf{cat}(x) \wedge \exists! y.\ \mathbf{mat}(y) \wedge \mathbf{on}(x)(y) \\ 0 & \text{otherwise} \end{cases}$$

[1]The meaning of *the* is more complex than what we do here: *the* should *presuppose* the existence and the uniqueness of a mat. This is just to get a rough idea of a possible denotation of the sentence.

Denotation of sentences

- We assume that whole sentences are defined by the conditions under which they are true (**truth conditions**).
- Note that this is slightly different from a simple Boolean value (0 or 1). For instance, the meaning of *a cat is on the mat* is not always 0 or 1; rather, it will evaluate to 1 **iff** there exists something that's a cat that is located on the unique salient mat; and 0 otherwise.
- We call the type of sentences (i.e. elements with truth conditions) t. We use the double-bracket notation ($[\![.]\!]$) to indicate the meaning (=denotation) of a given string.

$$[\![\textit{a cat is on the}^{1} \textit{ mat}]\!] = \begin{cases} 1 & \text{if } \exists x.\ \mathbf{cat}(x) \wedge \exists! y.\ \mathbf{mat}(y) \wedge\ \mathbf{on}(x)(y) \\ 0 & \text{otherwise} \end{cases}$$

[1] The meaning of *the* is more complex than what we do here: *the* should *presuppose* the existence and the uniqueness of a mat. This is just to get a rough idea of a possible denotation of the sentence.

Denotation of sentences

- We assume that whole sentences are defined by the conditions under which they are true (**truth conditions**).
- Note that this is slightly different from a simple Boolean value (0 or 1). For instance, the meaning of *a cat is on the mat* is not always 0 or 1; rather, it will evaluate to 1 **iff** there exists something that's a cat that is located on the unique salient mat; and 0 otherwise.
- We call the type of sentences (i.e. elements with truth conditions) `t`. We use the double-bracket notation ($[\![.]\!]$) to indicate the meaning (=denotation) of a given string.

$$[\![\textit{a cat is on the}^{1}\ \textit{mat}]\!] = \begin{cases} 1 & \text{if } \exists x.\ \textbf{cat}(x) \wedge \exists! y.\ \textbf{mat}(y) \wedge\ \textbf{on}(x)(y) \\ 0 & \text{otherwise} \end{cases}$$

[1] The meaning of *the* is more complex than what we do here: *the* should *presuppose* the existence and the uniqueness of a mat. This is just to get a rough idea of a possible denotation of the sentence.

Denotation of sentences

- We assume that whole sentences are defined by the conditions under which they are true (**truth conditions**).
- Note that this is slightly different from a simple Boolean value (0 or 1). For instance, the meaning of *a cat is on the mat* is not always 0 or 1; rather, it will evaluate to 1 **iff** there exists something that's a cat that is located on the unique salient mat; and 0 otherwise.
- We call the type of sentences (i.e. elements with truth conditions) t. We use the double-bracket notation ($[\![.]\!]$) to indicate the meaning (=denotation) of a given string.

$$[\![\textit{a cat is on the}^{1}\ \textit{mat}]\!] = \begin{cases} 1 & \text{if } \exists x.\ \mathbf{cat}(x) \wedge \exists! y.\ \mathbf{mat}(y) \wedge\ \mathbf{on}(x)(y) \\ 0 & \text{otherwise} \end{cases}$$

[1] The meaning of *the* is more complex than what we do here: *the* should *presuppose* the existence and the uniqueness of a mat. This is just to get a rough idea of a possible denotation of the sentence.

Denotation of terminals

- How to properly *compute* the meaning of sentences like *a cat is on the mat*? Principle of Compositionality: **from the meaning of the terminals** and how they merge in the tree.
- We assume that each terminal of the tree can be mapped to a lexical "meaning". For instance:
  - **Proper names** refer to fixed entities ($\sim$ constants) belonging to a certain domain D. We call $\mathtt{e}$ the type of entities.
  - **Predicates** (*happy*, *teacher*...) or verbs (*like*, *jump*...) are functions mapping one or more entities (type $\mathtt{e}$) to truth values (type $\mathtt{t}$).

$$\begin{aligned} [\![\textit{happy}]\!] &: \mathtt{e} \to \mathtt{t} & [\![\textit{happy}]\!] &= \lambda x.\ \mathbf{happy}(x) \\ [\![\textit{teacher}]\!] &: \mathtt{e} \to \mathtt{t} & [\![\textit{teacher}]\!] &= \lambda x.\ \mathbf{teacher}(x) \\ [\![\textit{like}]\!] &: \mathtt{e} \to (\mathtt{e} \to \mathtt{t}) & [\![\textit{like}]\!] &= \lambda x.\ \lambda y.\ \mathbf{like}(x)(y) \\ [\![\textit{jump}]\!] &: \mathtt{e} \to \mathtt{t} & [\![\textit{jump}]\!] &= \lambda x.\ \mathbf{jump}(x) \end{aligned}$$

  - Some special terminals ("traces"/pronouns) may denote bound variables or type $\mathtt{e}$.
  - We keep determiners for later.
- Now let's try to combine all those things together!

Denotation of terminals

- How to properly *compute* the meaning of sentences like *a cat is on the mat*? Principle of Compositionality: **from the meaning of the terminals** and how they merge in the tree.
- We assume that each terminal of the tree can be mapped to a lexical "meaning". For instance:
  - **Proper names** refer to fixed entities ($\sim$ constants) belonging to a certain domain D. We call $\mathtt{e}$ the type of entities.
  - **Predicates** (*happy*, *teacher*...) or verbs (*like*, *jump*...) are functions mapping one or more entities (type $\mathtt{e}$) to truth values (type $\mathtt{t}$).

$$
\begin{aligned}
[\![\textit{happy}]\!] &: \mathtt{e} \to \mathtt{t} & [\![\textit{happy}]\!] &= \lambda x.\ \textbf{happy}(x) \\
[\![\textit{teacher}]\!] &: \mathtt{e} \to \mathtt{t} & [\![\textit{teacher}]\!] &= \lambda x.\ \textbf{teacher}(x) \\
[\![\textit{like}]\!] &: \mathtt{e} \to (\mathtt{e} \to \mathtt{t}) & [\![\textit{like}]\!] &= \lambda x.\ \lambda y.\ \textbf{like}(x)(y) \\
[\![\textit{jump}]\!] &: \mathtt{e} \to \mathtt{t} & [\![\textit{jump}]\!] &= \lambda x.\ \textbf{jump}(x)
\end{aligned}
$$

  - Some special terminals ("traces"/pronouns) may denote bound variables or type $\mathtt{e}$.
  - We keep determiners for later.
- Now let's try to combine all those things together!

Denotation of terminals

- How to properly *compute* the meaning of sentences like *a cat is on the mat*? Principle of Compositionality: **from the meaning of the terminals** and how they merge in the tree.
- We assume that each terminal of the tree can be mapped to a lexical "meaning". For instance:
  - **Proper names** refer to fixed entities ($\sim$ constants) belonging to a certain domain D. We call $\mathtt{e}$ the type of entities.
  - **Predicates** (*happy*, *teacher*...) or verbs (*like*, *jump*...) are functions mapping one or more entities (type $\mathtt{e}$) to truth values (type $\mathtt{t}$).

$$\begin{array}{rll} [\![\mathit{happy}]\!] : \mathtt{e} \to \mathtt{t} & [\![\mathit{happy}]\!] & = \lambda x.\ \mathbf{happy}(x) \\ [\![\mathit{teacher}]\!] : \mathtt{e} \to \mathtt{t} & [\![\mathit{teacher}]\!] & = \lambda x.\ \mathbf{teacher}(x) \\ [\![\mathit{like}]\!] : \mathtt{e} \to (\mathtt{e} \to \mathtt{t}) & [\![\mathit{like}]\!] & = \lambda x.\ \lambda y.\ \mathbf{like}(x)(y) \\ [\![\mathit{jump}]\!] : \mathtt{e} \to \mathtt{t} & [\![\mathit{jump}]\!] & = \lambda x.\ \mathbf{jump}(x) \end{array}$$

  - Some special terminals ("traces"/pronouns) may denote bound variables or type $\mathtt{e}$.
  - We keep determiners for later.
- Now let's try to combine all those things together!

Denotation of terminals

- How to properly *compute* the meaning of sentences like *a cat is on the mat*? Principle of Compositionality: **from the meaning of the terminals** and how they merge in the tree.
- We assume that each terminal of the tree can be mapped to a lexical "meaning". For instance:
  - **Proper names** refer to fixed entities ($\sim$ constants) belonging to a certain domain D. We call $\mathtt{e}$ the type of entities.
  - **Predicates** (*happy*, *teacher*...) or verbs (*like*, *jump*...) are functions mapping one or more entities (type $\mathtt{e}$) to truth values (type $\mathtt{t}$).

$$
\begin{aligned}
[\![\textit{happy}]\!] &: \mathtt{e} \to \mathtt{t} & [\![\textit{happy}]\!] &= \lambda x.\ \textbf{happy}(x) \\
[\![\textit{teacher}]\!] &: \mathtt{e} \to \mathtt{t} & [\![\textit{teacher}]\!] &= \lambda x.\ \textbf{teacher}(x) \\
[\![\textit{like}]\!] &: \mathtt{e} \to (\mathtt{e} \to \mathtt{t}) & [\![\textit{like}]\!] &= \lambda x.\ \lambda y.\ \textbf{like}(x)(y) \\
[\![\textit{jump}]\!] &: \mathtt{e} \to \mathtt{t} & [\![\textit{jump}]\!] &= \lambda x.\ \textbf{jump}(x)
\end{aligned}
$$

  - Some special terminals ("traces"/pronouns) may denote bound variables or type $\mathtt{e}$.
  - We keep determiners for later.
- Now let's try to combine all those things together!

Denotation of terminals

- How to properly *compute* the meaning of sentences like *a cat is on the mat*? Principle of Compositionality: **from the meaning of the terminals** and how they merge in the tree.
- We assume that each terminal of the tree can be mapped to a lexical "meaning". For instance:
  - **Proper names** refer to fixed entities ($\sim$ constants) belonging to a certain domain D. We call $\mathtt{e}$ the type of entities.
  - **Predicates** (*happy*, *teacher*...) or verbs (*like*, *jump*...) are functions mapping one or more entities (type $\mathtt{e}$) to truth values (type $\mathtt{t}$).

$$\begin{aligned}
[\![\mathit{happy}]\!] &: \mathtt{e} \to \mathtt{t} & [\![\mathit{happy}]\!] &= \lambda x.\ \mathbf{happy}(x) \\
[\![\mathit{teacher}]\!] &: \mathtt{e} \to \mathtt{t} & [\![\mathit{teacher}]\!] &= \lambda x.\ \mathbf{teacher}(x) \\
[\![\mathit{like}]\!] &: \mathtt{e} \to (\mathtt{e} \to \mathtt{t}) & [\![\mathit{like}]\!] &= \lambda x.\ \lambda y.\ \mathbf{like}(x)(y) \\
[\![\mathit{jump}]\!] &: \mathtt{e} \to \mathtt{t} & [\![\mathit{jump}]\!] &= \lambda x.\ \mathbf{jump}(x)
\end{aligned}$$

  - Some special terminals ("traces"/pronouns) may denote bound variables or type $\mathtt{e}$.
  - We keep determiners for later.
- Now let's try to combine all those things together!

Denotation of terminals

- How to properly *compute* the meaning of sentences like *a cat is on the mat*? Principle of Compositionality: **from the meaning of the terminals** and how they merge in the tree.
- We assume that each terminal of the tree can be mapped to a lexical "meaning". For instance:
  - **Proper names** refer to fixed entities ($\sim$ constants) belonging to a certain domain D. We call $\mathtt{e}$ the type of entities.
  - **Predicates** (*happy*, *teacher*...) or verbs (*like*, *jump*...) are functions mapping one or more entities (type $\mathtt{e}$) to truth values (type $\mathtt{t}$).

$$
\begin{array}{rlll}
[\![\textit{happy}]\!] : \mathtt{e} \to \mathtt{t} & [\![\textit{happy}]\!] & = \lambda x.\ \textbf{happy}(x) \\
[\![\textit{teacher}]\!] : \mathtt{e} \to \mathtt{t} & [\![\textit{teacher}]\!] & = \lambda x.\ \textbf{teacher}(x) \\
[\![\textit{like}]\!] : \mathtt{e} \to (\mathtt{e} \to \mathtt{t}) & [\![\textit{like}]\!] & = \lambda x.\ \lambda y.\ \textbf{like}(x)(y) \\
[\![\textit{jump}]\!] : \mathtt{e} \to \mathtt{t} & [\![\textit{jump}]\!] & = \lambda x.\ \textbf{jump}(x)
\end{array}
$$

  - Some special terminals ("traces"/pronouns) may denote bound variables or type $\mathtt{e}$.
  - We keep determiners for later.
- Now let's try to combine all those things together!

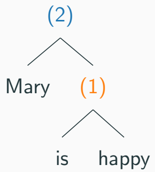
Denotation of terminals

- How to properly *compute* the meaning of sentences like *a cat is on the mat*? Principle of Compositionality: **from the meaning of the terminals** and how they merge in the tree.
- We assume that each terminal of the tree can be mapped to a lexical "meaning". For instance:
  - **Proper names** refer to fixed entities ($\sim$ constants) belonging to a certain domain D. We call $\mathtt{e}$ the type of entities.
  - **Predicates** (*happy*, *teacher*...) or verbs (*like*, *jump*...) are functions mapping one or more entities (type $\mathtt{e}$) to truth values (type $\mathtt{t}$).

$$
\begin{array}{rl}
[\![\mathit{happy}]\!] : \mathtt{e} \to \mathtt{t} & [\![\mathit{happy}]\!] = \lambda x.\ \mathbf{happy}(x) \\
[\![\mathit{teacher}]\!] : \mathtt{e} \to \mathtt{t} & [\![\mathit{teacher}]\!] = \lambda x.\ \mathbf{teacher}(x) \\
[\![\mathit{like}]\!] : \mathtt{e} \to (\mathtt{e} \to \mathtt{t}) & [\![\mathit{like}]\!] = \lambda x.\ \lambda y.\ \mathbf{like}(x)(y) \\
[\![\mathit{jump}]\!] : \mathtt{e} \to \mathtt{t} & [\![\mathit{jump}]\!] = \lambda x.\ \mathbf{jump}(x)
\end{array}
$$

  - Some special terminals ("traces"/pronouns) may denote bound variables or type $\mathtt{e}$.
  - We keep determiners for later.
- Now let's try to combine all those things together!

Functional Application



$[\![Mary]\!] = \mathbf{M} \in D$

$[\![is]\!] = \lambda P.\ P$

$[\![happy]\!] = \lambda x.\ \mathbf{happy}(x)$

- Compositionality, again: the meaning of *Mary is happy* depends on the meanings of *Mary*, *is*, and *happy*, and how they combine together.
- To combine 2 nodes together, we introduce the rule of **Functional Application (FA)**:

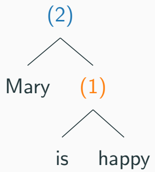
If $X : \alpha$ merges with $Y : \alpha \to \beta$, then $[\![\mathrm{Y\ X}]\!]^2 = \mathrm{Y(X)}$.

$$[\![(1)]\!] = [\![is\ happy]\!] \overset{FA}{=} [\![is]\!]([\![happy]\!]) = [\![happy]\!] = \lambda x.\ \mathbf{happy}(x)$$

$$\begin{aligned} [\![(2)]\!] = [\![Mary\ is\ happy]\!] &\overset{FA}{=} [\![is\ happy]\!]([\![Mary]\!]) \\ &= (\lambda x.\ \mathbf{happy}(x))(\mathbf{M}) \\ &= 1 \text{ iff } \mathbf{happy}(\mathbf{M}) \end{aligned}$$

[2] We assume X and Y are unordered here; i.e. FA also works if X comes before Y.

Functional Application



$[\![Mary]\!] = \mathbf{M} \in D$

$[\![is]\!] = \lambda P.\ P$

$[\![happy]\!] = \lambda x.\ \mathbf{happy}(x)$

- Compositionality, again: the meaning of *Mary is happy* depends on the meanings of *Mary*, *is*, and *happy*, and how they combine together.
- To combine 2 nodes together, we introduce the rule of **Functional Application (FA)**:

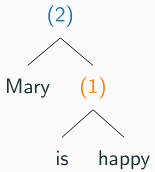
> If $X : \alpha$ merges with $Y : \alpha \to \beta$, then $[\![\text{Y X}]\!]$[2] $= \text{Y(X)}$.

$$[\![(1)]\!] = [\![is\ happy]\!] \overset{FA}{=} [\![is]\!]([\![happy]\!]) = [\![happy]\!] = \lambda x.\ \mathbf{happy}(x)$$

$$\begin{aligned} [\![(2)]\!] = [\![Mary\ is\ happy]\!] &\overset{FA}{=} [\![is\ happy]\!]([\![Mary]\!]) \\ &= (\lambda x.\ \mathbf{happy}(x))(\mathbf{M}) \\ &= 1 \text{ iff } \mathbf{happy}(\mathbf{M}) \end{aligned}$$

[2]We assume X and Y are unordered here; i.e. FA also works if X comes before Y.

Functional Application



$[\![Mary]\!] = \mathbf{M} \in D$

$[\![is]\!] = \lambda P.\ P$

$[\![happy]\!] = \lambda x.\ \mathbf{happy}(x)$

- Compositionality, again: the meaning of *Mary is happy* depends on the meanings of *Mary*, *is*, and *happy*, and how they combine together.
- To combine 2 nodes together, we introduce the rule of **Functional Application (FA)**:

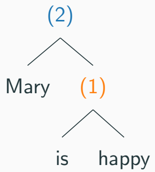
> If $X : \alpha$ merges with $Y : \alpha \to \beta$, then $[\![\text{Y X}]\!]$[2] $= \text{Y(X)}$.

$$[\![(1)]\!] = [\![is\ happy]\!] \overset{FA}{=} [\![is]\!]([\![happy]\!]) = [\![happy]\!] = \lambda x.\ \mathbf{happy}(x)$$

$$\begin{aligned} [\![(2)]\!] = [\![Mary\ is\ happy]\!] &\overset{FA}{=} [\![is\ happy]\!]([\![Mary]\!]) \\ &= (\lambda x.\ \mathbf{happy}(x))(\mathbf{M}) \\ &= 1 \text{ iff } \mathbf{happy}(\mathbf{M}) \end{aligned}$$

[2]We assume X and Y are unordered here; i.e. FA also works if X comes before Y.

Functional Application



$$[\![Mary]\!] = \mathbf{M} \in D$$

$$[\![is]\!] = \lambda P.\ P$$

$$[\![happy]\!] = \lambda x.\ \mathbf{happy}(x)$$

- Compositionality, again: the meaning of *Mary is happy* depends on the meanings of *Mary*, *is*, and *happy*, and how they combine together.
- To combine 2 nodes together, we introduce the rule of **Functional Application (FA)**:

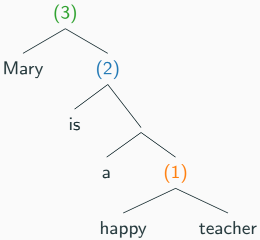
> If $X : \alpha$ merges with $Y : \alpha \rightarrow \beta$, then $[\![\text{Y X}]\!]$[2] $=$ Y(X).

$$[\![(1)]\!] = [\![is\ happy]\!] \overset{FA}{=} [\![is]\!]([\![happy]\!]) = [\![happy]\!] = \lambda x.\ \mathbf{happy}(x)$$

$$\begin{aligned} [\![(2)]\!] = [\![Mary\ is\ happy]\!] &\overset{FA}{=} [\![is\ happy]\!]([\![Mary]\!]) \\ &= (\lambda x.\ \mathbf{happy}(x))(\mathbf{M}) \\ &= 1 \text{ iff } \mathbf{happy}(\mathbf{M}) \end{aligned}$$

[2] We assume X and Y are unordered here; i.e. FA also works if X comes before Y.

Predicate Modification



$$[\![Mary]\!] = \mathbf{M} \in D$$

$$[\![is]\!] = [\![a]\!] = \lambda P.\ P$$

$$[\![happy]\!] = \lambda x.\ \mathbf{happy}(x)$$

$$[\![teacher]\!] = \lambda x.\ \mathbf{teacher}(x)$$

- Both *happy* and *teacher* denote functions of type $\mathtt{e} \rightarrow \mathtt{t}$… we can't combine them with Functional Application!
- To combine 2 nodes of type $\alpha \rightarrow \mathtt{t}$, we introduce the rule of **Predicate Modification (PM)**:

If $P : \alpha \rightarrow \mathtt{t}$ merges with $Q : \alpha \rightarrow \mathtt{t}$, then $[\![\mathrm{P\ Q}]\!] = \lambda x.\ P(x) \wedge Q(x)$

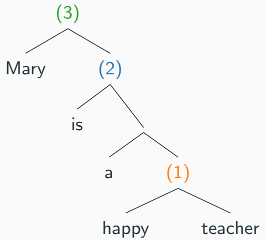
$$[\![(1)]\!] = [\![(2)]\!] = [\![happy\ teacher]\!] \overset{PM}{=} \lambda x.\ \mathbf{happy}(x) \wedge \mathbf{teacher}(x)$$

$$[\![(3)]\!] = [\![Mary\ is\ a\ happy\ teacher]\!] \overset{FA}{=} [\![happy\ teacher]\!]([\![Mary]\!])$$

$$= 1 \text{ iff } \mathbf{happy}(\mathbf{M}) \wedge \mathbf{teacher}(\mathbf{M})^{3}$$

[3] Food for thought: does the sentence really mean that Mary is happy, and is a teacher? Or does it rather mean that Mary is happy, *for a teacher*?

Predicate Modification



$[\![\textit{Mary}]\!] = \mathbf{M} \in D$

$[\![\textit{is}]\!] = [\![\textit{a}]\!] = \lambda P.\ P$

$[\![\textit{happy}]\!] = \lambda x.\ \textbf{happy}(x)$

$[\![\textit{teacher}]\!] = \lambda x.\ \textbf{teacher}(x)$

- Both *happy* and *teacher* denote functions of type $\mathtt{e} \to \mathtt{t}$... we can't combine them with Functional Application!
- To combine 2 nodes of type $\alpha \to \mathtt{t}$, we introduce the rule of **Predicate Modification (PM)**:

If $P : \alpha \to \mathtt{t}$ merges with $Q : \alpha \to \mathtt{t}$, then $[\![\mathrm{P\ Q}]\!] = \lambda x.\ P(x) \wedge Q(x)$

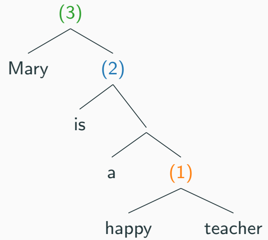
$$[\![(1)]\!] = [\![(2)]\!] = [\![\textit{happy teacher}]\!] \overset{PM}{=} \lambda x.\ \textbf{happy}(x) \wedge \textbf{teacher}(x)$$

$$[\![(3)]\!] = [\![\textit{Mary is a happy teacher}]\!] \overset{FA}{=} [\![\textit{happy teacher}]\!]([\![\textit{Mary}]\!])$$

$$= 1 \text{ iff } \textbf{happy}(\mathbf{M}) \wedge \textbf{teacher}(\mathbf{M})^3$$

[3] Food for thought: does the sentence really mean that Mary is happy, and is a teacher? Or does it rather mean that Mary is happy, *for a teacher*?

Predicate Modification



$$[\![\textit{Mary}]\!] = \mathbf{M} \in D$$

$$[\![\textit{is}]\!] = [\![\textit{a}]\!] = \lambda P.\ P$$

$$[\![\textit{happy}]\!] = \lambda x.\ \mathbf{happy}(x)$$

$$[\![\textit{teacher}]\!] = \lambda x.\ \mathbf{teacher}(x)$$

- Both *happy* and *teacher* denote functions of type $\mathtt{e} \to \mathtt{t}$... we can't combine them with Functional Application!
- To combine 2 nodes of type $\alpha \to \mathtt{t}$, we introduce the rule of **Predicate Modification (PM)**:

$$\boxed{\text{If } P : \alpha \to \mathtt{t} \text{ merges with } Q : \alpha \to \mathtt{t}, \text{ then } [\![\mathrm{P\ Q}]\!] = \lambda x.\ P(x) \wedge Q(x)}$$

$$[\![(1)]\!] = [\![(2)]\!] = [\![\textit{happy teacher}]\!] \overset{PM}{=} \lambda x.\ \mathbf{happy}(x) \wedge \mathbf{teacher}(x)$$

$$[\![(3)]\!] = [\![\textit{Mary is a happy teacher}]\!] \overset{FA}{=} [\![\textit{happy teacher}]\!]([\![\textit{Mary}]\!])$$

$$= 1 \text{ iff } \mathbf{happy}(\mathbf{M}) \wedge \mathbf{teacher}(\mathbf{M})^3$$

[3] Food for thought: does the sentence really mean that Mary is happy, and is a teacher? Or does it rather mean that Mary is happy, *for a teacher*?

Taking stock

- What you might think at this point: we started with a string saying "Mary is happy" and ended up with the meaning that "Mary is happy"...well that's not so impressive.
- First, we should keep in mind that the 2 "Mary is happy" are in different *languages*.
    - The **object language** (the one used in the string/nodes in the tree) is the one that is to be *interpreted*. It could be English, French, or Klingon.
    - The **meta-language** (the one used in the semantic denotation of the sentence) is the language used to *describe* the object language. It is logical in nature, although it often gets paraphrased using English, for convenience only.
- Second, our enterprise was not entirely vacuous in that we devised a simple tree-interpretation algorithm to convert (ideally) **any string from the object-language into the meta-language**.
- This entails that the meaning of each individual sentence does not need to get memorized separately!

Taking stock

- What you might think at this point: we started with a string saying "Mary is happy" and ended up with the meaning that "Mary is happy"...well that's not so impressive.
- First, we should keep in mind that the 2 "Mary is happy" are in different *languages*.
  - The **object language** (the one used in the string/nodes in the tree) is the one that is to be *interpreted*. It could be English, French, or Klingon.
  - The **meta-language** (the one used in the semantic denotation of the sentence) is the language used to *describe* the object language. It is logical in nature, although it often gets paraphrased using English, for convenience only.
- Second, our enterprise was not entirely vacuous in that we devised a simple tree-interpretation algorithm to convert (ideally) **any string from the object-language into the meta-language**.
- This entails that the meaning of each individual sentence does not need to get memorized separately!

Taking stock

- What you might think at this point: we started with a string saying "Mary is happy" and ended up with the meaning that "Mary is happy"...well that's not so impressive.
- First, we should keep in mind that the 2 "Mary is happy" are in different *languages*.
    - The **object language** (the one used in the string/nodes in the tree) is the one that is to be *interpreted*. It could be English, French, or Klingon.
    - The **meta-language** (the one used in the semantic denotation of the sentence) is the language used to *describe* the object language. It is logical in nature, although it often gets paraphrased using English, for convenience only.
- Second, our enterprise was not entirely vacuous in that we devised a simple tree-interpretation algorithm to convert (ideally) **any string from the object-language into the meta-language**.
- This entails that the meaning of each individual sentence does not need to get memorized separately!

Taking stock

- What you might think at this point: we started with a string saying "Mary is happy" and ended up with the meaning that "Mary is happy"...well that's not so impressive.
- First, we should keep in mind that the 2 "Mary is happy" are in different *languages*.
  - The **object language** (the one used in the string/nodes in the tree) is the one that is to be *interpreted*. It could be English, French, or Klingon.
  - The **meta-language** (the one used in the semantic denotation of the sentence) is the language used to *describe* the object language. It is logical in nature, although it often gets paraphrased using English, for convenience only.
- Second, our enterprise was not entirely vacuous in that we devised a simple tree-interpretation algorithm to convert (ideally) **any string from the object-language into the meta-language**.
- This entails that the meaning of each individual sentence does not need to get memorized separately!

Taking stock

- What you might think at this point: we started with a string saying "Mary is happy" and ended up with the meaning that "Mary is happy"...well that's not so impressive.
- First, we should keep in mind that the 2 "Mary is happy" are in different *languages*.
    - The **object language** (the one used in the string/nodes in the tree) is the one that is to be *interpreted*. It could be English, French, or Klingon.
    - The **meta-language** (the one used in the semantic denotation of the sentence) is the language used to *describe* the object language. It is logical in nature, although it often gets paraphrased using English, for convenience only.
- Second, our enterprise was not entirely vacuous in that we devised a simple tree-interpretation algorithm to convert (ideally) **any string from the object-language into the meta-language**.
- This entails that the meaning of each individual sentence does not need to get memorized separately!

Taking stock

- What you might think at this point: we started with a string saying "Mary is happy" and ended up with the meaning that "Mary is happy"...well that's not so impressive.
- First, we should keep in mind that the 2 "Mary is happy" are in different *languages*.
  - The **object language** (the one used in the string/nodes in the tree) is the one that is to be *interpreted*. It could be English, French, or Klingon.
  - The **meta-language** (the one used in the semantic denotation of the sentence) is the language used to *describe* the object language. It is logical in nature, although it often gets paraphrased using English, for convenience only.
- Second, our enterprise was not entirely vacuous in that we devised a simple tree-interpretation algorithm to convert (ideally) **any string from the object-language into the meta-language**.
- This entails that the meaning of each individual sentence does not need to get memorized separately!

Quantification

Generalized quantifiers

- Natural languages are endowed with various quantifiers: *every*, *some*, *most*, *few*...

(9) Every student smiled. $\rightsquigarrow \forall x.\ \mathbf{student}(x) \implies \mathbf{smiled}(x)$

(10) Some dogs barked. $\rightsquigarrow \exists x.\ \mathbf{dog}(x) \wedge \mathbf{barked}(x)$

- Natural language quantifiers are *restricted*: they do not quantify over the whole set of possible entities, but rather on specific subsets denoted by predicates of type $\mathrm{e} \to \mathtt{t}$ such as **student** in (9) and **dogs** in (10). Those are called **restrictors**.
- Quantifiers moreover relate elements verifying the restrictor to another property, e.g. smiling in (9) or barking in (10). This property, also of type $\mathrm{e} \to \mathtt{t}$, is called the **(nuclear) scope** of the quantifier.
- In brief, a **generalized quantifier** says something about the relation between its restrictor (predicate of type $\mathrm{e} \to \mathtt{t}$) and its scope (also (predicate of type $\mathrm{e} \to \mathtt{t}$)). It is thus a function of type $(\mathrm{e} \to \mathtt{t}) \to (\mathrm{e} \to \mathtt{t}) \to \mathtt{t}$

Generalized quantifiers

- Natural languages are endowed with various quantifiers: *every*, *some*, *most*, *few*...

  (11) Every student smiled. $\rightsquigarrow \forall x.\ \mathbf{student}(x) \implies \mathbf{smiled}(x)$

  (12) Some dogs barked. $\rightsquigarrow \exists x.\ \mathbf{dog}(x) \wedge \mathbf{barked}(x)$

- Natural language quantifiers are *restricted*: they do not quantify over the whole set of possible entities, but rather on specific subsets denoted by predicates of type $\mathtt{e} \to \mathtt{t}$ such as **student** in (9) and **dogs** in (10). Those are called **restrictors**.
- Quantifiers moreover relate elements verifying the restrictor to another property, e.g. smiling in (9) or barking in (10). This property, also of type $\mathtt{e} \to \mathtt{t}$, is called the **(nuclear) scope** of the quantifier.
- In brief, a **generalized quantifier** says something about the relation between its restrictor (predicate of type $\mathtt{e} \to \mathtt{t}$) and its scope (also (predicate of type $\mathtt{e} \to \mathtt{t}$)). It is thus a function of type $(\mathtt{e} \to \mathtt{t}) \to (\mathtt{e} \to \mathtt{t}) \to \mathtt{t}$

Generalized quantifiers

- Natural languages are endowed with various quantifiers: *every*, *some*, *most*, *few*...

  (13) Every student smiled. $\rightsquigarrow \forall x.\ \mathbf{student}(x) \implies \mathbf{smiled}(x)$

  (14) Some dogs barked. $\rightsquigarrow \exists x.\ \mathbf{dog}(x) \land \mathbf{barked}(x)$

- Natural language quantifiers are *restricted*: they do not quantify over the whole set of possible entities, but rather on specific subsets denoted by predicates of type $\mathtt{e} \to \mathtt{t}$ such as **student** in (9) and **dogs** in (10). Those are called **restrictors**.
- Quantifiers moreover relate elements verifying the restrictor to another property, e.g. smiling in (9) or barking in (10). This property, also of type $\mathtt{e} \to \mathtt{t}$, is called the **(nuclear) scope** of the quantifier.
- In brief, a **generalized quantifier** says something about the relation between its restrictor (predicate of type $\mathtt{e} \to \mathtt{t}$) and its scope (also (predicate of type $\mathtt{e} \to \mathtt{t}$)). It is thus a function of type $(\mathtt{e} \to \mathtt{t}) \to (\mathtt{e} \to \mathtt{t}) \to \mathtt{t}$

Generalized quantifiers

- Natural languages are endowed with various quantifiers: *every*, *some*, *most*, *few*...

(15) Every student smiled. $\rightsquigarrow \forall x.\ \mathbf{student}(x) \implies \mathbf{smiled}(x)$

(16) Some dogs barked. $\rightsquigarrow \exists x.\ \mathbf{dog}(x) \land \mathbf{barked}(x)$

- Natural language quantifiers are *restricted*: they do not quantify over the whole set of possible entities, but rather on specific subsets denoted by predicates of type $\mathtt{e} \to \mathtt{t}$ such as **student** in (9) and **dogs** in (10). Those are called **restrictors**.
- Quantifiers moreover relate elements verifying the restrictor to another property, e.g. smiling in (9) or barking in (10). This property, also of type $\mathtt{e} \to \mathtt{t}$, is called the **(nuclear) scope** of the quantifier.
- In brief, a **generalized quantifier** says something about the relation between its restrictor (predicate of type $\mathtt{e} \to \mathtt{t}$) and its scope (also (predicate of type $\mathtt{e} \to \mathtt{t}$)). It is thus a function of type $(\mathtt{e} \to \mathtt{t}) \to (\mathtt{e} \to \mathtt{t}) \to \mathtt{t}$

Interpretation of quantification within set theory

- It might be easier to understand what quantifiers do by **viewing predicates as sets**.
- We can do this, because a function of type $\alpha \rightarrow \mathtt{t}$ is the indicator function of a subset of elements of type α. So in particular, **a function P of type $\mathtt{e} \rightarrow \mathtt{t}$ is the indicator function the set of all entities of type $\mathtt{e}$ verifying P.**
- For instance, the predicate $[\![\textit{teacher}]\!]$ is equivalent to the set of all individuals that are teachers.
- Given this equivalence, we can see generalized quantifiers as functions from a pair of sets (restrictor set, nuclear scope set), to a truth value.
  - $[\![\ \textit{some}\ ]\!](P)(Q) = 1$ iff $P \cap Q \neq \emptyset$
  - $[\![\ \textit{all}\ ]\!](P)(Q) = 1$ iff $P \subseteq Q$
  - $[\![\ \textit{exactly 3}\ ]\!](P)(Q) = 1$ iff $|P \cap Q| = 3$
  - $[\![\ \textit{less than half}\ ]\!](P)(Q) = 1$ iff $\frac{|P \cap Q|}{|P|} < 1/2$
  - ...

Interpretation of quantification within set theory

- It might be easier to understand what quantifiers do by **viewing predicates as sets**.
- We can do this, because a function of type $\alpha \to \texttt{t}$ is the indicator function of a subset of elements of type α. So in particular, **a function P of type $\texttt{e} \to \texttt{t}$ is the indicator function the set of all entities of type $\texttt{e}$ verifying P.**
- For instance, the predicate $[\![\textit{teacher}]\!]$ is equivalent to the set of all individuals that are teachers.
- Given this equivalence, we can see generalized quantifiers as functions from a pair of sets (restrictor set, nuclear scope set), to a truth value.
  - $[\![\ \textit{some}\ ]\!](P)(Q) = 1$ iff $P \cap Q \neq \emptyset$
  - $[\![\ \textit{all}\ ]\!](P)(Q) = 1$ iff $P \subseteq Q$
  - $[\![\ \textit{exactly 3}\ ]\!](P)(Q) = 1$ iff $|P \cap Q| = 3$
  - $[\![\ \textit{less than half}\ ]\!](P)(Q) = 1$ iff $\frac{|P \cap Q|}{|P|} < 1/2$
  - ...

Interpretation of quantification within set theory

- It might be easier to understand what quantifiers do by **viewing predicates as sets**.
- We can do this, because a function of type $\alpha \to \mathtt{t}$ is the indicator function of a subset of elements of type α. So in particular, **a function P of type $\mathtt{e} \to \mathtt{t}$ is the indicator function the set of all entities of type $\mathtt{e}$ verifying P.**
- For instance, the predicate $[\![\textit{teacher}]\!]$ is equivalent to the set of all individuals that are teachers.
- Given this equivalence, we can see generalized quantifiers as functions from a pair of sets (restrictor set, nuclear scope set), to a truth value.
  - $[\![\ \textit{some}\ ]\!](P)(Q) = 1$ iff $P \cap Q \neq \emptyset$
  - $[\![\ \textit{all}\ ]\!](P)(Q) = 1$ iff $P \subseteq Q$
  - $[\![\ \textit{exactly 3}\ ]\!](P)(Q) = 1$ iff $|P \cap Q| = 3$
  - $[\![\ \textit{less than half}\ ]\!](P)(Q) = 1$ iff $\frac{|P \cap Q|}{|P|} < 1/2$
  - ...

Interpretation of quantification within set theory

- It might be easier to understand what quantifiers do by **viewing predicates as sets**.
- We can do this, because a function of type $\alpha \to \mathtt{t}$ is the indicator function of a subset of elements of type α. So in particular, **a function P of type $\mathtt{e} \to \mathtt{t}$ is the indicator function the set of all entities of type $\mathtt{e}$ verifying P.**
- For instance, the predicate $[\![\textit{teacher}]\!]$ is equivalent to the set of all individuals that are teachers.
- Given this equivalence, we can see generalized quantifiers as functions from a pair of sets (restrictor set, nuclear scope set), to a truth value.
  - $[\![\ \textit{some}\ ]\!](P)(Q) = 1$ iff $P \cap Q \neq \emptyset$
  - $[\![\ \textit{all}\ ]\!](P)(Q) = 1$ iff $P \subseteq Q$
  - $[\![\ \textit{exactly 3}\ ]\!](P)(Q) = 1$ iff $|P \cap Q| = 3$
  - $[\![\ \textit{less than half}\ ]\!](P)(Q) = 1$ iff $\frac{|P \cap Q|}{|P|} < 1/2$
  - ...

Interpretation of quantification within set theory

- It might be easier to understand what quantifiers do by **viewing predicates as sets**.
- We can do this, because a function of type $\alpha \to \texttt{t}$ is the indicator function of a subset of elements of type α. So in particular, **a function P of type $\texttt{e} \to \texttt{t}$ is the indicator function the set of all entities of type $\texttt{e}$ verifying P.**
- For instance, the predicate $[\![\textit{teacher}]\!]$ is equivalent to the set of all individuals that are teachers.
- Given this equivalence, we can see generalized quantifiers as functions from a pair of sets (restrictor set, nuclear scope set), to a truth value.
  - $[\![\ \textit{some}\ ]\!](P)(Q) = 1$ iff $P \cap Q \neq \emptyset$
  - $[\![\ \textit{all}\ ]\!](P)(Q) = 1$ iff $P \subseteq Q$
  - $[\![\ \textit{exactly 3}\ ]\!](P)(Q) = 1$ iff $|P \cap Q| = 3$
  - $[\![\ \textit{less than half}\ ]\!](P)(Q) = 1$ iff $\frac{|P \cap Q|}{|P|} < 1/2$
  - ...

Interpretation of quantification within set theory

- It might be easier to understand what quantifiers do by **viewing predicates as sets**.
- We can do this, because a function of type $\alpha \to \mathtt{t}$ is the indicator function of a subset of elements of type α. So in particular, **a function P of type $\mathtt{e} \to \mathtt{t}$ is the indicator function the set of all entities of type $\mathtt{e}$ verifying P.**
- For instance, the predicate $[\![\textit{teacher}]\!]$ is equivalent to the set of all individuals that are teachers.
- Given this equivalence, we can see generalized quantifiers as functions from a pair of sets (restrictor set, nuclear scope set), to a truth value.
  - $[\![\ \textit{some}\ ]\!](P)(Q) = 1$ iff $P \cap Q \neq \emptyset$
  - $[\![\ \textit{all}\ ]\!](P)(Q) = 1$ iff $P \subseteq Q$
  - $[\![\ \textit{exactly 3}\ ]\!](P)(Q) = 1$ iff $|P \cap Q| = 3$
  - $[\![\ \textit{less than half}\ ]\!](P)(Q) = 1$ iff $\frac{|P \cap Q|}{|P|} < 1/2$
  - ...

Interpretation of quantification within set theory

- It might be easier to understand what quantifiers do by **viewing predicates as sets**.
- We can do this, because a function of type $\alpha \to \mathtt{t}$ is the indicator function of a subset of elements of type α. So in particular, **a function P of type $\mathtt{e} \to \mathtt{t}$ is the indicator function the set of all entities of type $\mathtt{e}$ verifying P.**
- For instance, the predicate $[\![\textit{teacher}]\!]$ is equivalent to the set of all individuals that are teachers.
- Given this equivalence, we can see generalized quantifiers as functions from a pair of sets (restrictor set, nuclear scope set), to a truth value.
  - $[\![\ \textit{some}\ ]\!](P)(Q) = 1$ iff $P \cap Q \neq \emptyset$
  - $[\![\ \textit{all}\ ]\!](P)(Q) = 1$ iff $P \subseteq Q$
  - $[\![\ \textit{exactly 3}\ ]\!](P)(Q) = 1$ iff $|P \cap Q| = 3$
  - $[\![\ \textit{less than half}\ ]\!](P)(Q) = 1$ iff $\frac{|P \cap Q|}{|P|} < 1/2$
  - ...

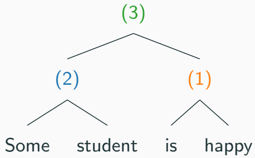
Interpretation of quantification within set theory

- It might be easier to understand what quantifiers do by **viewing predicates as sets**.
- We can do this, because a function of type $\alpha \to \mathtt{t}$ is the indicator function of a subset of elements of type α. So in particular, **a function P of type $\mathtt{e} \to \mathtt{t}$ is the indicator function the set of all entities of type $\mathtt{e}$ verifying P.**
- For instance, the predicate $[\![\textit{teacher}]\!]$ is equivalent to the set of all individuals that are teachers.
- Given this equivalence, we can see generalized quantifiers as functions from a pair of sets (restrictor set, nuclear scope set), to a truth value.
    - $[\![\ \textit{some}\ ]\!](P)(Q) = 1$ iff $P \cap Q \neq \emptyset$
    - $[\![\ \textit{all}\ ]\!](P)(Q) = 1$ iff $P \subseteq Q$
    - $[\![\ \textit{exactly 3}\ ]\!](P)(Q) = 1$ iff $|P \cap Q| = 3$
    - $[\![\ \textit{less than half}\ ]\!](P)(Q) = 1$ iff $\frac{|P \cap Q|}{|P|} < 1/2$
    - ...

Interpretation of quantification within set theory

- It might be easier to understand what quantifiers do by **viewing predicates as sets**.
- We can do this, because a function of type $\alpha \to \mathtt{t}$ is the indicator function of a subset of elements of type α. So in particular, **a function P of type $\mathtt{e} \to \mathtt{t}$ is the indicator function the set of all entities of type $\mathtt{e}$ verifying P.**
- For instance, the predicate $[\![\textit{teacher}]\!]$ is equivalent to the set of all individuals that are teachers.
- Given this equivalence, we can see generalized quantifiers as functions from a pair of sets (restrictor set, nuclear scope set), to a truth value.
  - $[\![\ \textit{some}\ ]\!](P)(Q) = 1$ iff $P \cap Q \neq \emptyset$
  - $[\![\ \textit{all}\ ]\!](P)(Q) = 1$ iff $P \subseteq Q$
  - $[\![\ \textit{exactly 3}\ ]\!](P)(Q) = 1$ iff $|P \cap Q| = 3$
  - $[\![\ \textit{less than half}\ ]\!](P)(Q) = 1$ iff $\frac{|P \cap Q|}{|P|} < 1/2$
  - ...

Denotation of a quantified sentence



$$[\![\textit{Some}]\!] = \lambda P.\lambda Q.\ \exists x.\ P(x) \wedge Q(x)$$

$$[\![\textit{student}]\!] = \lambda x.\ \textbf{student}(x)$$

$$[\![\textit{is}]\!] = \lambda P.\ P$$

$$[\![\textit{happy}]\!] = \lambda x.\ \textbf{happy}(x)$$

$$[\![(1)]\!] = [\![\textit{is happy}]\!] = [\![\textit{happy}]\!] = \lambda x.\ \textbf{happy}(x)$$

$$\begin{aligned}
[\![(2)]\!] &= [\![\textit{Some student}]\!] \overset{FA}{=} [\![\textit{some}]\!]([\![\textit{student}]\!]) \\
&= (\lambda P.\ \lambda Q.\ \exists x.\ P(x) \wedge Q(x))(\lambda y.\ \textbf{student}(y)) \\
&= \lambda Q.\ \exists x.\ (\lambda y.\ \textbf{student}(y))(x) \wedge Q(x) \\
&= \lambda Q.\ \exists x.\ \textbf{student}(x) \wedge Q(x)
\end{aligned}$$

$$\begin{aligned}
[\![(3)]\!] &= [\![\textit{Some student is happy}]\!] \overset{FA}{=} [\![\textit{Some student}]\!]([\![\textit{is happy}]\!]) \\
&= (\lambda Q.\ \exists x.\ \textbf{student}(x) \wedge Q(x))(\lambda y.\ \textbf{happy}(y)) \\
&= \exists x.\ \textbf{student}(x) \wedge (\lambda y.\ \textbf{happy}(y))(x) \\
&= \exists x.\ \textbf{student}(x) \wedge \textbf{happy}(x)
\end{aligned}$$ [4]

[4] Food for thought: this meaning is compatible with *all* ($\forall$) students being happy. Is this consistent with your intuitions about *some*? Should we then change the lexical entry of *some*?

Quantifier monotonicity

- An interesting property to study with quantifiers is **monotonicity**, i.e. how quantifiers influence entailment patterns verified by their arguments (restrictor, and scope).
- Recall from basic functional analysis that a function is monotone (increasing or decreasing), if resp. it preserves or reverses the ordering of its arguments:
  - f is (strictly) increasing if $\forall x_1 < x_2.\ f(x_1) < f(x_2)$.
  - f is (strictly) decreasing if $\forall x_1 < x_2.\ f(x_1) > f(x_2)$.
- Likewise, a function Q applying to predicates is upward monotone if it leaves the entailment pattern between any 2 of its potential arguments unchanged; and it is downward monotone if it reverses any entailment pattern between its potential arguments.
  - Q is **upward monotone** if $\forall P_1, P_2 :\ P_1 \subseteq P_2.\ Q(P_1) \Rightarrow Q(P_2)$.
  - Q is **downward monotone** if $\forall P_1, P_2 :\ P_1 \subseteq P_2.\ Q(P_1) \Leftarrow Q(P_2)$.
- Generalized quantifiers are functions from pairs of predicates to truth values. To assess monotonicity, one must thus look at a partially applied generalized quantifier. We'll see how this works for *all* on the next slide.

Quantifier monotonicity

- An interesting property to study with quantifiers is **monotonicity**, i.e. how quantifiers influence entailment patterns verified by their arguments (restrictor, and scope).
- Recall from basic functional analysis that a function is monotone (increasing or decreasing), if resp. it preserves or reverses the ordering of its arguments:
  - f is (strictly) increasing if $\forall x_1 < x_2.\ f(x_1) < f(x_2)$.
  - f is (strictly) decreasing if $\forall x_1 < x_2.\ f(x_1) > f(x_2)$.
- Likewise, a function Q applying to predicates is upward monotone if it leaves the entailment pattern between any 2 of its potential arguments unchanged; and it is downward monotone if it reverses any entailment pattern between its potential arguments.
  - Q is **upward monotone** if $\forall P_1, P_2 :\ P_1 \subseteq P_2.\ Q(P_1) \Rightarrow Q(P_2)$.
  - Q is **downward monotone** if $\forall P_1, P_2 :\ P_1 \subseteq P_2.\ Q(P_1) \Leftarrow Q(P_2)$.
- Generalized quantifiers are functions from pairs of predicates to truth values. To assess monotonicity, one must thus look at a partially applied generalized quantifier. We'll see how this works for *all* on the next slide.

Quantifier monotonicity

- An interesting property to study with quantifiers is **monotonicity**, i.e. how quantifiers influence entailment patterns verified by their arguments (restrictor, and scope).
- Recall from basic functional analysis that a function is monotone (increasing or decreasing), if resp. it preserves or reverses the ordering of its arguments:
  - f is (strictly) increasing if $\forall x_1 < x_2.\ f(x_1) < f(x_2)$.
  - f is (strictly) decreasing if $\forall x_1 < x_2.\ f(x_1) > f(x_2)$.
- Likewise, a function Q applying to predicates is upward monotone if it leaves the entailment pattern between any 2 of its potential arguments unchanged; and it is downward monotone if it reverses any entailment pattern between its potential arguments.
  - Q is **upward monotone** if $\forall P_1, P_2 :\ P_1 \subseteq P_2.\ Q(P_1) \Rightarrow Q(P_2)$.
  - Q is **downward monotone** if $\forall P_1, P_2 :\ P_1 \subseteq P_2.\ Q(P_1) \Leftarrow Q(P_2)$.
- Generalized quantifiers are functions from pairs of predicates to truth values. To assess monotonicity, one must thus look at a partially applied generalized quantifier. We'll see how this works for *all* on the next slide.

Quantifier monotonicity

- An interesting property to study with quantifiers is **monotonicity**, i.e. how quantifiers influence entailment patterns verified by their arguments (restrictor, and scope).
- Recall from basic functional analysis that a function is monotone (increasing or decreasing), if resp. it preserves or reverses the ordering of its arguments:
  - f is (strictly) increasing if $\forall x_1 < x_2.\ f(x_1) < f(x_2)$.
  - f is (strictly) decreasing if $\forall x_1 < x_2.\ f(x_1) > f(x_2)$.
- Likewise, a function Q applying to predicates is upward monotone if it leaves the entailment pattern between any 2 of its potential arguments unchanged; and it is downward monotone if it reverses any entailment pattern between its potential arguments.
  - Q is **upward monotone** if $\forall P_1, P_2 :\ P_1 \subseteq P_2.\ Q(P_1) \Rightarrow Q(P_2)$.
  - Q is **downward monotone** if $\forall P_1, P_2 :\ P_1 \subseteq P_2.\ Q(P_1) \Leftarrow Q(P_2)$.
- Generalized quantifiers are functions from pairs of predicates to truth values. To assess monotonicity, one must thus look at a partially applied generalized quantifier. We'll see how this works for *all* on the next slide.

Quantifier monotonicity

- An interesting property to study with quantifiers is **monotonicity**, i.e. how quantifiers influence entailment patterns verified by their arguments (restrictor, and scope).
- Recall from basic functional analysis that a function is monotone (increasing or decreasing), if resp. it preserves or reverses the ordering of its arguments:
  - f is (strictly) increasing if $\forall x_1 < x_2.\ f(x_1) < f(x_2)$.
  - f is (strictly) decreasing if $\forall x_1 < x_2.\ f(x_1) > f(x_2)$.
- Likewise, a function Q applying to predicates is upward monotone if it leaves the entailment pattern between any 2 of its potential arguments unchanged; and it is downward monotone if it reverses any entailment pattern between its potential arguments.
  - Q is **upward monotone** if $\forall P_1, P_2 :\ P_1 \subseteq P_2.\ Q(P_1) \Rightarrow Q(P_2)$.
  - Q is **downward monotone** if $\forall P_1, P_2 :\ P_1 \subseteq P_2.\ Q(P_1) \Leftarrow Q(P_2)$.
- Generalized quantifiers are functions from pairs of predicates to truth values. To assess monotonicity, one must thus look at a partially applied generalized quantifier. We'll see how this works for *all* on the next slide.

Quantifier monotonicity

- An interesting property to study with quantifiers is **monotonicity**, i.e. how quantifiers influence entailment patterns verified by their arguments (restrictor, and scope).
- Recall from basic functional analysis that a function is monotone (increasing or decreasing), if resp. it preserves or reverses the ordering of its arguments:
    - f is (strictly) increasing if $\forall x_1 < x_2.\ f(x_1) < f(x_2)$.
    - f is (strictly) decreasing if $\forall x_1 < x_2.\ f(x_1) > f(x_2)$.
- Likewise, a function Q applying to predicates is upward monotone if it leaves the entailment pattern between any 2 of its potential arguments unchanged; and it is downward monotone if it reverses any entailment pattern between its potential arguments.
    - Q is **upward monotone** if $\forall P_1, P_2 :\ P_1 \subseteq P_2.\ Q(P_1) \Rightarrow Q(P_2)$.
    - Q is **downward monotone** if $\forall P_1, P_2 :\ P_1 \subseteq P_2.\ Q(P_1) \Leftarrow Q(P_2)$.
- Generalized quantifiers are functions from pairs of predicates to truth values. To assess monotonicity, one must thus look at a partially applied generalized quantifier. We'll see how this works for *all* on the next slide.

Quantifier monotonicity

- An interesting property to study with quantifiers is **monotonicity**, i.e. how quantifiers influence entailment patterns verified by their arguments (restrictor, and scope).
- Recall from basic functional analysis that a function is monotone (increasing or decreasing), if resp. it preserves or reverses the ordering of its arguments:
    - f is (strictly) increasing if $\forall x_1 < x_2.\ f(x_1) < f(x_2)$.
    - f is (strictly) decreasing if $\forall x_1 < x_2.\ f(x_1) > f(x_2)$.
- Likewise, a function Q applying to predicates is upward monotone if it leaves the entailment pattern between any 2 of its potential arguments unchanged; and it is downward monotone if it reverses any entailment pattern between its potential arguments.
    - Q is **upward monotone** if $\forall P_1, P_2 :\ P_1 \subseteq P_2.\ Q(P_1) \Rightarrow Q(P_2)$.
    - Q is **downward monotone** if $\forall P_1, P_2 :\ P_1 \subseteq P_2.\ Q(P_1) \Leftarrow Q(P_2)$.
- Generalized quantifiers are functions from pairs of predicates to truth values. To assess monotonicity, one must thus look at a partially applied generalized quantifier. We'll see how this works for *all* on the next slide.

Quantifier monotonicity

- An interesting property to study with quantifiers is **monotonicity**, i.e. how quantifiers influence entailment patterns verified by their arguments (restrictor, and scope).
- Recall from basic functional analysis that a function is monotone (increasing or decreasing), if resp. it preserves or reverses the ordering of its arguments:
  - f is (strictly) increasing if $\forall x_1 < x_2.\ f(x_1) < f(x_2)$.
  - f is (strictly) decreasing if $\forall x_1 < x_2.\ f(x_1) > f(x_2)$.
- Likewise, a function Q applying to predicates is upward monotone if it leaves the entailment pattern between any 2 of its potential arguments unchanged; and it is downward monotone if it reverses any entailment pattern between its potential arguments.
  - Q is **upward monotone** if $\forall P_1, P_2 :\ \ P_1 \subseteq P_2.\ Q(P_1) \Rightarrow Q(P_2)$.
  - Q is **downward monotone** if $\forall P_1, P_2 :\ \ P_1 \subseteq P_2.\ Q(P_1) \Leftarrow Q(P_2)$.
- Generalized quantifiers are functions from pairs of predicates to truth values. To assess monotonicity, one must thus look at a partially applied generalized quantifier. We'll see how this works for *all* on the next slide.

Quantifier monotonicity: the case of all ($\forall$)

- Let's consider $[\![\textit{all students}]\!] = \lambda P.\ \forall x.\ \textbf{student}(x) \Rightarrow P(x)$. It is the quantifier *all* partially applied to its restrictor (the set of students). Is it monotone w.r.t. its nuclear scope argument?
  - Let's consider $P_1 = \lambda x.\ \textbf{french}(x)$ and $P_2 = \lambda x.\ \textbf{european}(x)$. We have $P_1 \subseteq P_2$.
  - Moreover, if all students are French then all students are European, in other words, $[\![\textit{all students}]\!](P_1) \Rightarrow [\![\textit{all students}]\!](P_2)$.
  - **All is upward monotone w.r.t. is nuclear scope**.
- Let's consider $\lambda P.\ [\![\textit{all}]\!](P)([\![\textit{delicious}]\!]) = \lambda P.\ \forall x.\ P(x) \Rightarrow \textbf{delicious}(x)$. It is the quantifier *all* partially applied to its nuclear scope (the set of delicious things). Is it monotone w.r.t. its restrictor?
  - Let's consider $P_1 = \lambda x.\ \textbf{choco-cookie}(x)$ and $P_2 = \lambda x.\ \textbf{cookie}(x)$. We have $P_1 \subseteq P_2$.
  - Moreover, if every cookie is delicious then every choco-cookie is too, in other words, $[\![\textit{all}]\!](P_1)([\![\textit{delicious}]\!]) \Leftarrow [\![\textit{all}]\!](P_2)([\![\textit{delicious}]\!])$.
  - **All is downward monotone w.r.t. its restrictor**.

Quantifier monotonicity: the case of all ($\forall$)

- Let's consider $[\![\textit{all students}]\!] = \lambda P.\ \forall x.\ \textbf{student}(x) \Rightarrow P(x)$. It is the quantifier *all* partially applied to its restrictor (the set of students). Is it monotone w.r.t. its nuclear scope argument?
  - Let's consider $P_1 = \lambda x.\ \textbf{french}(x)$ and $P_2 = \lambda x.\ \textbf{european}(x)$. We have $P_1 \subseteq P_2$.
  - Moreover, if all students are French then all students are European, in other words, $[\![\textit{all students}]\!](P_1) \Rightarrow [\![\textit{all students}]\!](P_2)$.
  - **All is upward monotone w.r.t. is nuclear scope**.
- Let's consider $\lambda P.\ [\![\textit{all}]\!](P)([\![\textit{delicious}]\!]) = \lambda P.\ \forall x.\ P(x) \Rightarrow \textbf{delicious}(x)$. It is the quantifier *all* partially applied to its nuclear scope (the set of delicious things). Is it monotone w.r.t. its restrictor?
  - Let's consider $P_1 = \lambda x.\ \textbf{choco-cookie}(x)$ and $P_2 = \lambda x.\ \textbf{cookie}(x)$. We have $P_1 \subseteq P_2$.
  - Moreover, if every cookie is delicious then every choco-cookie is too, in other words, $[\![\textit{all}]\!](P_1)([\![\textit{delicious}]\!]) \Leftarrow [\![\textit{all}]\!](P_2)([\![\textit{delicious}]\!])$.
  - **All is downward monotone w.r.t. its restrictor**.

Quantifier monotonicity: the case of all ($\forall$)

- Let's consider $[\![\textit{all students}]\!] = \lambda P.\ \forall x.\ \textbf{student}(x) \Rightarrow P(x)$. It is the quantifier *all* partially applied to its restrictor (the set of students). Is it monotone w.r.t. its nuclear scope argument?
  - Let's consider $P_1 = \lambda x.\ \textbf{french}(x)$ and $P_2 = \lambda x.\ \textbf{european}(x)$. We have $P_1 \subseteq P_2$.
  - Moreover, if all students are French then all students are European, in other words, $[\![\textit{all students}]\!](P_1) \Rightarrow [\![\textit{all students}]\!](P_2)$.
  - **All is upward monotone w.r.t. is nuclear scope**.
- Let's consider $\lambda P.\ [\![\textit{all}]\!](P)([\![\textit{delicious}]\!]) = \lambda P.\ \forall x.\ P(x) \Rightarrow \textbf{delicious}(x)$. It is the quantifier *all* partially applied to its nuclear scope (the set of delicious things). Is it monotone w.r.t. its restrictor?
  - Let's consider $P_1 = \lambda x.\ \textbf{choco-cookie}(x)$ and $P_2 = \lambda x.\ \textbf{cookie}(x)$. We have $P_1 \subseteq P_2$.
  - Moreover, if every cookie is delicious then every choco-cookie is too, in other words, $[\![\textit{all}]\!](P_1)([\![\textit{delicious}]\!]) \Leftarrow [\![\textit{all}]\!](P_2)([\![\textit{delicious}]\!])$.
  - **All is downward monotone w.r.t. its restrictor**.

Quantifier monotonicity: the case of all ($\forall$)

- Let's consider $[\![\textit{all students}]\!] = \lambda P.\ \forall x.\ \textbf{student}(x) \Rightarrow P(x)$. It is the quantifier *all* partially applied to its restrictor (the set of students). Is it monotone w.r.t. its nuclear scope argument?
  - Let's consider $P_1 = \lambda x.\ \textbf{french}(x)$ and $P_2 = \lambda x.\ \textbf{european}(x)$. We have $P_1 \subseteq P_2$.
  - Moreover, if all students are French then all students are European, in other words, $[\![\textit{all students}]\!](P_1) \Rightarrow [\![\textit{all students}]\!](P_2)$.
  - **All is upward monotone w.r.t. is nuclear scope**.
- Let's consider $\lambda P.\ [\![\textit{all}]\!](P)([\![\textit{delicious}]\!]) = \lambda P.\ \forall x.\ P(x) \Rightarrow \textbf{delicious}(x)$. It is the quantifier *all* partially applied to its nuclear scope (the set of delicious things). Is it monotone w.r.t. its restrictor?
  - Let's consider $P_1 = \lambda x.\ \textbf{choco-cookie}(x)$ and $P_2 = \lambda x.\ \textbf{cookie}(x)$. We have $P_1 \subseteq P_2$.
  - Moreover, if every cookie is delicious then every choco-cookie is too, in other words, $[\![\textit{all}]\!](P_1)([\![\textit{delicious}]\!]) \Leftarrow [\![\textit{all}]\!](P_2)([\![\textit{delicious}]\!])$.
  - **All is downward monotone w.r.t. its restrictor**.

Quantifier monotonicity: the case of all ($\forall$)

- Let's consider $[\![\textit{all students}]\!] = \lambda P.\ \forall x.\ \textbf{student}(x) \Rightarrow P(x)$. It is the quantifier *all* partially applied to its restrictor (the set of students). Is it monotone w.r.t. its nuclear scope argument?
    - Let's consider $P_1 = \lambda x.\ \textbf{french}(x)$ and $P_2 = \lambda x.\ \textbf{european}(x)$. We have $P_1 \subseteq P_2$.
    - Moreover, if all students are French then all students are European, in other words, $[\![\textit{all students}]\!](P_1) \Rightarrow [\![\textit{all students}]\!](P_2)$.
    - **All is upward monotone w.r.t. is nuclear scope**.
- Let's consider $\lambda P.\ [\![\textit{all}]\!](P)([\![\textit{delicious}]\!]) = \lambda P.\ \forall x.\ P(x) \Rightarrow \textbf{delicious}(x)$. It is the quantifier *all* partially applied to its nuclear scope (the set of delicious things). Is it monotone w.r.t. its restrictor?
    - Let's consider $P_1 = \lambda x.\ \textbf{choco-cookie}(x)$ and $P_2 = \lambda x.\ \textbf{cookie}(x)$. We have $P_1 \subseteq P_2$.
    - Moreover, if every cookie is delicious then every choco-cookie is too, in other words, $[\![\textit{all}]\!](P_1)([\![\textit{delicious}]\!]) \Leftarrow [\![\textit{all}]\!](P_2)([\![\textit{delicious}]\!])$.
    - **All is downward monotone w.r.t. its restrictor**.

Quantifier monotonicity: the case of all ($\forall$)

- Let's consider $[\![\textit{all students}]\!] = \lambda P.\ \forall x.\ \mathbf{student}(x) \Rightarrow P(x)$. It is the quantifier *all* partially applied to its restrictor (the set of students). Is it monotone w.r.t. its nuclear scope argument?
  - Let's consider $P_1 = \lambda x.\ \mathbf{french}(x)$ and $P_2 = \lambda x.\ \mathbf{european}(x)$. We have $P_1 \subseteq P_2$.
  - Moreover, if all students are French then all students are European, in other words, $[\![\textit{all students}]\!](P_1) \Rightarrow [\![\textit{all students}]\!](P_2)$.
  - **All is upward monotone w.r.t. is nuclear scope**.
- Let's consider $\lambda P.\ [\![\textit{all}]\!](P)([\![\textit{delicious}]\!]) = \lambda P.\ \forall x.\ P(x) \Rightarrow \mathbf{delicious}(x)$. It is the quantifier *all* partially applied to its nuclear scope (the set of delicious things). Is it monotone w.r.t. its restrictor?
  - Let's consider $P_1 = \lambda x.\ \mathbf{choco\text{-}cookie}(x)$ and $P_2 = \lambda x.\ \mathbf{cookie}(x)$. We have $P_1 \subseteq P_2$.
  - Moreover, if every cookie is delicious then every choco-cookie is too, in other words, $[\![\textit{all}]\!](P_1)([\![\textit{delicious}]\!]) \Leftarrow [\![\textit{all}]\!](P_2)([\![\textit{delicious}]\!])$.
  - **All is downward monotone w.r.t. its restrictor**.

Quantifier monotonicity: the case of all ($\forall$)

- Let's consider $[\![\textit{all students}]\!] = \lambda P.\ \forall x.\ \textbf{student}(x) \Rightarrow P(x)$. It is the quantifier *all* partially applied to its restrictor (the set of students). Is it monotone w.r.t. its nuclear scope argument?
  - Let's consider $P_1 = \lambda x.\ \textbf{french}(x)$ and $P_2 = \lambda x.\ \textbf{european}(x)$. We have $P_1 \subseteq P_2$.
  - Moreover, if all students are French then all students are European, in other words, $[\![\textit{all students}]\!](P_1) \Rightarrow [\![\textit{all students}]\!](P_2)$.
  - **All is upward monotone w.r.t. is nuclear scope**.
- Let's consider $\lambda P.\ [\![\textit{all}]\!](P)([\![\textit{delicious}]\!]) = \lambda P.\ \forall x.\ P(x) \Rightarrow \textbf{delicious}(x)$. It is the quantifier *all* partially applied to its nuclear scope (the set of delicious things). Is it monotone w.r.t. its restrictor?
  - Let's consider $P_1 = \lambda x.\ \textbf{choco-cookie}(x)$ and $P_2 = \lambda x.\ \textbf{cookie}(x)$. We have $P_1 \subseteq P_2$.
  - Moreover, if every cookie is delicious then every choco-cookie is too, in other words, $[\![\textit{all}]\!](P_1)([\![\textit{delicious}]\!]) \Leftarrow [\![\textit{all}]\!](P_2)([\![\textit{delicious}]\!])$.
  - **All is downward monotone w.r.t. its restrictor**.

Quantifier monotonicity: the case of all ($\forall$)

- Let's consider $[\![\textit{all students}]\!] = \lambda P.\ \forall x.\ \textbf{student}(x) \Rightarrow P(x)$. It is the quantifier *all* partially applied to its restrictor (the set of students). Is it monotone w.r.t. its nuclear scope argument?
  - Let's consider $P_1 = \lambda x.\ \textbf{french}(x)$ and $P_2 = \lambda x.\ \textbf{european}(x)$. We have $P_1 \subseteq P_2$.
  - Moreover, if all students are French then all students are European, in other words, $[\![\textit{all students}]\!](P_1) \Rightarrow [\![\textit{all students}]\!](P_2)$.
  - **All is upward monotone w.r.t. is nuclear scope**.
- Let's consider $\lambda P.\ [\![\textit{all}]\!](P)([\![\textit{delicious}]\!]) = \lambda P.\ \forall x.\ P(x) \Rightarrow \textbf{delicious}(x)$. It is the quantifier *all* partially applied to its nuclear scope (the set of delicious things). Is it monotone w.r.t. its restrictor?
  - Let's consider $P_1 = \lambda x.\ \textbf{choco-cookie}(x)$ and $P_2 = \lambda x.\ \textbf{cookie}(x)$. We have $P_1 \subseteq P_2$.
  - Moreover, if every cookie is delicious then every choco-cookie is too, in other words, $[\![\textit{all}]\!](P_1)([\![\textit{delicious}]\!]) \Leftarrow [\![\textit{all}]\!](P_2)([\![\textit{delicious}]\!])$.
  - **All is downward monotone w.r.t. its restrictor**.

Bonus: quantification in object position, and scope ambiguity

A case of semantic ambiguity

- The sentence:

  (17) Every boy admires some girl.

- Has 2 readings: one in which each boy admires a different girl ("$\forall > \exists$"), and one in which there is a single girl s.t. each boy admires her ("$\exists > \forall$"). How to derive those 2 readings?
- First problem: there is no obvious way of combining the quantified NP *some girl* in the object position to the 2-place predicate *admire*: **type-mismatch**!

- Ideally, we'd like something of type e in place of *some girl*...

A case of semantic ambiguity

- The sentence:

  (18) Every boy admires some girl.

- Has 2 readings: one in which each boy admires a different girl ("$\forall > \exists$"), and one in which there is a single girl s.t. each boy admires her ("$\exists > \forall$"). How to derive those 2 readings?
- First problem: there is no obvious way of combining the quantified NP *some girl* in the object position to the 2-place predicate *admire*: **type-mismatch**!

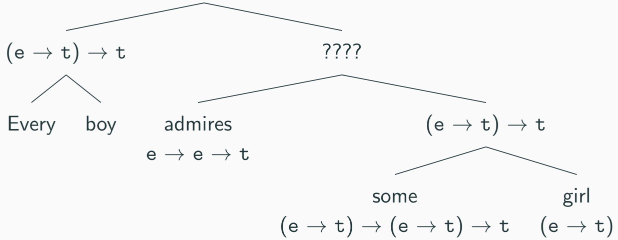
- Ideally, we'd like something of type e in place of *some girl*...

A case of semantic ambiguity

- The sentence:

  (19) Every boy admires some girl.

- Has 2 readings: one in which each boy admires a different girl ("$\forall > \exists$"), and one in which there is a single girl s.t. each boy admires her ("$\exists > \forall$"). How to derive those 2 readings?
- First problem: there is no obvious way of combining the quantified NP *some girl* in the object position to the 2-place predicate *admire*: **type-mismatch**!



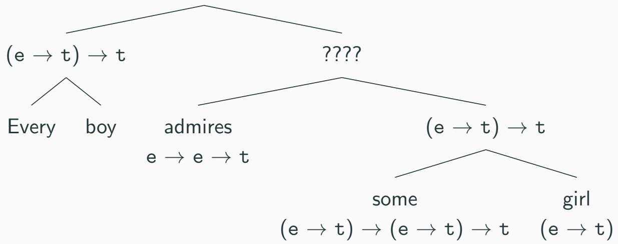
- Ideally, we'd like something of type e in place of *some girl*...

A case of semantic ambiguity

- The sentence:

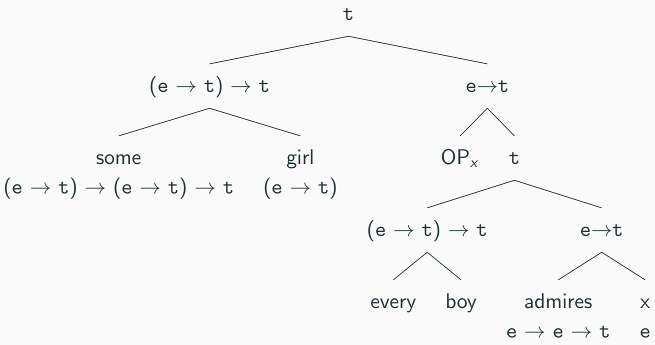
  (20) Every boy admires some girl.

- Has 2 readings: one in which each boy admires a different girl ("$\forall > \exists$"), and one in which there is a single girl s.t. each boy admires her ("$\exists > \forall$"). How to derive those 2 readings?
- First problem: there is no obvious way of combining the quantified NP *some girl* in the object position to the 2-place predicate *admire*: **type-mismatch**!



- Ideally, we'd like something of type e in place of *some girl*...

Resolving type-mismatch, and deriving the $\exists > \forall$ reading

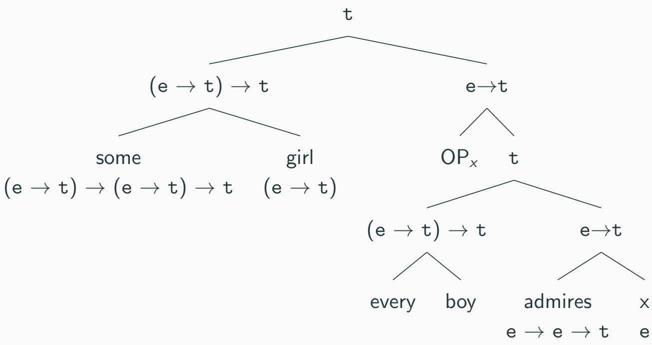


- To resolve the type-mismatch, we moved the quantified NP *some girl* to the top of the tree, and replaced its "trace" by an e-type variable x. We also introduced a λ-abstractor OP$_x$ binding x and changing its input sentence back into a predicate (**type shifting**):[5]

$$[\![\mathrm{OP}_x]\!] = \lambda S.\ \lambda x.\ S$$

[5] This is a very simplified account of binding. A proper account would involve indices and assignment functions.

Resolving type-mismatch, and deriving the $\exists > \forall$ reading

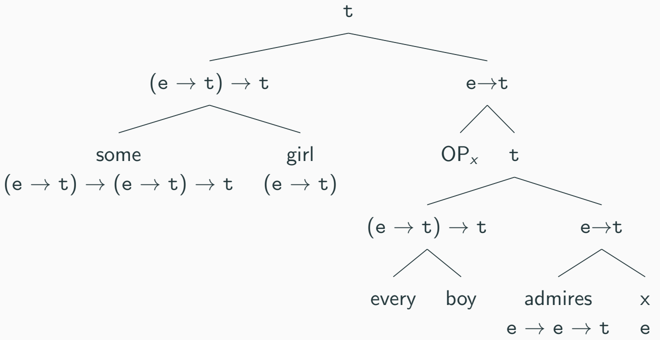


- To resolve the type-mismatch, we moved the quantified NP *some girl* to the top of the tree, and replaced its "trace" by an e-type variable x. We also introduced a λ-abstractor OP_x binding x and changing its input sentence back into a predicate (**type shifting**):[5]

$$[\![\mathrm{OP}_x]\!] = \lambda S.\ \lambda x.\ S$$

[5] This is a very simplified account of binding. A proper account would involve indices and assignment functions.

Resolving type-mismatch, and deriving the $\exists > \forall$ reading



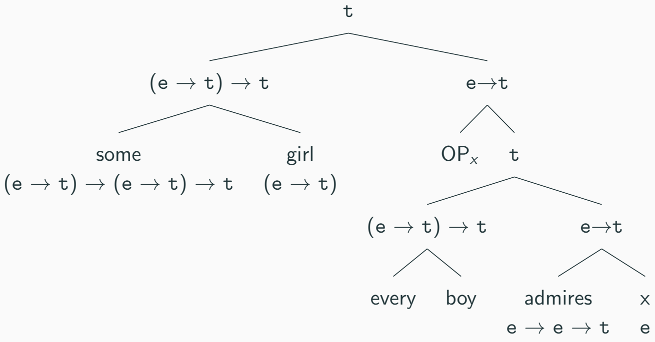
$$[\![\textit{ every boy admires x }]\!] = 1 \text{ iff } \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(x)$$

$$[\![\textit{ OP}_x \textit{ every boy admires x }]\!] = \lambda x.\ \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(x)$$

$$[\![\textit{ Some girl ... admires x }]\!] = 1 \text{ iff } \exists z.\ \mathbf{girl}(z) \wedge (\lambda x.\ \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(x))(z)$$
$$= 1 \text{ iff } \exists z.\ \mathbf{girl}(z) \wedge \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(z)$$

- That is the reading according to which there is one girl that every boy admires. To get the other reading, we need to do one more thing.

Resolving type-mismatch, and deriving the $\exists > \forall$ reading



$$[\![\textit{ every boy admires x }]\!] = 1 \text{ iff } \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(x)$$

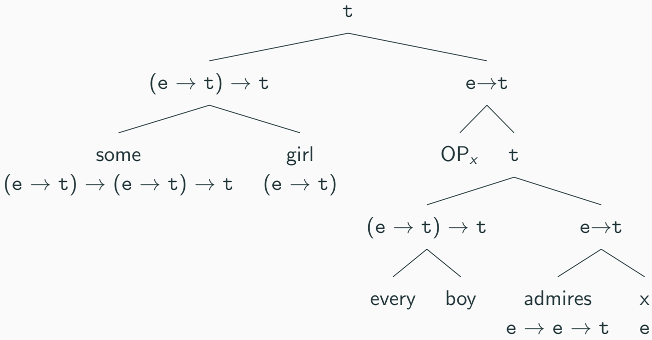
$$[\![\textit{ OP}_x \textit{ every boy admires x }]\!] = \lambda x.\ \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(x)$$

$$[\![\textit{ Some girl ... admires x }]\!] = 1 \text{ iff } \exists z.\ \mathbf{girl}(z) \wedge (\lambda x.\ \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(x))(z)$$

$$= 1 \text{ iff } \exists z.\ \mathbf{girl}(z) \wedge \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(z)$$

- That is the reading according to which there is one girl that every boy admires. To get the other reading, we need to do one more thing.

Resolving type-mismatch, and deriving the $\exists > \forall$ reading



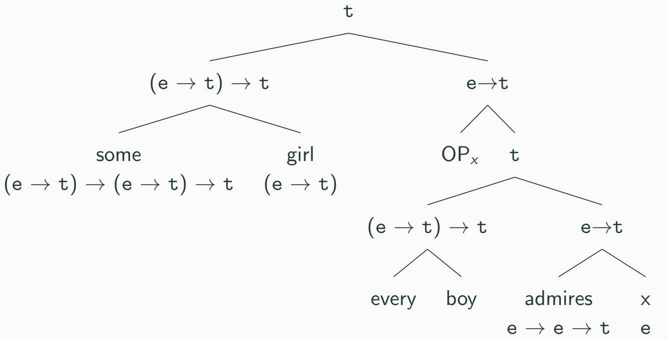
$$[\![\textit{ every boy admires x }]\!] = 1 \text{ iff } \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(x)$$

$$[\![\textit{ OP}_x \textit{ every boy admires x }]\!] = \lambda x.\ \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(x)$$

$$\begin{aligned} [\![\textit{ Some girl ... admires x }]\!] &= 1 \text{ iff } \exists z.\ \mathbf{girl}(z) \wedge (\lambda x.\ \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(x))(z) \\ &= 1 \text{ iff } \exists z.\ \mathbf{girl}(z) \wedge \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(z) \end{aligned}$$

- That is the reading according to which there is one girl that every boy admires. To get the other reading, we need to do one more thing.

Resolving type-mismatch, and deriving the $\exists > \forall$ reading



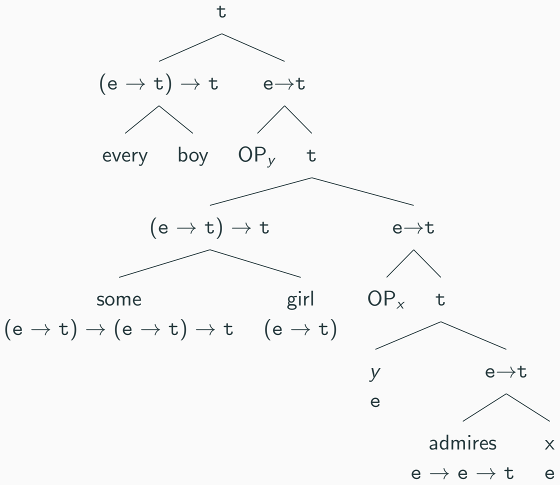
$$[\![\ \textit{every boy admires x}\ ]\!] = 1 \text{ iff } \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(x)$$

$$[\![\ \textit{OP}_x \textit{ every boy admires x}\ ]\!] = \lambda x.\ \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(x)$$

$$[\![\ \textit{Some girl ... admires x}\ ]\!] = 1 \text{ iff } \exists z.\ \mathbf{girl}(z) \wedge (\lambda x.\ \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(x))(z)$$
$$= 1 \text{ iff } \exists z.\ \mathbf{girl}(z) \wedge \forall y.\ \mathbf{boy}(y) \Rightarrow \mathbf{admire}(y)(z)$$

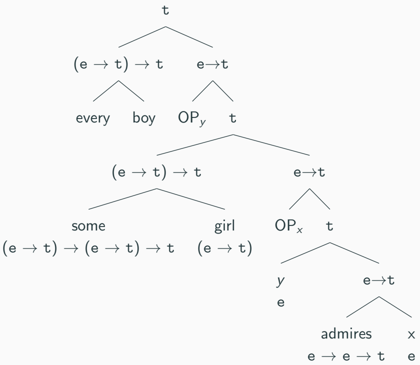
- That is the reading according to which there is one girl that every boy admires. To get the other reading, we need to do one more thing.

Resolving type-mismatch, and deriving the $\forall > \exists$ reading



- We have now moved *every boy* to the top of the tree (above *some girl*) and replaced its "trace" by an e-type variable y bound by an abstractor OP_y...

Quantification in object position: the $\forall > \exists$ reading



$$[\![y \textit{ admires } x]\!] = 1 \text{ iff } \textbf{admires}(y)(x)$$

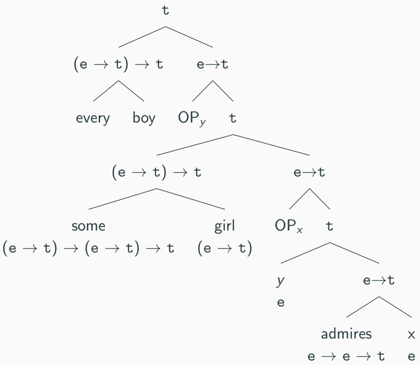
$$[\![\text{OP}_x \; y \textit{ admires } x]\!] = \lambda x.\ \textbf{admires}(y)(x)$$

$$[\![\textit{some girl} \ldots \textit{ admires } x]\!] = 1 \text{ iff } \exists x.\ \textbf{girl}(x) \land \textbf{admires}(y)(x)$$

$$[\![OP_y \textit{ some girl} \ldots \textit{ admires } x]\!] = \lambda y.\ \exists x.\ \textbf{girl}(x) \land \textbf{admires}(y)(x)$$

$$[\![\textit{every boy} \ldots \textit{ admires } x]\!] = 1 \text{ iff } \forall y.\ \textbf{boy}(y) \Rightarrow \exists x.\ \textbf{girl}(x) \land \textbf{admires}(y)(x)$$

Quantification in object position: the $\forall > \exists$ reading



$$[\![y \textit{ admires } x]\!] = 1 \text{ iff } \mathbf{admires}(y)(x)$$

$$[\![\mathrm{OP}_x\ y \textit{ admires } x]\!] = \lambda x.\ \mathbf{admires}(y)(x)$$

$$[\![\textit{some girl ... admires } x]\!] = 1 \text{ iff } \exists x.\ \mathbf{girl}(x) \wedge \mathbf{admires}(y)(x)$$

$$[\![\mathit{OP}_y \textit{ some girl ... admires } x]\!] = \lambda y.\ \exists x.\ \mathbf{girl}(x) \wedge \mathbf{admires}(y)(x)$$

$$[\![\textit{every boy ... admires } x]\!] = 1 \text{ iff } \forall y.\ \mathbf{boy}(y) \Rightarrow \exists x.\ \mathbf{girl}(x) \wedge \mathbf{admires}(y)(x)$$

Take away

- We derived the desired semantic scope ambiguity by moving the quantified NPs to the top of the tree. This is known as **quantifier raising** (QR). Semantic ambiguity was thus cashed out as some form of structural ambiguity in the tree.
- This might sound fishy, especially given that this kind of movement is not *audible*, and that the quantified NPs do not have the same type as their traces ($(\mathtt{e}\rightarrow\mathtt{t})\rightarrow\mathtt{t}$ vs. $\mathtt{e}$).
- However, recall QR was originally motivated by a type issue posed by the quantifier *some girl* interpreted in the object position.
- There might be other solutions to this puzzle, in particular solutions making use of covert type-shifting operators instead of movement. But the analysis we gave here is widely accepted and remains relatively tractable.

Take away

- We derived the desired semantic scope ambiguity by moving the quantified NPs to the top of the tree. This is known as **quantifier raising** (QR). Semantic ambiguity was thus cashed out as some form of structural ambiguity in the tree.
- This might sound fishy, especially given that this kind of movement is not *audible*, and that the quantified NPs do not have the same type as their traces ($(\mathtt{e}\rightarrow\mathtt{t})\rightarrow\mathtt{t}$ vs. $\mathtt{e}$).
- However, recall QR was originally motivated by a type issue posed by the quantifier *some girl* interpreted in the object position.
- There might be other solutions to this puzzle, in particular solutions making use of covert type-shifting operators instead of movement. But the analysis we gave here is widely accepted and remains relatively tractable.

Take away

- We derived the desired semantic scope ambiguity by moving the quantified NPs to the top of the tree. This is known as **quantifier raising** (QR). Semantic ambiguity was thus cashed out as some form of structural ambiguity in the tree.
- This might sound fishy, especially given that this kind of movement is not *audible*, and that the quantified NPs do not have the same type as their traces ($(\texttt{e}\rightarrow\texttt{t})\rightarrow\texttt{t}$ vs. $\texttt{e}$).
- However, recall QR was originally motivated by a type issue posed by the quantifier *some girl* interpreted in the object position.
- There might be other solutions to this puzzle, in particular solutions making use of covert type-shifting operators instead of movement. But the analysis we gave here is widely accepted and remains relatively tractable.

Take away

- We derived the desired semantic scope ambiguity by moving the quantified NPs to the top of the tree. This is known as **quantifier raising** (QR). Semantic ambiguity was thus cashed out as some form of structural ambiguity in the tree.
- This might sound fishy, especially given that this kind of movement is not *audible*, and that the quantified NPs do not have the same type as their traces ($(\mathtt{e} \rightarrow \mathtt{t}) \rightarrow \mathtt{t}$ vs. $\mathtt{e}$).
- However, recall QR was originally motivated by a type issue posed by the quantifier *some girl* interpreted in the object position.
- There might be other solutions to this puzzle, in particular solutions making use of covert type-shifting operators instead of movement. But the analysis we gave here is widely accepted and remains relatively tractable.

References i

Allen, M. R. (1978).
Morphological investigations.
PhD thesis, University of Connecticut.

Kamp, H. (1973).
Free choice permission.
Proceedings of the Aristotelian Society, 74(1):57–74.

Pustejovsky, J. (1995).
The Generative Lexicon.
MIT Press, Cambridge, MA.

Quine, W. V. (1956).
Quantifiers and propositional attitudes.
The Journal of Philosophy, 53(5):177.

von Fintel, K. and Iatridou, S. (2007).
Anatomy of a modal construction.
Linguistic Inquiry, 38(3):445–483.