NLP, Philosophy, and Logic

Jan van Eijck
current affiliation: NIAS, Wassenaar
jve@cwi.nl

NLP Course, 11 December 2006

Abstract

In this tutorial, the meaning of natural language is analysed along the lines proposed by Gottlob Frege and Richard Montague. In building meaning representations, we assume that the meaning of a complex expression derives from the meanings of its components. Typed logic is a convenient tool to make this process of composition explicit. Typed logic allows for the building of semantic representations for formal languages and fragments of natural language in a compositional way. The tutorial ends with the discussion of an example fragment, implemented in the functional programming language Haskell Haskell Team; Jones [2003].

A Philosophical Puzzle about Meaning

Example from Rohit Parikh Parikh [2006].

A Philosophical Puzzle about Meaning

Example from Rohit Parikh Parikh [2006].

- Suppose a thermostat is designed to turn on the heating when the room temperature falls below 18°C. Now 18°C is also 50°F.

A Philosophical Puzzle about Meaning

Example from Rohit Parikh Parikh [2006].

- Suppose a thermostat is designed to turn on the heating when the room temperature falls below 18°C. Now 18°C is also 50°F.
- Suppose the thermostat is taken to the US. Will it have to be told that 18°C is 50°F? No, of course not. It will turn on the heat when the room temperature falls below 50°F.

A Philosophical Puzzle about Meaning

Example from Rohit Parikh Parikh [2006].

- Suppose a thermostat is designed to turn on the heating when the room temperature falls below 18°C. Now 18°C is also 50°F.
- Suppose the thermostat is taken to the US. Will it have to be told that 18°C is 50°F? No, of course not. It will turn on the heat when the room temperature falls below 50°F.
- When listening to Nature, we use propositions rather than sentences. When the temperature is 32°F we feel cold, and when the temperature is 0°C we also feel cold.

A Philosophical Puzzle about Meaning

Example from Rohit Parikh Parikh [2006].

- Suppose a thermostat is designed to turn on the heating when the room temperature falls below 18°C. Now 18°C is also 50°F.
- Suppose the thermostat is taken to the US. Will it have to be told that 18°C is 50°F? No, of course not. It will turn on the heat when the room temperature falls below 50°F.
- When listening to Nature, we use propositions rather than sentences. When the temperature is 32°F we feel cold, and when the temperature is 0°C we also feel cold.
- But depending on our mother tongue, we will utter different sentences.

A Philosophical Puzzle about Meaning (ctd)

A Philosophical Puzzle about Meaning (ctd)

- Now suppose we only have a thermometer, and we have instructed a person with the following sentence: “Turn on the heat when the temperature falls below 18°C.” If the thermometer has a Fahrenheit scale, will she turn on the heat if it shows 50°F?

A Philosophical Puzzle about Meaning (ctd)

- Now suppose we only have a thermometer, and we have instructed a person with the following sentence: "Turn on the heat when the temperature falls below 18°C." If the thermometer has a Fahrenheit scale, will she turn on the heat if it shows 50°F?
- It looks like our beliefs involve both propositions (sets of possible worlds) and sentences (in some natural or formal language).

A Philosophical Puzzle about Meaning (ctd)

- Now suppose we only have a thermometer, and we have instructed a person with the following sentence: "Turn on the heat when the temperature falls below 18°C." If the thermometer has a Fahrenheit scale, will she turn on the heat if it shows 50°F?
- It looks like our beliefs involve both propositions (sets of possible worlds) and sentences (in some natural or formal language).
- How are sentences connected to propositions?

A Philosophical Puzzle about Meaning (ctd)

- Now suppose we only have a thermometer, and we have instructed a person with the following sentence: "Turn on the heat when the temperature falls below 18°C." If the thermometer has a Fahrenheit scale, will she turn on the heat if it shows 50°F?
- It looks like our beliefs involve both propositions (sets of possible worlds) and sentences (in some natural or formal language).
- How are sentences connected to propositions?
- Meaning representations for natural language must involve propositions and equivalence relations on sentences, and their interrelation.

A Philosophical Puzzle about Meaning (ctd)

- Now suppose we only have a thermometer, and we have instructed a person with the following sentence: “Turn on the heat when the temperature falls below 18°C.” If the thermometer has a Fahrenheit scale, will she turn on the heat if it shows 50°F?
- It looks like our beliefs involve both propositions (sets of possible worlds) and sentences (in some natural or formal language).
- How are sentences connected to propositions?
- Meaning representations for natural language must involve propositions and equivalence relations on sentences, and their interrelation.
- Many different proposals for how this works . . .

A Philosophical Puzzle about Meaning (ctd)

- Now suppose we only have a thermometer, and we have instructed a person with the following sentence: "Turn on the heat when the temperature falls below 18°C." If the thermometer has a Fahrenheit scale, will she turn on the heat if it shows 50°F?
- It looks like our beliefs involve both propositions (sets of possible worlds) and sentences (in some natural or formal language).
- How are sentences connected to propositions?
- Meaning representations for natural language must involve propositions and equivalence relations on sentences, and their interrelation.
- Many different proposals for how this works ...
- Fortunately, philosophical problems can be put 'on hold' ...

The Use of Logic: Checking for Consistency

The Use of Logic: Checking for Consistency

- Consider the following set of sentences:

  John likes all girls. Some girl likes everybody but John. Mary likes only John. No boy except John likes every girl.

The Use of Logic: Checking for Consistency

- Consider the following set of sentences:

  John likes all girls. Some girl likes everybody but John. Mary likes only John. No boy except John likes every girl.

- How do we check whether this piece of text is consistent?

The Use of Logic: Checking for Consistency

- Consider the following set of sentences:

  John likes all girls. Some girl likes everybody but John. Mary likes only John. No boy except John likes every girl.

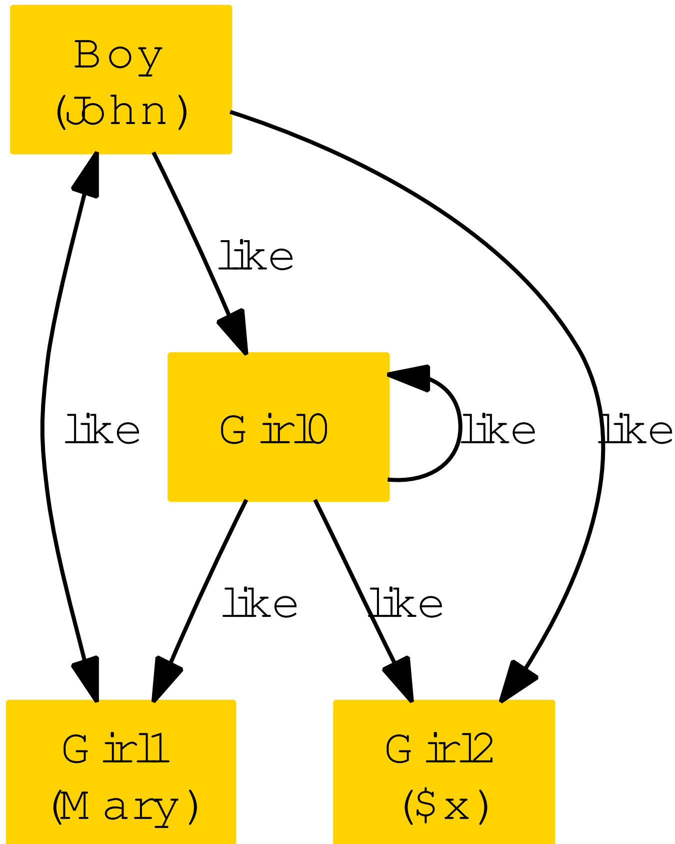
- How do we check whether this piece of text is consistent?
- One possibility: translate the sentences into predicate logic and use an analyzer for predicate logic, e.g. Alloy MIT Software Design Group; Jackson [2006].

First Order Analysis

```
abstract sig Person { like: set Person }
sig Boy extends Person {}
one sig John in Boy {}
sig Girl extends Person {}
one sig Mary in Girl {}

fact likesAndDislikes {
   all x: Girl | x in like[John]
   some x: Girl | all y: Person | y != John => y in like[x]
   all x: Person | x in like[Mary] <=> x = John
   no x: Boy - John | all y: Girl | y in like[x]
}
run {} for exactly 4 Person
```

Example Model



The Use of Logic: Checking for Consequence

The Use of Logic: Checking for Consequence

- Suppose we want to know whether it follows from the story that there is only one boy.

The Use of Logic: Checking for Consequence

- Suppose we want to know whether it follows from the story that there is only one boy.
- Again we can do a first order analysis.

The Use of Logic: Checking for Consequence

- Suppose we want to know whether it follows from the story that there is only one boy.
- Again we can do a first order analysis.
- Using Alloy, we can check for this consequence, as follows:

```
onlyJohn: check { all x: Boy | x = John } for 4 Person
```

The Use of Logic: Checking for Consequence

- Suppose we want to know whether it follows from the story that there is only one boy.
- Again we can do a first order analysis.
- Using Alloy, we can check for this consequence, as follows:

```
onlyJohn: check { all x: Boy | x = John } for 4 Person
```

- And . . . , we get a counterexample.

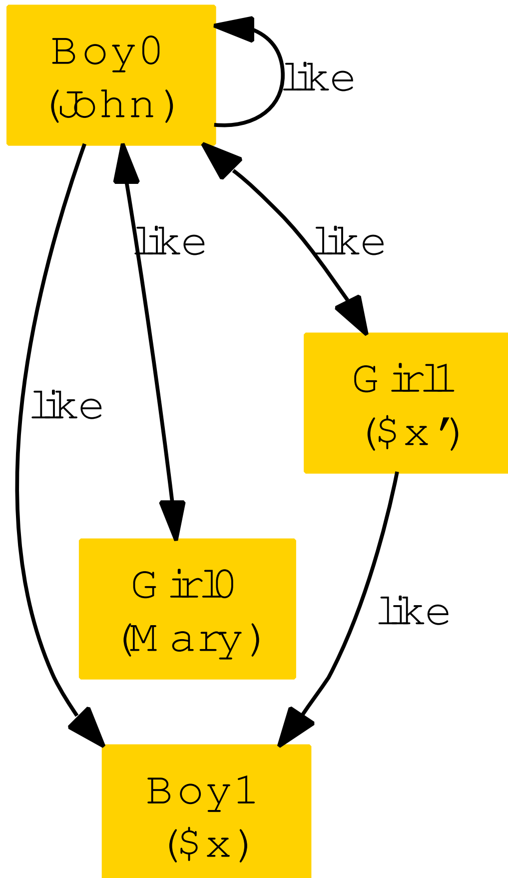
The Use of Logic: Checking for Consequence

- Suppose we want to know whether it follows from the story that there is only one boy.
- Again we can do a first order analysis.
- Using Alloy, we can check for this consequence, as follows:

```
onlyJohn: check { all x: Boy | x = John } for 4 Person
```

- And . . . , we get a counterexample.
- This shows that the text does not entail that there is only one boy.

CounterExample Model



Translation for Meaning Representation: Rules of the Game

Translation for Meaning Representation: Rules of the Game

- An invitation to translate English sentences from an sample of natural language given by some set of grammar rules into meaning representations presupposes two things:
  1. that you understand the meanings of the English sentences;
  2. that you grasp the meanings of the expressions of the representation language.

Translation for Meaning Representation: Rules of the Game

- An invitation to translate English sentences from an sample of natural language given by some set of grammar rules into meaning representations presupposes two things:
  1. that you understand the meanings of the English sentences;
  2. that you grasp the meanings of the expressions of the representation language.
- Knowledge of the second kind can be made fully explicit; semantic truth definitions for the representation languages do the job.

Translation for Meaning Representation: Rules of the Game

- An invitation to translate English sentences from an sample of natural language given by some set of grammar rules into meaning representations presupposes two things:
  1. that you understand the meanings of the English sentences;
  2. that you grasp the meanings of the expressions of the representation language.
- Knowledge of the second kind can be made fully explicit; semantic truth definitions for the representation languages do the job.
- Is it also possible to make knowledge of the meaning of a fragment of natural language fully explicit?

Translation as Explication of Meaning

Translation as Explication of Meaning

- Does a translation procedure from natural language into some kind of logical representation language count as an explication of the concept of meaning for natural language?

Translation as Explication of Meaning

- Does a translation procedure from natural language into some kind of logical representation language count as an explication of the concept of meaning for natural language?
- The procedure should not presuppose knowledge of the meaning of complete natural language sentences, but rather should specify how sentence meanings are derived from the meanings of smaller building blocks.

Translation as Explication of Meaning

- Does a translation procedure from natural language into some kind of logical representation language count as an explication of the concept of meaning for natural language?
- The procedure should not presuppose knowledge of the meaning of complete natural language sentences, but rather should specify how sentence meanings are derived from the meanings of smaller building blocks.
- The meanings of complex expressions should be derivable in a systematic fashion from the meanings of the smallest building blocks occurring in those expressions.

Translation as Explication of Meaning

- Does a translation procedure from natural language into some kind of logical representation language count as an explication of the concept of meaning for natural language?
- The procedure should not presuppose knowledge of the meaning of complete natural language sentences, but rather should specify how sentence meanings are derived from the meanings of smaller building blocks.
- The meanings of complex expressions should be derivable in a systematic fashion from the meanings of the smallest building blocks occurring in those expressions.
- The meaning of these smallest building blocks is taken as given.

Where the Mystery Is

Where the Mystery Is

- The real mystery of semantics lies in the way human beings grasp the meanings of single words.

Where the Mystery Is

- The real mystery of semantics lies in the way human beings grasp the meanings of single words.
- This mystery is not explained away by logical analysis.

Where the Mystery Is

- The real mystery of semantics lies in the way human beings grasp the meanings of single words.
- This mystery is not explained away by logical analysis.
- Logic is a manifestation of the Power of Pure Intelligence rather than an explication of it.

Where the Mystery Is

- The real mystery of semantics lies in the way human beings grasp the meanings of single words.
- This mystery is not explained away by logical analysis.
- Logic is a manifestation of the Power of Pure Intelligence rather than an explication of it.
- Logic gives us patterns and connections.

Where the Mystery Is

- The real mystery of semantics lies in the way human beings grasp the meanings of single words.
- This mystery is not explained away by logical analysis.
- Logic is a manifestation of the Power of Pure Intelligence rather than an explication of it.
- Logic gives us patterns and connections.
- Logic is simple. This simplicity comes at a price. It is achieved by abstracting away from what is difficult.

The Principle of Compositionality

The Principle of Compositionality

- Formal semantics has little or nothing to say about the interpretation of semantic atoms.

The Principle of Compositionality

- Formal semantics has little or nothing to say about the interpretation of semantic atoms.
- It has rather a lot to say about the process of composing complex meanings in a systematic way out of the meanings of components.

The Principle of Compositionality

- Formal semantics has little or nothing to say about the interpretation of semantic atoms.
- It has rather a lot to say about the process of composing complex meanings in a systematic way out of the meanings of components.
- The intuition that this is always possible can be stated somewhat more precisely; it is called the Principle of Compositionality:

  > The meaning of an expression is a function of the meanings of its immediate syntactic components plus their syntactic mode of composition.

The Principle of Compositionality

- Formal semantics has little or nothing to say about the interpretation of semantic atoms.

- It has rather a lot to say about the process of composing complex meanings in a systematic way out of the meanings of components.

- The intuition that this is always possible can be stated somewhat more precisely; it is called the Principle of Compositionality:

  > The meaning of an expression is a function of the meanings of its immediate syntactic components plus their syntactic mode of composition.

- The principle of compositionality is implicit in Gottlob Frege's writings on philosophy of language Frege [1892]; it has been made fully explicit in Richard Montague's approach to natural language semantics.

Misleading Form and Logical Form

‘I see nobody on the road,’ said Alice. ‘I only wish I had such eyes,’ the King remarked in a fretful tone. ‘To be able to see Nobody! And at that distance too!’

Lewis Carroll, Alice in Wonderland.

Misleading Form and Logical Form

> ‘I see nobody on the road,’ said Alice. ‘I only wish I had such eyes,’ the King remarked in a fretful tone. ‘To be able to see Nobody! And at that distance too!’
>
> Lewis Carroll, Alice in Wonderland.

- From Alice walked on the road it follows that someone walked.

Misleading Form and Logical Form

'I see nobody on the road,' said Alice. 'I only wish I had such eyes,' the King remarked in a fretful tone. 'To be able to see Nobody! And at that distance too!'

Lewis Carroll, Alice in Wonderland.

- From Alice walked on the road it follows that someone walked.
- From Nobody walked on the road it does not follow that someone walked.

No Quantified Constituents

No Quantified Constituents

- The logical translation of Some king saw nobody on the road does not reveal constituents corresponding to the quantified subject and object noun phrases.

  $\exists x(\text{king } x \wedge \neg \exists y(\text{person\_on\_the\_road } y \wedge x \text{ saw } y))$.

No Quantified Constituents

- The logical translation of Some king saw nobody on the road does not reveal constituents corresponding to the quantified subject and object noun phrases.

  $\exists x(\text{king } x \wedge \neg\exists y(\text{person\_on\_the\_road } y \wedge x \text{ saw } y))$.

- In the logical translation, quantified expressions seemed to have disappeared.

No Quantified Constituents

- The logical translation of Some king saw nobody on the road does not reveal constituents corresponding to the quantified subject and object noun phrases.

  $\exists x(\textsf{king}\ x \wedge \neg\exists y(\textsf{person\_on\_the\_road}\ y \wedge x\ \textsf{saw}\ y))$.

- In the logical translation, quantified expressions seemed to have disappeared.

- Frege remarks that a quantified expression like every unmarried man or nobody does not give rise to a concept by itself (eine selbstandige Vorstellung), but can only be interpreted in the context of the translation of the whole sentence.

Lambda Notation; Type Theory

Lambda Notation; Type Theory

- According to Frege, the meaning of John arrived can be represented by a function argument expression Aj where A denotes a function and j an argument to that function.

Lambda Notation; Type Theory

- According to Frege, the meaning of John arrived can be represented by a function argument expression Aj where A denotes a function and j an argument to that function.

- Strictly speaking the expression A does not reveal that it is supposed to combine with an individual term to form a formula (an expression denoting a truth value).

Lambda Notation; Type Theory

- According to Frege, the meaning of John arrived can be represented by a function argument expression Aj where A denotes a function and j an argument to that function.
- Strictly speaking the expression A does not reveal that it is supposed to combine with an individual term to form a formula (an expression denoting a truth value).
- One way to make this explicit is by means of lambda notation. The function expression of this example is then written as $(\lambda x.Ax)$. It is also possible to be even more explicit, and write

  $\lambda x.(Ax) :: e \rightarrow t$

  to indicate the type of the expression, or even:

  $(\lambda x_e.A_{e\rightarrow t}x)_{e\rightarrow t}$.

  The subscripts reveal the types.

Types

Types

- The set of types over e, t is given by the following BNF rule:

$$T_1, T_2 \ ::= \ e \mid t \mid (T_1 \rightarrow T_2).$$

Types

- The set of types over e, t is given by the following BNF rule:

$$T_1, T_2 \;::=\; e \mid t \mid (T_1 \rightarrow T_2).$$

- The basic type e is the type of expressions denoting individual objects (or entities). The basic type t is the type of formulas (of expressions which denote truth values). Complex types are the types of functions.

Types

- The set of types over e, t is given by the following BNF rule:

$$T_1, T_2 \ ::= \ e \mid t \mid (T_1 \to T_2).$$

- The basic type e is the type of expressions denoting individual objects (or entities). The basic type t is the type of formulas (of expressions which denote truth values). Complex types are the types of functions.

- For example, $(e \to t)$ or $e \to t$ (we assume that $\to$ is right-associative, and leave out parentheses when this does not create ambiguity) is the type of functions from entities to truth values.

Types

- The set of types over e, t is given by the following BNF rule:

$$T_1, T_2 \ ::= \ e \mid t \mid (T_1 \rightarrow T_2).$$

- The basic type e is the type of expressions denoting individual objects (or entities). The basic type t is the type of formulas (of expressions which denote truth values). Complex types are the types of functions.

- For example, $(e \rightarrow t)$ or $e \rightarrow t$ (we assume that $\rightarrow$ is right-associative, and leave out parentheses when this does not create ambiguity) is the type of functions from entities to truth values.

- In general: $T_1 \rightarrow T_2$ is the type of expressions denoting functions from denotations of T_1 expressions to denotations of T_2 expressions.

The Hierarchy of Typed Domains

The Hierarchy of Typed Domains

- The types e and t are given.

The Hierarchy of Typed Domains

- The types e and t are given.
- Individual objects or entities are objects taken from some domain of discussion D, so e type expressions denote objects in D.

The Hierarchy of Typed Domains

- The types e and t are given.
- Individual objects or entities are objects taken from some domain of discussion D, so e type expressions denote objects in D.
- The truth values are $\{0, 1\}$, so type t expression denotes values in $\{0, 1\}$.

The Hierarchy of Typed Domains

- The types e and t are given.
- Individual objects or entities are objects taken from some domain of discussion D, so e type expressions denote objects in D.
- The truth values are $\{0, 1\}$, so type t expression denotes values in $\{0, 1\}$.
- For complex types we use recursion.

The Hierarchy of Typed Domains

- The types e and t are given.
- Individual objects or entities are objects taken from some domain of discussion D, so e type expressions denote objects in D.
- The truth values are $\{0,1\}$, so type t expression denotes values in $\{0,1\}$.
- For complex types we use recursion.
- This gives:

$$D_e = D, D_t = \{0,1\}, D_{A\to B} = {D_B}^{D_A}.$$

Here ${D_B}^{D_A}$ denotes the set of all functions from D_A to D_B.

Exercise 1 *Let a set $D_e = \{a,b,c\}$ be given. Draw a picture of an element of $D_{e\to t}$.*

Characteristic Functions

A function with range $\{0, 1\}$ is called a characteristic function, because it characterizes a set (namely, the set of those things which get mapped to 1). If T is some arbitrary type, then any member of $D_{T \to t}$ is a characteristic function.

Characteristic Functions

A function with range $\{0, 1\}$ is called a characteristic function, because it characterizes a set (namely, the set of those things which get mapped to 1). If T is some arbitrary type, then any member of $D_{T \to t}$ is a characteristic function.

- The members of $D_{e \to t}$, for instance, characterize subsets of the domain of individuals D_e.

Characteristic Functions

A function with range $\{0, 1\}$ is called a characteristic function, because it characterizes a set (namely, the set of those things which get mapped to 1). If T is some arbitrary type, then any member of $D_{T \to t}$ is a characteristic function.

- The members of $D_{e \to t}$, for instance, characterize subsets of the domain of individuals D_e.
- As another example, consider $D_{(e \to t) \to t}$. According to the type definition this is the domain of functions ${D_t}^{D_{e \to t}}$, i.e., the functions in $\{0, 1\}^{D_{e \to t}}$. These functions characterize sets of subsets of the domain of individuals D_e.

Characteristic Functions

A function with range $\{0, 1\}$ is called a characteristic function, because it characterizes a set (namely, the set of those things which get mapped to 1). If T is some arbitrary type, then any member of $D_{T \to t}$ is a characteristic function.

- The members of $D_{e \to t}$, for instance, characterize subsets of the domain of individuals D_e.
- As another example, consider $D_{(e \to t) \to t}$. According to the type definition this is the domain of functions ${D_t}^{D_{e \to t}}$, i.e., the functions in $\{0, 1\}^{D_{e \to t}}$. These functions characterize sets of subsets of the domain of individuals D_e.

Exercise 2 *Let a set $D_e = \{a, b, c\}$ be given. Draw a picture of an element of $D_{(e \to t) \to t}$.*

The Domain $D_{e \rightarrow e \rightarrow t}$

The Domain $D_{e \to e \to t}$

- As a next example, consider the domain $D_{e \to e \to t}$. This is shorthand for $D_{e \to (e \to t)}$. Assume for simplicity that D_e is the set $\{a, b, c\}$. Then we have:

$$D_{e \to e \to t} = {D_{e \to t}}^{D_e} = (D_t^{D_e})^{D^e} = (\{0,1\}^{\{a,b,c\}})^{\{a,b,c\}}.$$

The Domain $D_{e\to e\to t}$

- As a next example, consider the domain $D_{e\to e\to t}$. This is shorthand for $D_{e\to(e\to t)}$. Assume for simplicity that D_e is the set $\{a,b,c\}$. Then we have:

$$D_{e\to e\to t} = {D_{e\to t}}^{D_e} = (D_t^{D_e})^{D^e} = (\{0,1\}^{\{a,b,c\}})^{\{a,b,c\}}.$$

- A picture of a single element of $D_{e \to e \to t}$.

$$
\begin{aligned}
a &\mapsto \begin{pmatrix} a \mapsto 1 \\ b \mapsto 0 \\ c \mapsto 0 \end{pmatrix} \\
b &\mapsto \begin{pmatrix} a \mapsto 0 \\ b \mapsto 1 \\ c \mapsto 0 \end{pmatrix} \\
c &\mapsto \begin{pmatrix} a \mapsto 0 \\ b \mapsto 1 \\ c \mapsto 1 \end{pmatrix}
\end{aligned}
$$

- A picture of a single element of $D_{e \to e \to t}$.

$$
\begin{aligned}
a &\mapsto \begin{pmatrix} a \mapsto 1 \\ b \mapsto 0 \\ c \mapsto 0 \end{pmatrix} \\
b &\mapsto \begin{pmatrix} a \mapsto 0 \\ b \mapsto 1 \\ c \mapsto 0 \end{pmatrix} \\
c &\mapsto \begin{pmatrix} a \mapsto 0 \\ b \mapsto 1 \\ c \mapsto 1 \end{pmatrix}
\end{aligned}
$$

- The elements of $D_{e \to e \to t}$ can in fact be regarded as functional encodings of two-placed relations R on D_e.

- A picture of a single element of $D_{e \to e \to t}$.

$$
\begin{array}{lcl}
a & \mapsto & \begin{pmatrix} a \mapsto 1 \\ b \mapsto 0 \\ c \mapsto 0 \end{pmatrix} \\
b & \mapsto & \begin{pmatrix} a \mapsto 0 \\ b \mapsto 1 \\ c \mapsto 0 \end{pmatrix} \\
c & \mapsto & \begin{pmatrix} a \mapsto 0 \\ b \mapsto 1 \\ c \mapsto 1 \end{pmatrix}
\end{array}
$$

- The elements of $D_{e \to e \to t}$ can in fact be regarded as functional encodings of two-placed relations R on D_e.

- A function in $D_{e \to e \to t}$ maps every element d of D_e to (the characteristic function of) the set of those elements of D_e to which d has the R-relation, i.e., to the set $\{x \in D_e \mid (d, x) \in R\}$.

The Domain $D_{t \to t}$

The Domain $D_{t \to t}$

- As another example, note that $D_{t \to t}$ has precisely four members, namely:

identity	negation	constant 1	constant 0
$1 \mapsto 1$ $0 \mapsto 0$	$1 \mapsto 0$ $0 \mapsto 1$	$1 \mapsto 1$ $0 \mapsto 1$	$0 \mapsto 0$ $0 \mapsto 0$

The Domain $D_{t \rightarrow t \rightarrow t}$

The Domain $D_{t \to t \to t}$

- The elements of $D_{t \to t \to t}$ are functions from the set of truth values to the functions in $D_{t \to t}$, i.e., to the set of four functions pictured above.

The Domain $D_{t \to t \to t}$

- The elements of $D_{t \to t \to t}$ are functions from the set of truth values to the functions in $D_{t \to t}$, i.e., to the set of four functions pictured above.

- Here is an example, the function which maps 1 to the constant 1 function, and 0 to the identity:

$$
\begin{array}{lll}
1 & \mapsto & \left(\begin{array}{c} 1 \mapsto 1 \\ 0 \mapsto 1 \end{array} \right) \\
0 & \mapsto & \left(\begin{array}{c} 1 \mapsto 1 \\ 0 \mapsto 0 \end{array} \right)
\end{array}
$$

The Domain $D_{t \to t \to t}$

- The elements of $D_{t \to t \to t}$ are functions from the set of truth values to the functions in $D_{t \to t}$, i.e., to the set of four functions pictured above.

- Here is an example, the function which maps 1 to the constant 1 function, and 0 to the identity:

$$\begin{aligned} 1 &\mapsto \begin{pmatrix} 1 \mapsto 1 \\ 0 \mapsto 1 \end{pmatrix} \\ 0 &\mapsto \begin{pmatrix} 1 \mapsto 1 \\ 0 \mapsto 0 \end{pmatrix} \end{aligned}$$

- Note that we can view this as a 'two step' version of the semantic operation of taking a disjunction.

- If the truth value of its first argument is 1, then the disjunction becomes true, and the truth value of the second argument does not matter (hence the constant 1 function).

- If the truth value of its first argument is 1, then the disjunction becomes true, and the truth value of the second argument does not matter (hence the constant 1 function).
- If the truth value of the first argument is 0, then the truth value of the disjunction as a whole is determined by the truth value of the second argument (hence the identity function).

- If the truth value of its first argument is 1, then the disjunction becomes true, and the truth value of the second argument does not matter (hence the constant 1 function).
- If the truth value of the first argument is 0, then the truth value of the disjunction as a whole is determined by the truth value of the second argument (hence the identity function).

Exercise 3 *Specify the conjunction function in* $D_{t \to t \to t}$.

Abstraction and Application

Abstraction and Application

- Now that we know in principle what the type domains D_T look like, for every type T in the type hierarchy, it should be clear that the process of abstraction (creating new functions) brings one higher up in the hierarchy, while the operation of application (applying a function to an argument) brings one down in the hierarchy.

Abstraction and Application

- Now that we know in principle what the type domains D_T look like, for every type T in the type hierarchy, it should be clear that the process of abstraction (creating new functions) brings one higher up in the hierarchy, while the operation of application (applying a function to an argument) brings one down in the hierarchy.

- If Px is an expression of type t and x is of type e, then $(\lambda x.Px)$ is an expression of type $e \to t$. In fact, $(\lambda x.Px)$ is nothing but a more explicit notation for the so-called set abstraction notation $\{x \mid Px\}$.

Abstraction and Application

- Now that we know in principle what the type domains D_T look like, for every type T in the type hierarchy, it should be clear that the process of abstraction (creating new functions) brings one higher up in the hierarchy, while the operation of application (applying a function to an argument) brings one down in the hierarchy.

- If Px is an expression of type t and x is of type e, then $(\lambda x.Px)$ is an expression of type $e \to t$. In fact, $(\lambda x.Px)$ is nothing but a more explicit notation for the so-called set abstraction notation $\{x \mid Px\}$.

- If P is an expression of type $e \to t$ and x is of type e, then (Px) denotes the application of P to x; it is an expression of type t.

Abstraction and Application

- Now that we know in principle what the type domains D_T look like, for every type T in the type hierarchy, it should be clear that the process of abstraction (creating new functions) brings one higher up in the hierarchy, while the operation of application (applying a function to an argument) brings one down in the hierarchy.

- If Px is an expression of type t and x is of type e, then $(\lambda x.Px)$ is an expression of type $e \rightarrow t$. In fact, $(\lambda x.Px)$ is nothing but a more explicit notation for the so-called set abstraction notation $\{x \mid Px\}$.

- If P is an expression of type $e \rightarrow t$ and x is of type e, then (Px) denotes the application of P to x; it is an expression of type t.

- Compositional functional viewpoint: write everything as function application.

Example

The natural language sentence Jan admires the Dalai Lama will get represented as $((Aj)d)$, where j is an argument of the functional expression A, and d in turn is an argument of the functional expression (Aj).

Example

The natural language sentence Jan admires the Dalai Lama will get represented as $((Aj)d)$, where j is an argument of the functional expression A, and d in turn is an argument of the functional expression (Aj).

- A is an expression of type $e \longrightarrow e \longrightarrow t$, and j and d are both of type e.

Example

The natural language sentence Jan admires the Dalai Lama will get represented as $((Aj)d)$, where j is an argument of the functional expression A, and d in turn is an argument of the functional expression (Aj).

- A is an expression of type $e \longrightarrow e \longrightarrow t$, and j and d are both of type e.
- The property of admiring the Dalai Lama is expressed by $(\lambda x.((Ax)d))$

Example

The natural language sentence Jan admires the Dalai Lama will get represented as $((Aj)d)$, where j is an argument of the functional expression A, and d in turn is an argument of the functional expression (Aj).

- A is an expression of type $e \longrightarrow e \longrightarrow t$, and j and d are both of type e.
- The property of admiring the Dalai Lama is expressed by $(\lambda x.((Ax)d))$
- the property of being admired by Jan is expressed by $(\lambda x.((Aj)x))$.

Example

The natural language sentence Jan admires the Dalai Lama will get represented as $((Aj)d)$, where j is an argument of the functional expression A, and d in turn is an argument of the functional expression (Aj).

- A is an expression of type $e \longrightarrow e \longrightarrow t$, and j and d are both of type e.
- The property of admiring the Dalai Lama is expressed by $(\lambda x.((Ax)d))$
- the property of being admired by Jan is expressed by $(\lambda x.((Aj)x))$.
- The property of admiring oneself is expressed by $(\lambda x.((Ax)x))$.

Example

The natural language sentence Jan admires the Dalai Lama will get represented as $((Aj)d)$, where j is an argument of the functional expression A, and d in turn is an argument of the functional expression (Aj).

- A is an expression of type $e \rightarrow e \rightarrow t$, and j and d are both of type e.
- The property of admiring the Dalai Lama is expressed by $(\lambda x.((Ax)d))$
- the property of being admired by Jan is expressed by $(\lambda x.((Aj)x))$.
- The property of admiring oneself is expressed by $(\lambda x.((Ax)x))$.
- The property of being a property of Jan is given by $\lambda X.(Xj)$.

Example

The natural language sentence Jan admires the Dalai Lama will get represented as $((Aj)d)$, where j is an argument of the functional expression A, and d in turn is an argument of the functional expression (Aj).

- A is an expression of type $e \rightarrow e \rightarrow t$, and j and d are both of type e.
- The property of admiring the Dalai Lama is expressed by $(\lambda x.((Ax)d))$
- the property of being admired by Jan is expressed by $(\lambda x.((Aj)x))$.
- The property of admiring oneself is expressed by $(\lambda x.((Ax)x))$.
- The property of being a property of Jan is given by $\lambda X.(Xj)$.
- The property of being a relation between Jan and the Dalai Lama is given by $\lambda Y.((Yj)d)$.

Example

The natural language sentence Jan admires the Dalai Lama will get represented as $((Aj)d)$, where j is an argument of the functional expression A, and d in turn is an argument of the functional expression (Aj).

- A is an expression of type $e \rightarrow e \rightarrow t$, and j and d are both of type e.
- The property of admiring the Dalai Lama is expressed by $(\lambda x.((Ax)d))$
- the property of being admired by Jan is expressed by $(\lambda x.((Aj)x))$.
- The property of admiring oneself is expressed by $(\lambda x.((Ax)x))$.
- The property of being a property of Jan is given by $\lambda X.(Xj)$.
- The property of being a relation between Jan and the Dalai Lama is given by $\lambda Y.((Yj)d)$.

- If X in $\lambda X.(Xj)$ is of type $e \longrightarrow t$, then the expression $\lambda X.(Xj)$ has type $(e \longrightarrow t) \longrightarrow t$.

- If X in $\lambda X.(Xj)$ is of type $e \rightarrow t$, then the expression $\lambda X.(Xj)$ has type $(e \rightarrow t) \rightarrow t$.
- Y in $\lambda Y.((Yj)d)$ should have type $e \rightarrow e \rightarrow t$, so expression $\lambda Y.((Yj)d)$ itself has type $(e \rightarrow e \rightarrow t) \rightarrow t$.

Type Logic as Solution to the Misleading Form Problem

Type Logic as Solution to the Misleading Form Problem

- Natural language constituents correspond to typed expressions that combine with one another as functions and arguments.

Type Logic as Solution to the Misleading Form Problem

- Natural language constituents correspond to typed expressions that combine with one another as functions and arguments.
- After full reduction of the results, quantified expressions and other constituents may have been contextually eliminated, but this elimination is a result of the reduction process, not of the supposed misleading form of the original natural language sentence.

Type Logic as Solution to the Misleading Form Problem

- Natural language constituents correspond to typed expressions that combine with one another as functions and arguments.
- After full reduction of the results, quantified expressions and other constituents may have been contextually eliminated, but this elimination is a result of the reduction process, not of the supposed misleading form of the original natural language sentence.
- While fully reduced logical translations of natural language sentences may be misleading in some sense, the fully unreduced original expressions are not.

Type Logic as Solution to the Misleading Form Problem

- Natural language constituents correspond to typed expressions that combine with one another as functions and arguments.
- After full reduction of the results, quantified expressions and other constituents may have been contextually eliminated, but this elimination is a result of the reduction process, not of the supposed misleading form of the original natural language sentence.
- While fully reduced logical translations of natural language sentences may be misleading in some sense, the fully unreduced original expressions are not.
- Consider the logic of the combination of subjects and predicates. In the simplest cases (John walked) one could say that the predicate takes the subject as an argument, but this does not work for quantified subjects (nobody walked).

Type Logic as Solution to the Misleading Form Problem

- Natural language constituents correspond to typed expressions that combine with one another as functions and arguments.
- After full reduction of the results, quantified expressions and other constituents may have been contextually eliminated, but this elimination is a result of the reduction process, not of the supposed misleading form of the original natural language sentence.
- While fully reduced logical translations of natural language sentences may be misleading in some sense, the fully unreduced original expressions are not.
- Consider the logic of the combination of subjects and predicates. In the simplest cases (John walked) one could say that the predicate takes the subject as an argument, but this does not work for quantified subjects (nobody walked).

- The subject always takes the predicate as its argument. To make this work for simple subjects we logically raising their status from argument to function.

- The subject always takes the predicate as its argument. To make this work for simple subjects we logically raising their status from argument to function.

- We translate John not as the constant j, but as the expression $(\lambda P.(Pj))$. This expression denotes a function from properties to truth values, so it can take a predicate translation as its argument.

- The subject always takes the predicate as its argument. To make this work for simple subjects we logically raising their status from argument to function.

- We translate John not as the constant j, but as the expression $(\lambda P.(Pj))$. This expression denotes a function from properties to truth values, so it can take a predicate translation as its argument.

- The translation of nobody is of the same type:
  $(\lambda P.\neg\exists x((\text{person } x) \wedge (Px)))$.

- The subject always takes the predicate as its argument. To make this work for simple subjects we logically raising their status from argument to function.

- We translate John not as the constant j, but as the expression $(\lambda P.(Pj))$. This expression denotes a function from properties to truth values, so it can take a predicate translation as its argument.

- The translation of nobody is of the same type:
  $(\lambda P.\neg\exists x((\text{person } x) \wedge (Px)))$.

- Before reduction, the translations of John walked and nobody walked look very similar. These similarities disappear only after both translations have been reduced to their simplest forms.

Meaning in Natural Language

Meaning in Natural Language

- Every syntax rule has a semantic counterpart to specify how the meaning representation of the whole is built from the meaning representations of the components.

Meaning in Natural Language

- Every syntax rule has a semantic counterpart to specify how the meaning representation of the whole is built from the meaning representations of the components.
- X is always used for the meaning of the whole.

Meaning in Natural Language

- Every syntax rule has a semantic counterpart to specify how the meaning representation of the whole is built from the meaning representations of the components.
- X is always used for the meaning of the whole.
- X_n refers to the meaning representation of the n-th daughther.

S	$\longrightarrow$	**NP VP**	X	$\longrightarrow$	$(X_1 X_2)$
NP	$\longrightarrow$	*Mary*	X	$\longrightarrow$	$(\lambda P.(Pm))$
NP	$\longrightarrow$	*Bill*	X	$\longrightarrow$	$(\lambda P.(Pb))$
NP	$\longrightarrow$	**DET CN**	X	$\longrightarrow$	$(X_1 X_2)$
NP	$\longrightarrow$	**DET RCN**	X	$\longrightarrow$	$(X_1 X_2)$
DET	$\longrightarrow$	*every*	X	$\longrightarrow$	$(\lambda P.(\lambda Q.\forall x((Px) \Rightarrow (Qx)))))$
DET	$\longrightarrow$	*some*	X	$\longrightarrow$	$(\lambda P.(\lambda Q.\exists x((Px) \land (Qx)))))$
DET	$\longrightarrow$	*no*	X	$\longrightarrow$	$(\lambda P.(\lambda Q.\forall x((Px) \Rightarrow \neg(Qx)))))$
DET	$\longrightarrow$	*the*	X	$\longrightarrow$	$(\lambda P.(\lambda Q.\exists x(\forall y((Py) \leftrightarrow x = y)$
CN	$\longrightarrow$	*man*	X	$\longrightarrow$	$(\lambda x.(Mx))$
CN	$\longrightarrow$	*woman*	X	$\longrightarrow$	$(\lambda x.(Wx))$
CN	$\longrightarrow$	*boy*	X	$\longrightarrow$	$(\lambda x.(Bx))$
RCN	$\longrightarrow$	**CN** *that* **VP**	X	$\longrightarrow$	$(\lambda x.((X_1\ x) \land (X_3\ x)))$
RCN	$\longrightarrow$	**CN** *that* **NP TV**	X	$\longrightarrow$	$(\lambda x.((X_1\ x) \land (X_3(\lambda y.((X_4\ x)y)$
VP	$\longrightarrow$	*laughed*	X	$\longrightarrow$	$(\lambda x.(Lx))$
VP	$\longrightarrow$	*smiled*	X	$\longrightarrow$	$(\lambda x.(Sx))$
VP	$\longrightarrow$	**TV NP**	X	$\longrightarrow$	$(\lambda x.(X_2\ (\lambda y.((X_1\ x)y))))$
TV	$\longrightarrow$	*loved*	X	$\longrightarrow$	L

Consider the sentence Bill loved Mary:

$$[_S[_{NP}\textit{Bill}\ ][_{VP}[_{TV}\textit{loved}\ ][_{NP}\textit{Mary}\ ]]]$$

According to the rules above, this gets assigned the following meaning:

$$((\lambda P.(Pb))(\lambda x.(\lambda P.(Pm))(\lambda y.Lxy)))$$

Reducing this gives:

$$\xrightarrow{\beta} ((\lambda P.(Pb))(\lambda x.((\lambda y.Lxy)m)))$$

$$\xrightarrow{\beta} ((\lambda P.(Pb))(\lambda x.(Lxm)))$$

$$\xrightarrow{\beta} (\lambda x.Lxm)b$$

$$\xrightarrow{\beta} Lbm$$

Exercise 4 *Give the compositional translation for 'Bill loved some woman', and reduce it to normal form.*

Datastructures for Syntax

Datastructures for Syntax

- It is straightforward to give an implementation of compositional semantics of natural language if we use a programming language that is itself based on type theory.

Datastructures for Syntax

- It is straightforward to give an implementation of compositional semantics of natural language if we use a programming language that is itself based on type theory.
- First we define the data structures for the predicates, the variables, and the formulas, with data declarations for the various syntactic categories.

Datastructures for Syntax

- It is straightforward to give an implementation of compositional semantics of natural language if we use a programming language that is itself based on type theory.
- First we define the data structures for the predicates, the variables, and the formulas, with data declarations for the various syntactic categories.
- The text in the square boxes below is the actual program code.

```
module CM where


import MOTT
import EPLIH
import Domain
import Model
import List
```

```haskell
data Sent = Sent NP VP
     deriving (Eq,Show)

data NP = Ann | Mary | Bill | Johnny | NP1 DET CN
        | NP2 DET RCN
     deriving (Eq,Show)

data DET = Every | Some | No | The | Most
         | Atleast Int
     deriving (Eq,Show)

data CN = Man | Woman | Boy | Person | Thing | House
     deriving (Eq,Show)
```

```
data RCN = CN1 CN VP | CN2 CN NP TV
     deriving (Eq,Show)

data VP = Laughed | Smiled | VP1 TV NP
     deriving (Eq,Show)

data TV = Loved | Respected | Hated | Owned
     deriving (Eq,Show)
```

The suffix `deriving (Eq,Show)` in the data type declarations is the Haskell way to ensure that equality is defined for these data types, and that they can be displayed on the screen.

Semantic Interpretation: Sentences

Next, we define for every syntactic category an interpretation function of the appropriate type, using `Entity` for e and `Bool` for t. The interpretation of sentences has type `Bool`, so the interpretation function `intS` gets type `Sent -> Bool`. Since there is only one rewrite rule for S, the interpretation function `intS` consists of only one equation:

```
intSent :: Sent -> Bool
intSent (Sent np vp) = (intNP np) (intVP vp)
```

Semantic Interpretation: NPs

Semantic Interpretation: NPs

- The interpretation function `intNP` consists of four equations, one for every rewrite rule for NP in the grammar fragment.

Semantic Interpretation: NPs

- The interpretation function `intNP` consists of four equations, one for every rewrite rule for NP in the grammar fragment.
- The function has type `NP -> (Entity -> Bool) -> Bool`.

Semantic Interpretation: NPs

- The interpretation function `intNP` consists of four equations, one for every rewrite rule for NP in the grammar fragment.
- The function has type `NP -> (Entity -> Bool) -> Bool`.
- Here `Entity -> Bool` is the Haskell counterpart to $e \longrightarrow t$, which is the type of the VP interpretation that the NP combines with to form a sentence.

```haskell
intNP :: NP -> (Entity -> Bool) -> Bool
intNP Ann = \ p -> p ann
intNP Mary = \ p -> p mary
intNP Bill = \ p -> p bill
intNP Johnny = \ p -> p johnny
intNP (NP1 det cn) = (intDET det) (intCN cn)
intNP (NP2 det rcn) = (intDET det) (intRCN rcn)
```

Note the close connection between `\ p -> p mary` and $\lambda P.(Pm)$ that we get by employing the Haskell counterpart to λ.

Semantic Interpretation: VPs

For the interpretation of verb phrases we invoke the information encoded in our first order model.

```
intVP :: VP -> Entity -> Bool
intVP Laughed = laugh
intVP Smiled = smile
```

The interpretation of complex VPs is a bit more involved. We have to find a way to make reference to the property of 'standing into the TV relation to the subject of the sentence'. We do this in the same way as in the type logic specification of the semantic clause for $[\mathrm{TV\ NP}]_{\mathrm{VP}}$.

```
intVP (VP1 tv np) =
   \ subj -> intNP np (\ obj -> intTV tv (subj,obj))
```

Note that `subj` refers to the sentence subject and `obj` to the sentence direct object.

Semantic Interpretation: TVs

The interpretation of transitive verbs discloses another bit of information about the world. Again, we invoke the information about the world encoded in our first order model.

```
intTV :: TV -> (Entity,Entity) -> Bool
intTV Loved     = love
intTV Respected = respect
intTV Hated     = hate
intTV Owned     = own
```

Semantic Interpretation: Common Nouns

The interpreation of CNs is similar to that of VPs.

```
intCN :: CN -> Entity -> Bool
intCN Man = man
intCN Boy = boy
intCN Woman = woman
intCN Person = person
intCN Thing = thing
intCN House = house
```

Semantic Interpretation: Determiners

The most involved part of the implementation: the definition of the determiner interpretations. First the type. The interpretation of a DET needs two properties (type `Entity -> Bool`): one for the CN and one for the VP, to yield the type of an S interpretation, i.e., `Bool`.

```
intDET :: DET -> (Entity -> Bool)
              -> (Entity -> Bool) -> Bool
```

The interpretation of Some just checks whether the two properties corresponding to CN and VP have anything in common. This is what this check looks like in Haskell:

```
intDET Some p q = any q (filter p entities)
```

Semantic Interpretation: Determiners

The most involved part of the implementation: the definition of the determiner interpretations. First the type. The interpretation of a DET needs two properties (type `Entity -> Bool`): one for the CN and one for the VP, to yield the type of an S interpretation, i.e., `Bool`.

```
intDET :: DET -> (Entity -> Bool)
              -> (Entity -> Bool) -> Bool
```

The interpretation of Some just checks whether the two properties corresponding to CN and VP have anything in common. This is what this check looks like in Haskell:

```
intDET Some p q = any q (filter p entities)
```

- Here `filter p entities` gives the list of all members of `entities` that satisfy property `p`.

- Here `filter p entities` gives the list of all members of `entities` that satisfy property `p`.
- `entities` gives the domain of entities in the form of a list.

- Here `filter p entities` gives the list of all members of `entities` that satisfy property `p`.
- `entities` gives the domain of entities in the form of a list.
- `any` is a function taking a property and a list that returns `True` if the sublist of elements satisfying the property is non-empty, `False` otherwise.

- Here `filter p entities` gives the list of all members of `entities` that satisfy property `p`.
- `entities` gives the domain of entities in the form of a list.
- `any` is a function taking a property and a list that returns `True` if the sublist of elements satisfying the property is non-empty, `False` otherwise.
- Thus, `any q list` checks whether any element of the list satisfies property `q`.

The interpretation of Every checks whether the CN property is included in the VP property:

```
intDET Every p q = all q (filter p entities)
```

The interpretation of Every checks whether the CN property is included in the VP property:

```
intDET Every p q = all q (filter p entities)
```

- Here `filter p entities` gives the list of all members of `entities` that satisfy property `p`.

The interpretation of Every checks whether the CN property is included in the VP property:

```
intDET Every p q = all q (filter p entities)
```

- Here `filter p entities` gives the list of all members of `entities` that satisfy property `p`.
- `all q list` checks whether every member of `list` satisfies property `q`.

The interpretation of The consists of two parts:

1. a check that the CN property is unique, i.e., that it is true of precisely one entity in the domain,
2. a check that the CN and the VP property have an element in common, in other words, the Some check on the two properties.

```
intDET The p q = singleton plist && q (head plist)
  where
    plist = filter p entities
    singleton [x] = True
    singleton _   = False
```

The interpretation of No is just the negation of the interpretation of Some:

```
intDET No p q = not (intDET Some p q)
```

The interpretation of Most compares the length of the list of entities satisfying both arguments (the restrictor argument and the body argument) with the length of the list of entities satisfying only the restrictor argument.

```
intDET Most p q =
  length pqlist > length (plist \\ qlist)
  where
   plist  = filter p entities
   qlist  = filter q entities
   pqlist = filter q plist
```

The interpretation of Most compares the length of the list of entities satisfying both arguments (the restrictor argument and the body argument) with the length of the list of entities satisfying only the restrictor argument.

```
intDET Most p q =
  length pqlist > length (plist \\ qlist)
  where
   plist  = filter p entities
   qlist  = filter q entities
   pqlist = filter q plist
```

Exercise 5 *Implement the interpretation function for* `(Atleast n)`*.*

Semantic Interpretation: Relativised Common Nouns

The interpretation of relativised common nouns of the form That CN VP checks whether an entity has both the CN and the VP property:

```
intRCN :: RCN -> Entity -> Bool
intRCN (CN1 cn vp) =
  \ e -> ((intCN cn e) && (intVP vp e))
```

The interpretation of relativised common nouns of the form That CN NP TV checks whether an entity has both the CN property as the property of being the object of NP TV.

```
intRCN (CN2 cn np tv) =
   \ e -> ((intCN cn e) &&
            (intNP np
               (\ subj -> (intTV tv (subj,e)))))
```

Examples

```
example1 = intSent (Sent (NP1 The Boy) Smiled)
example2 = intSent (Sent (NP1 The Boy) Laughed)
example3 = intSent (Sent (NP1 Some Man) Laughed)
example4 = intSent (Sent (NP1 No Man) Laughed)
example5 = intSent
 (Sent (NP1 Some Man)(VP1 Loved (NP1 Some Woman)))
example6 = intSent
 (Sent (NP2 No (CN1 Man (VP1 Loved Mary))) Laughed)
```

```
CM> example1
True
CM> example2
False
CM> example3
False
CM> example4
True
CM> example5
True
CM> example6
True
```

It is a bit awkward that we have to provide the datastructures of syntax ourselves. The process of constructing syntax datastructures from strings of words is called parsing; this topic will be taken up in various other tutorials.

Translation into Logical Form

Translation into Logical Form

- One of the insights of Montague grammar is that the presence of a level of logical form is superfluous for model theoretic interpretation.

Translation into Logical Form

- One of the insights of Montague grammar is that the presence of a level of logical form is superfluous for model theoretic interpretation.
- In Montague [1973] and Montague [1974b] a typed language of logical formulas is used as a stepping stone on the way to semantic specification, but in Montague [1974a] the meaning of of a fragment of English is specified without a detour through logical form.

Translation into Logical Form

- One of the insights of Montague grammar is that the presence of a level of logical form is superfluous for model theoretic interpretation.
- In Montague [1973] and Montague [1974b] a typed language of logical formulas is used as a stepping stone on the way to semantic specification, but in Montague [1974a] the meaning of of a fragment of English is specified without a detour through logical form.
- The set-up above is along the lines of Montague [1974a].

Translation into Logical Form

- One of the insights of Montague grammar is that the presence of a level of logical form is superfluous for model theoretic interpretation.
- In Montague [1973] and Montague [1974b] a typed language of logical formulas is used as a stepping stone on the way to semantic specification, but in Montague [1974a] the meaning of of a fragment of English is specified without a detour through logical form.
- The set-up above is along the lines of Montague [1974a].
- We will now demonstrate how translation into logical form can be implemented. We model our logical form language after the language of predicate logic. In particular, we use data types for `Var` and `Term`.

Translation into Logical Form

- One of the insights of Montague grammar is that the presence of a level of logical form is superfluous for model theoretic interpretation.
- In Montague [1973] and Montague [1974b] a typed language of logical formulas is used as a stepping stone on the way to semantic specification, but in Montague [1974a] the meaning of of a fragment of English is specified without a detour through logical form.
- The set-up above is along the lines of Montague [1974a].
- We will now demonstrate how translation into logical form can be implemented. We model our logical form language after the language of predicate logic. In particular, we use data types for `Var` and `Term`.
- Advantage of generating LFs: can also be used for consistency clecking.

Here is a data type for generalized quantifiers.

```
data GQ = ALL | SOME | THE | NO | MOST | ATLEAST Int
          deriving (Show,Eq,Ord)
```

In fact, the only difference between logical forms and formulas of predicate logic is the presence of generalized quantifiers.

```
data LF = Atom1 Name [Term]
        | Eq1 Term Term
        | Neg1 LF
        | Impl1 LF LF
        | Equi1 LF LF
        | Conj1 [LF]
        | Disj1 [LF]
        | Quant GQ Var LF LF
     deriving (Eq,Ord)
```

NOTE: `Term` is defined elsewhere as

```
data Term = Vari Var
          | Struct String [Term] deriving (Eq,Ord)
```

A `show` function for logical forms:

```
instance Show LF where
  show (Atom1 id []) = id
  show (Atom1 id ts) = id ++ concat [ show ts ]
  show (Eq1 t1 t2)   = show t1 ++ "==" ++ show t2
  show (Neg1 form)   = '~': (show form)
  show (Impl1 f1 f2) =
     "(" ++ show f1 ++ "==>" ++ show f2 ++ ")"
  show (Equi1 f1 f2) =
     "(" ++ show f1 ++ "<=>" ++ show f2 ++ ")"
  show (Conj1 []) =  "true"
  show (Conj1 fs) =  "conj" ++ concat [ show fs ]
  show (Disj1 []) =  "false"
  show (Disj1 fs) =  "disj" ++ concat [ show fs ]
  show (Quant gq var f1 f2) =
      show gq ++ " " ++ show var ++
      " (" ++ show f1 ++ "," ++ show f2 ++ ")"
```

Translation to LF: Sentences

The process of translating to logical form is a very easy variation on the interpretation process for syntactic structures. Instead of the type `Bool`, for interpretation in a model, we take `LF`, for a logical form of type t, and instead of the type `Entity` for entities in the model, take `Var`, for variables that are supposed to get mapped to entities.

In the basic cases, we translate proper names into constant terms and lexical CNs, VPs, TVs into atomic formulas.

```
lfSent :: Sent -> LF
lfSent (Sent np vp) = (lfNP np) (lfVP vp)
```

Translation to LF: Noun Phrases

```
lfNP :: NP -> (Term -> LF) -> LF
lfNP Ann  = \ p -> p (Struct "Ann" [])
lfNP Mary = \ p -> p (Struct "Mary" [])
lfNP Bill = \ p -> p (Struct "Bill" [])
lfNP Johnny = \ p -> p (Struct "Johnny" [])
lfNP (NP1 det cn) = (lfDET det) (lfCN cn)
lfNP (NP2 det rcn) = (lfDET det) (lfRCN rcn)
```

Translation to LF: VPs

```
lfVP :: VP -> Term -> LF
lfVP Laughed = \ t -> Atom1 "laugh" [t]
lfVP Smiled  = \ t -> Atom1 "smile" [t]
lfVP (VP1 tv np) =
  \ subj -> lfNP np (\ obj -> lfTV tv (subj,obj))
```

Translation to LF: TVs

```
lfTV :: TV -> (Term,Term) -> LF
lfTV Loved     =
  \ (t1,t2) -> Atom1 "love"    [t1,t2]
lfTV Respected =
  \ (t1,t2) -> Atom1 "respect" [t1,t2]
lfTV Hated     =
  \ (t1,t2) -> Atom1 "hate"    [t1,t2]
lfTV Owned     =
  \ (t1,t2) -> Atom1 "own"     [t1,t2]
```

Translation to LF: CNs

```
lfCN :: CN -> Term -> LF
lfCN Man    = \ t -> Atom1 "man"    [t]
lfCN Boy    = \ t -> Atom1 "boy"    [t]
lfCN Woman  = \ t -> Atom1 "woman"  [t]
lfCN Person = \ t -> Atom1 "person" [t]
lfCN Thing  = \ t -> Atom1 "thing"  [t]
lfCN House  = \ t -> Atom1 "house"  [t]
```

Translation to LF: Determiners

The type of the logical form translation of determiners indicates that the translation takes a determiner phrase and two arguments for objects of type `Term -> LF` (logical forms with term holes in them), and produces a logical form.

```
lfDET :: DET -> (Term -> LF) -> (Term -> LF) -> LF
```

The translation of determiners should be done with some care. It involves the construction of a logical form where a variable gets bound. To ensure proper binding, we have to make sure that the newly introduced variable will not get accidentally bound by a quantifier already present in the logical form. For this, we first list the (free) variables in logical forms.

```
varsInLf :: LF -> [Var]
varsInLf (Atom1 _ ts)  = varsInTerms ts
varsInLf (Eq1 t1 t2)   = varsInTerms [t1,t2]
varsInLf (Neg1 form)   = varsInLf form
varsInLf (Impl1 f1 f2) = varsInLfs [f1,f2]
varsInLf (Equi1 f1 f2) = varsInLfs [f1,f2]
varsInLf (Conj1 fs)    = varsInLfs fs
varsInLf (Disj1 fs)    = varsInLfs fs
varsInLf (Quant gq var f1 f2) =
   varsInLfs [f1,f2] \\ [var]

varsInLfs :: [LF] -> [Var]
varsInLfs = nub . concat . map varsInLf
```

We need the list of variable indices of a list of logical forms in order to compute a fresh variable index. All variables will get introduced by means of this mechanism, so if we start out with variables of the form `Var "x" [0]`, and only introduce new variables of the form `Var "x" [i]`, we can assume that all variables occurring in our logical form will have the shape `Var "x" [i]` for some integer `i`.

```
freshvar :: [LF] -> Var
freshvar lfs = (Var "x" [i+1])
  where
  i = foldr max 0 (xindices (varsInLfs lfs))
  xindices = map (\ (Var "x" [j]) -> j)
```

The definition uses `foldr` for defining the maximum of a list of integers.

The term `zero` is defined elsewhere as a constant. Use this term as a dummy to provisionally fill in the term slot in formulas with term holes in them.

```
lfDET Some p q =
  Quant SOME v (p (Vari v)) (q (Vari v))
   where v = freshvar [p zero,q zero]
lfDET Every p q =
  Quant ALL v (p (Vari v)) (q (Vari v))
   where v = freshvar [p zero,q zero]
lfDET No p q =
  Quant NO v (p (Vari v)) (q (Vari v))
   where v = freshvar [p zero,q zero]
lfDET The p q =
  Quant THE v (p (Vari v)) (q (Vari v))
   where v = freshvar [p zero,q zero]
lfDET Most p q =
  Quant MOST v (p (Vari v)) (q (Vari v))
   where v = freshvar [p zero,q zero]
```

Exercise 6 *Implement the translation function for* `(Atleast n)`.

Translation to LF: Relativized CNs

Use `Conj1` to conjoin the logical form for a common noun and the logical form for a relative clause into a logical form for a complex common noun.

```
lfRCN :: RCN -> Term -> LF
lfRCN (CN1 cn vp) =
   \ t -> Conj1 [lfCN cn t, lfVP vp t]
lfRCN (CN2 cn np tv) =
   \ t -> Conj1 [lfCN cn t,
                 lfNP np (\ subj -> lfTV tv (subj,t))]
```

Examples

```
lf1 = lfSent
 (Sent (NP1 Some Man)
       (VP1 Loved (NP1 Some Woman)))
lf2 = lfSent
  (Sent (NP2 No (CN1 Man (VP1 Loved Mary))) Laughed)
```

```
CM> lf1
SOME x1 (man[x1],SOME x2 (woman[x2],love[x1,x2]))
CM> lf2
NO x1 (conj[man[x1],love[x1,Mary]],laugh[x1])
```

Exercise 7 *Implement an evaluation function for logical forms in appropriate models.*

Conclusions, References

Conclusions, References

- Typed functional languages are natural choice for semantics of natural language.

Conclusions, References

- Typed functional languages are natural choice for semantics of natural language.
- Implementing Montague grammar in Haskell is a breeze.

Conclusions, References

- Typed functional languages are natural choice for semantics of natural language.
- Implementing Montague grammar in Haskell is a breeze.
- For more on Haskell see Doets and van Eijck [2004]

Conclusions, References

- Typed functional languages are natural choice for semantics of natural language.
- Implementing Montague grammar in Haskell is a breeze.
- For more on Haskell see Doets and van Eijck [2004]
- This talk partly based on: Eijck [2000]

Conclusions, References

- Typed functional languages are natural choice for semantics of natural language.
- Implementing Montague grammar in Haskell is a breeze.
- For more on Haskell see Doets and van Eijck [2004]
- This talk partly based on: Eijck [2000]
- Further info on Computational Semantics with Type Theory: Eijck [2004] `http://www.cwi.nl/~jve/cs`

Conclusions, References

- Typed functional languages are natural choice for semantics of natural language.
- Implementing Montague grammar in Haskell is a breeze.
- For more on Haskell see Doets and van Eijck [2004]
- This talk partly based on: Eijck [2000]
- Further info on Computational Semantics with Type Theory: Eijck [2004] http://www.cwi.nl/~jve/cs
- Up-to-date overview of dynamic semantics for natural language, with links to computer science: Eijck and Stokhof [2006].

References

K. Doets and J. van Eijck. The Haskell Road to Logic, Maths and Programming, volume 4 of Texts in Computing. King's College Publications, London, 2004.

J. van Eijck and M. Stokhof. The gamut of dymamic logics. In D.M. Gabbay and J. Woods, editors, The Handbook of the History of Logic, volume 7 — Logic and the Modalities in the Twentieth Century, pages 499–600. Elsevier, 2006.

Jan van Eijck. Tutorial on the composition of meaning. Lecture Note, Uil-OTS, November 2000.

Jan van Eijck. Computational semantics with type theory. Book manuscript, CWI Amsterdam, 2004.

G. Frege. Ueber sinn und bedeutung. Translated as 'On Sense and Reference' in Geach and Black (eds.), Translations from the Philosophical Writings of Gottlob Frege, Blackwell, Oxford (1952), 1892.

The Haskell Team. The Haskell homepage. http://www.haskell.org.

Daniel Jackson. Software Abstractions; Logic, Language and Analysis. MIT Press, 2006.

S. Peyton Jones, editor. Haskell 98 Language and Libraries; The Revised Report. Cambridge University Press, 2003.

MIT Software Design Group. The Alloy Analyzer. http://alloy.mit.edu.

R. Montague. The proper treatment of quantification in ordinary English. In J. Hintikka, editor, Approaches to Natural Language, pages 221–242. Reidel, 1973.

R. Montague. English as a formal language. In R.H. Thomason, editor, *Formal Philosophy; Selected Papers of Richard Montague*, pages 188–221. Yale University Press, New Haven and London, 1974.

R. Montague. Universal grammar. In R.H. Thomason, editor, *Formal Philosophy; Selected Papers of Richard Montague*, pages 222–246. Yale University Press, New Haven and London, 1974.

Rohit Parikh. Sentences, propositions, and beliefs. City University of New York, May 2006.