Computational Semantics
Day 2: Meaning representations and (predicate) logic

Jan van Eijck[1] & Christina Unger[2]

[1]CWI, Amsterdam, and UiL-OTS, Utrecht, The Netherlands
[2]CITEC, Bielefeld University, Germany

ESSLLI 2011, Ljubljana

Goals

Whatever we decide meanings to be, we want:

- a finite way to specify the meanings of the infinite set of sentences, i.e. a recursive procedure to determine the meaning of complex expressions given the meanings of lexical items and a syntactic structure **(compositionality)**
- to capture the relation of a natural language expression and the real world **(modeltheoretic semantics)**
- to capture certain semantic intuitions

Semantic intuitions

- Semantic anomalies (despite syntactic well-formedness)
    - *Colourless green ideas sleep furiously.*
    - *Forty-seven frightened sincerity.*
- Contradictions
    - *It is raining and it is not raining.*
    - *He is a bachelor and merrily married to Mary.*
- Entailments
    - *Speedy Gonzales ran fast.* → *Speedy Gonzalez ran.*
    - *Every human is mortal.* → *Chomsky is mortal.*

Outline

1. Form and Content
2. First-order predicate logic
3. Logical formulas as meaning representations

Form and Content

Form in Haskell: User defined data types

Imagine we want to use types that correspond to syntactic categories like S, NP, VP and capture their internal structure.

Form in Haskell: User defined data types

Imagine we want to use types that correspond to syntactic categories like S, NP, VP and capture their internal structure.

Instead of coding them as a combination of strings, we want to define *structure trees* for them.

Form in Haskell: User defined data types

Imagine we want to use types that correspond to syntactic categories like S, NP, VP and capture their internal structure.

Instead of coding them as a combination of strings, we want to define *structure trees* for them.

This can be done with *user defined data types*.

Type definitions

General form:

$$\begin{array}{lll}\texttt{data}\ \text{type\_name (type\_parameters)} & = & \text{constructor}_1\ t_{11}\ldots t_{1i} \\ & \mid & \text{constructor}_2\ t_{21}\ldots t_{2j} \\ & \mid & \ldots \\ & \mid & \text{constructor}_n\ t_{n1}\ldots t_{nk}\end{array}$$

This can be used to create:

- enumeration types
- composite types
- recursive types
- parametric types

Example: Enumeration types

$$\begin{aligned} \text{data type\_name } (\text{type\_parameters}) = {} & \text{constructor}_1\ t_{11} \dots t_{1i} \\ \mid {} & \text{constructor}_2\ t_{21} \dots t_{2j} \\ \mid {} & \dots \\ \mid {} & \text{constructor}_n\ t_{n1} \dots t_{nk} \end{aligned}$$

Examples:

```
module Day2 where
--data Bool = True | False

data Season = Spring | Summer | Autumn | Winter

data Temperature = Hot | Cold | Moderate
```

Example: Enumeration types

Now, we can define a function using objects of type `Season` and `Temperature`.

```
weather :: Season -> Temperature
weather Summer = Hot
weather Winter = Cold
weather _      = Moderate
```

Example: Enumeration types

Now, we can define a function using objects of type `Season` and `Temperature`.

```
weather :: Season -> Temperature
weather Summer = Hot
weather Winter = Cold
weather _      = Moderate
```

But user-defined types do not automatically have operators for equality, ordering, show, etc.

```
> weather Spring
No instance for (Show Temperature)
arising from a use of ‘print’ at <interactive>:1:0-13
```

Instance declarations for `Show`

In order to display user-defined types, we can either define the function `show :: Typename -> String` explicitely . . .

```
instance Show Season where
  show Spring = "Spring"
  show Summer = "Summer"
  show Autumn = "Autumn"
  show Winter = "Winter"
```

Instance declarations for `Show`

In order to display user-defined types, we can either define the function `show :: Typename -> String` explicitely . . .

```
instance Show Season where
  show Spring = "Spring"
  show Summer = "Summer"
  show Autumn = "Autumn"
  show Winter = "Winter"
```

. . . or derive it.

```
data Season = Spring | Summer | Autumn | Winter
              deriving Show
```

Example: Composite types

$$\begin{array}{lll} \texttt{data}\ \text{type\_name}\ (\text{type\_parameters}) & = & \text{constructor}_1\ t_{11} \ldots t_{1i} \\ & | & \text{constructor}_2\ t_{21} \ldots t_{2j} \\ & | & \ \ldots \\ & | & \text{constructor}_n\ t_{n1} \ldots t_{nk} \end{array}$$

Examples:

```
data Book = Book Int String [String]

data Color = White | Black | RGB Int Int Int
```

Example: Recursive types

$$
\begin{array}{rl}
\texttt{data}\ \text{type\_name}\ (\text{type\_parameters}) = & \text{constructor}_1\ t_{11} \ldots t_{1i} \\
\mid & \text{constructor}_2\ t_{21} \ldots t_{2j} \\
\mid & \ldots \\
\mid & \text{constructor}_n\ t_{n1} \ldots t_{nk}
\end{array}
$$

Example:

```
data Tree = Leaf | Branch Tree Tree
```

Example: Polymorphic types

$$\begin{array}{lcl}\texttt{data}\ \text{type\_name}\ (\text{type\_parameters}) & = & \text{constructor}_1\ t_{11} \ldots t_{1i} \\ & \mid & \text{constructor}_2\ t_{21} \ldots t_{2j} \\ & \mid & \ldots \\ & \mid & \text{constructor}_n\ t_{n1} \ldots t_{nk}\end{array}$$

Examples:

```
data Maybe a = Nothing | Just a

data List a = Nil | Cons a (List a)

data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

From context-free grammars to datatypes

Now we can define types like the following:

```
data S = S NP VP
data N = Boy | Princess | Dwarf | Giant | Wizard
```

I.e. we treat categories (non-terminals) as types, and words (terminals) as data constructors. This gives us a very straightforward way to express a context-free grammar by means of datatypes.

A context-free grammar in Haskell

S ::= **NP** **VP**
NP ::= **NAME** | **DET** **N** | **DET** **RN**
ADJ ::= *happy* | *drunken* | *evil*
NAME ::= *Atreyu* | *Dorothy* | *Goldilocks* | *Snow White*
N ::= *boy* | *princess* | *dwarf* | *wizard* | **ADJ** **N**
RN ::= **N** **REL** **VP** | **N** **REL** **NP** **TV**
REL ::= *that*
DET ::= *some* | *every* | *no*
VP ::= **IV** | **TV** **NP** | **DV** **NP** **NP**
IV ::= *cheered* | *laughed* | *shuddered*
TV ::= *admired* | *helped* | *defeated* | *found*
DV ::= *gave*

A context-free grammar in Haskell

```
data S = S NP VP
```

NP ::= **NAME** | **DET** **N** | **DET** **RN**
ADJ ::= *happy* | *drunken* | *evil*
NAME ::= *Atreyu* | *Dorothy* | *Goldilocks* | *Snow White*
N ::= *boy* | *princess* | *dwarf* | *wizard* | **ADJ** **N**
RN ::= **N** **REL** **VP** | **N** **REL** **NP** **TV**
REL ::= *that*
DET ::= *some* | *every* | *no*
VP ::= **IV** | **TV** **NP** | **DV** **NP** **NP**
IV ::= *cheered* | *laughed* | *shuddered*
TV ::= *admired* | *helped* | *defeated* | *found*
DV ::= *gave*

A context-free grammar in Haskell

```
data S = S NP VP
data NP = NP1 NAME | NP2 DET N | NP3 DET RN
```

ADJ ::= *happy* | *drunken* | *evil*

NAME ::= *Atreyu* | *Dorothy* | *Goldilocks* | *Snow White*

N ::= *boy* | *princess* | *dwarf* | *wizard* | **ADJ N**

RN ::= **N REL VP** | **N REL NP TV**

REL ::= *that*

DET ::= *some* | *every* | *no*

VP ::= **IV** | **TV NP** | **DV NP NP**

IV ::= *cheered* | *laughed* | *shuddered*

TV ::= *admired* | *helped* | *defeated* | *found*

DV ::= *gave*

A context-free grammar in Haskell

```
  data S = S NP VP
 data NP = NP1 NAME | NP2 DET N | NP3 DET RN
data ADJ = Happy | Drunken | Evil
```

NAME ::= *Atreyu* | *Dorothy* | *Goldilocks* | *Snow White*
N ::= *boy* | *princess* | *dwarf* | *wizard* | **ADJ N**
RN ::= **N REL VP** | **N REL NP TV**
REL ::= *that*
DET ::= *some* | *every* | *no*
VP ::= **IV** | **TV NP** | **DV NP NP**
IV ::= *cheered* | *laughed* | *shuddered*
TV ::= *admired* | *helped* | *defeated* | *found*
DV ::= *gave*

A context-free grammar in Haskell

```
   data S = S NP VP
  data NP = NP1 NAME | NP2 DET N | NP3 DET RN
 data ADJ = Happy | Drunken | Evil
data NAME = Atreyu | Dorothy | Goldilocks | SnowWhite
```

N ::= *boy* | *princess* | *dwarf* | *wizard* | **ADJ N**

RN ::= **N REL VP** | **N REL NP TV**

REL ::= *that*

DET ::= *some* | *every* | *no*

VP ::= **IV** | **TV NP** | **DV NP NP**

IV ::= *cheered* | *laughed* | *shuddered*

TV ::= *admired* | *helped* | *defeated* | *found*

DV ::= *gave*

A context-free grammar in Haskell

```
   data S = S NP VP
  data NP = NP1 NAME | NP2 DET N | NP3 DET RN
 data ADJ = Happy | Drunken | Evil
data NAME = Atreyu | Dorothy | Goldilocks | SnowWhite
   data N = Boy | Princess | Dwarf | Giant | Wizard | N ADJ N
```

RN ::= **N REL VP** | **N REL NP TV**

REL ::= *that*

DET ::= *some* | *every* | *no*

VP ::= **IV** | **TV NP** | **DV NP NP**

IV ::= *cheered* | *laughed* | *shuddered*

TV ::= *admired* | *helped* | *defeated* | *found*

DV ::= *gave*

A context-free grammar in Haskell

```
   data S = S NP VP
  data NP = NP1 NAME | NP2 DET N | NP3 DET RN
 data ADJ = Happy | Drunken | Evil
data NAME = Atreyu | Dorothy | Goldilocks | SnowWhite
   data N = Boy | Princess | Dwarf | Giant | Wizard | N ADJ N
  data RN = RN1 N REL VP | RN2 N REL NP TV
```

REL ::= *that*

DET ::= *some* | *every* | *no*

VP ::= **IV** | **TV NP** | **DV NP NP**

IV ::= *cheered* | *laughed* | *shuddered*

TV ::= *admired* | *helped* | *defeated* | *found*

DV ::= *gave*

A context-free grammar in Haskell

```
   data S = S NP VP
  data NP = NP1 NAME | NP2 DET N | NP3 DET RN
 data ADJ = Happy | Drunken | Evil
data NAME = Atreyu | Dorothy | Goldilocks | SnowWhite
   data N = Boy | Princess | Dwarf | Giant | Wizard | N ADJ N
  data RN = RN1 N REL VP | RN2 N REL NP TV
 data REL = That
```

DET ::= *some* | *every* | *no*

VP ::= **IV** | **TV NP** | **DV NP NP**

IV ::= *cheered* | *laughed* | *shuddered*

TV ::= *admired* | *helped* | *defeated* | *found*

DV ::= *gave*

A context-free grammar in Haskell

```
   data S = S NP VP
  data NP = NP1 NAME | NP2 DET N | NP3 DET RN
 data ADJ = Happy | Drunken | Evil
data NAME = Atreyu | Dorothy | Goldilocks | SnowWhite
   data N = Boy | Princess | Dwarf | Giant | Wizard | N ADJ N
  data RN = RN1 N REL VP | RN2 N REL NP TV
 data REL = That
 data DET = Some | Every | No
```

VP ::= **IV** | **TV** **NP** | **DV** **NP** **NP**

IV ::= *cheered* | *laughed* | *shuddered*

TV ::= *admired* | *helped* | *defeated* | *found*

DV ::= *gave*

A context-free grammar in Haskell

```
   data S = S NP VP
  data NP = NP1 NAME | NP2 DET N | NP3 DET RN
 data ADJ = Happy | Drunken | Evil
data NAME = Atreyu | Dorothy | Goldilocks | SnowWhite
   data N = Boy | Princess | Dwarf | Giant | Wizard | N ADJ N
  data RN = RN1 N REL VP | RN2 N REL NP TV
 data REL = That
 data DET = Some | Every | No
  data VP = VP1 IV | VP2 TV NP | VP3 DV NP NP
```

IV ::= *cheered* | *laughed* | *shuddered*

TV ::= *admired* | *helped* | *defeated* | *found*

DV ::= *gave*

A context-free grammar in Haskell

```
   data S = S NP VP
  data NP = NP1 NAME | NP2 DET N | NP3 DET RN
 data ADJ = Happy | Drunken | Evil
data NAME = Atreyu | Dorothy | Goldilocks | SnowWhite
   data N = Boy | Princess | Dwarf | Giant | Wizard | N ADJ N
  data RN = RN1 N REL VP | RN2 N REL NP TV
 data REL = That
 data DET = Some | Every | No
  data VP = VP1 IV | VP2 TV NP | VP3 DV NP NP
  data IV = Cheered | Laughed | Shuddered
```

TV ::= *admired* | *helped* | *defeated* | *found*

DV ::= *gave*

A context-free grammar in Haskell

```
   data S = S NP VP
  data NP = NP1 NAME | NP2 DET N | NP3 DET RN
 data ADJ = Happy | Drunken | Evil
data NAME = Atreyu | Dorothy | Goldilocks | SnowWhite
   data N = Boy | Princess | Dwarf | Giant | Wizard | N ADJ N
  data RN = RN1 N REL VP | RN2 N REL NP TV
 data REL = That
 data DET = Some | Every | No
  data VP = VP1 IV | VP2 TV NP | VP3 DV NP NP
  data IV = Cheered | Laughed | Shuddered
  data TV = Admired | Helped | Defeated | Found
```

DV ::= *gave*

A context-free grammar in Haskell

```
   data S = S NP VP
  data NP = NP1 NAME | NP2 DET N | NP3 DET RN
 data ADJ = Happy | Drunken | Evil
data NAME = Atreyu | Dorothy | Goldilocks | SnowWhite
   data N = Boy | Princess | Dwarf | Giant | Wizard | N ADJ N
  data RN = RN1 N REL VP | RN2 N REL NP TV
 data REL = That
 data DET = Some | Every | No
  data VP = VP1 IV | VP2 TV NP | VP3 DV NP NP
  data IV = Cheered | Laughed | Shuddered
  data TV = Admired | Helped | Defeated | Found
  data DV = Gave
```

Haskell Version Again

```
data S = S NP VP deriving Show
data NP = NP1 NAME | NP2 DET N | NP3 DET RN
  deriving Show
data ADJ = Happy | Drunken | Evil
  deriving Show
data NAME = Atreyu | Dorothy | Goldilocks | SnowWhite
  deriving Show
data N = Boy | Princess | Dwarf | Giant | Wizard | N ADJ N
  deriving Show
data RN = RN1 N That VP | RN2 N That NP TV
  deriving Show
data That = That deriving Show
data DET = A_ | Some | Every | No | The
  deriving Show
data VP = VP1 IV | VP2 TV NP | VP3 DV NP NP deriving Show
data IV = Cheered | Laughed | Shuddered deriving Show
data TV = Admired | Helped | Defeated | Found deriving Show
data DV = Gave deriving Show
```

Examples

From well-formed expressions we can read off their structure.

Examples

From well-formed expressions we can read off their structure.

Well, not quite. Building structures from strings is called *parsing* . . .

Examples

From well-formed expressions we can read off their structure.

Well, not quite. Building structures from strings is called *parsing* . . .

- *every drunken wizard*

```
np :: NP
np = NP2 Every (N Drunken Wizard)
```

Examples

From well-formed expressions we can read off their structure.

Well, not quite. Building structures from strings is called *parsing* . . .

- *every drunken wizard*

```
np :: NP
np = NP2 Every (N Drunken Wizard)
```

- *No princess laughed.*

```
s1 :: S
s1 = S (NP2 No Princess) (VP1 Laughed)
```

Examples

From well-formed expressions we can read off their structure.

Well, not quite. Building structures from strings is called *parsing* . . .

- *every drunken wizard*

```
np :: NP
np = NP2 Every (N Drunken Wizard)
```

- *No princess laughed.*

```
s1 :: S
s1 = S (NP2 No Princess) (VP1 Laughed)
```

- *Atreyu found the princess.*

```
s2 :: S
s2 = S (NP1 Atreyu) (VP2 Found (NP2 The Princess))
```

Examples

Note that these examples are *typed structure trees*.

Examples

Note that these examples are *typed structure trees*.

Structure trees that do not ‘belong’ to the tree language are not well-typed.

Examples

Note that these examples are *typed structure trees*.

Structure trees that do not ‘belong’ to the tree language are not well-typed.

```
> S Princess Cheered

<interactive>:1:2:
Couldn’t match expected type ‘NP’ against inferred type ‘N’
In the first argument of ‘S’, namely ‘Princess’
In the expression:  S Princess Laughed
In the definition of ‘it’:  it = S Princess Laughed
```

Content

It was relatively easy to say what form is: Form is what can be captured by tree structures.

Content

It was relatively easy to say what form is: Form is what can be captured by tree structures.

It is harder to say what content is. But the relevant notion is sameness of content.

Content

It was relatively easy to say what form is: Form is what can be captured by tree structures.

It is harder to say what content is. But the relevant notion is sameness of content.

Replace the question ‘What is the meaning of a sentence?’ by the more precise question ‘When do two sentences express the same meaning’?

Content

It was relatively easy to say what form is: Form is what can be captured by tree structures.

It is harder to say what content is. But the relevant notion is sameness of content.

Replace the question ‘What is the meaning of a sentence?’ by the more precise question ‘When do two sentences express the same meaning’?

Restrict attention to declarative sentences. Declarative sentences are sentences that can be either true or false in a given context.

Content

It was relatively easy to say what form is: Form is what can be captured by tree structures.

It is harder to say what content is. But the relevant notion is sameness of content.

Replace the question 'What is the meaning of a sentence?' by the more precise question 'When do two sentences express the same meaning'?

Restrict attention to declarative sentences. Declarative sentences are sentences that can be either true or false in a given context.

It is raining today in Ljubljana and *I am Dutch* are declarative sentences. If they are uttered, the context of utterance fixes the meaning of *today* and *I*, and the uttered sentences are either true or false in that context.

Content

It was relatively easy to say what form is: Form is what can be captured by tree structures.

It is harder to say what content is. But the relevant notion is sameness of content.

Replace the question 'What is the meaning of a sentence?' by the more precise question 'When do two sentences express the same meaning'?

Restrict attention to declarative sentences. Declarative sentences are sentences that can be either true or false in a given context.

It is raining today in Ljubljana and *I am Dutch* are declarative sentences. If they are uttered, the context of utterance fixes the meaning of *today* and *I*, and the uttered sentences are either true or false in that context.

Let's try to be smarter next time is not a declarative sentence. *Is drinking coffee bad for you?* is not a declarative sentence either.

Sameness of Meaning

Jan van Eijck is Dutch and *I am Dutch* do not have the same meaning, for if one of the teachers of this course utter them they are both true, and if the other teacher of the course utters them one will be true and the other false.

Sameness of Meaning

Jan van Eijck is Dutch and *I am Dutch* do not have the same meaning, for if one of the teachers of this course utter them they are both true, and if the other teacher of the course utters them one will be true and the other false.

Jan van Eijck is Dutch and *Jan van Eijck is Nederlander* have the same meaning, as have *Cinderella est belle* and *Assepoester is mooi*.

Sameness of Meaning

Jan van Eijck is Dutch and *I am Dutch* do not have the same meaning, for if one of the teachers of this course utter them they are both true, and if the other teacher of the course utters them one will be true and the other false.

Jan van Eijck is Dutch and *Jan van Eijck is Nederlander* have the same meaning, as have *Cinderella est belle* and *Assepoester is mooi*.

To check for sameness of meaning one has to interpret sentences in many different situations, and check if the resulting truth values are always the same.

Sameness of Meaning

Jan van Eijck is Dutch and *I am Dutch* do not have the same meaning, for if one of the teachers of this course utter them they are both true, and if the other teacher of the course utters them one will be true and the other false.

Jan van Eijck is Dutch and *Jan van Eijck is Nederlander* have the same meaning, as have *Cinderella est belle* and *Assepoester is mooi*.

To check for sameness of meaning one has to interpret sentences in many different situations, and check if the resulting truth values are always the same.

But what does 'interpretation of a sentence in a situation' mean?

Sameness of Meaning

Jan van Eijck is Dutch and *I am Dutch* do not have the same meaning, for if one of the teachers of this course utter them they are both true, and if the other teacher of the course utters them one will be true and the other false.

Jan van Eijck is Dutch and *Jan van Eijck is Nederlander* have the same meaning, as have *Cinderella est belle* and *Assepoester is mooi*.

To check for sameness of meaning one has to interpret sentences in many different situations, and check if the resulting truth values are always the same.

But what does 'interpretation of a sentence in a situation' mean?

To replace the intuitive understanding by a precise understanding one can look at formal examples: the language of predicate logic and its semantics, or the Haskell language, and its interpretation.

The study of meaning

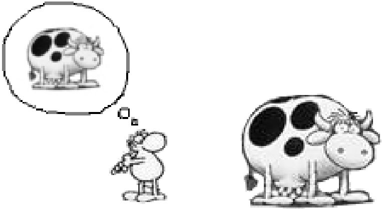
Lexical semantics:

- What are the meanings of words?

Compositional semantics:

- What are the meanings of phrases and sentences?
- And how are the meanings of phrases and sentences derived from the meanings of words?

What is the meaning of words?



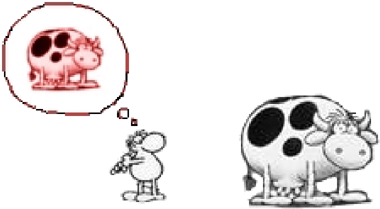
Meaning is...

What is the meaning of words?

Meaning is...

- about the world out there

What is the meaning of words?



Meaning is...

- about the world out there
- related to something in the mind (thoughts, ideas, concepts)

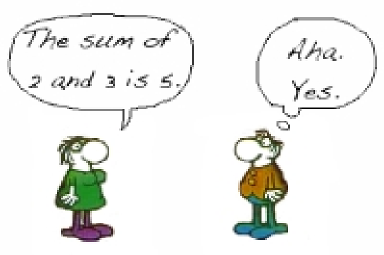
Formalizing word meanings

- **semantic feature sets**, e.g.
  $[\![\textit{bachelor}]\!] = [+\text{MALE}, +\text{ADULT}, -\text{MARRIED}]$
- **conceptual representations**, e.g. fuzzy concepts with a prototype centroid

We will stay agnostic to what the meanings of words are.

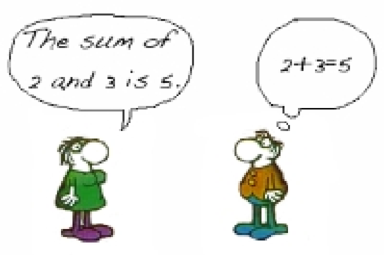
For us it will suffice to have a formal representation of them, that stands proxy for whatever we assume meanings to be (i.e. that are pointers to concepts or real world objects or something else).

Sentence meanings



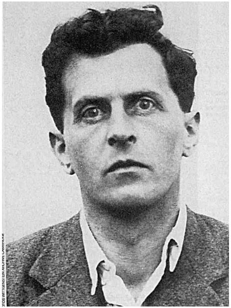
Denotational meaning (*knowing what*)
can be formalized as conditions for truth in situations.

Sentence meanings



Operational meaning (*knowing how*)
can be formalized as algorithms for performing an action.

Meanings as truth conditions



To know the meaning of a sentence is to know how the world would have to be for the sentence to be true.

(Ludwig Wittgenstein, 1889–1951)

The meaning of words and sentence parts is their contribution to the truth-conditions of the whole sentence.

Example

Intuitively, the sentence *It is raining in Amsterdam* is true if and only if it is raining in Amsterdam.

This sounds trivial, but it is not!

- How does the sentence get its truth conditions?
- What do the words contribute and how are these contributions combined?
- How does structure affect truth conditions?

Also, in order to specify a formal procedure for computing the truth conditions of a sentence, the metalanguage should be a formal language (and not English).

The principle of compositionality

The meaning of a complex expression is a function of the meanings of its parts and of the syntactic rules by which they are combined.

The principle of compositionality

> *The meaning of a complex expression is a function of the meanings of its parts and of the syntactic rules by which they are combined.*

This is a methodological issue:
The question is not whether natural languages satisfy the principle of compositionality, but rather whether we can and want to design meaning assembly in a way that this principle is respected.

Modeltheoretic semantics

A particular approach to truth-conditional semantics is modeltheoretic semantics. It represents the world as a mathematical structure — a model — and relates natural language expressions to this structure.

Modeltheoretic semantics

A model should comprise all parts of the world relevant for interpretation:

- **entities**
  (Atreyu, princesses, wizards, and other people)
- information about **which properties these entities satisfy**
  (being happy, laughing, etc.)
- information about **which relations hold between which entities**
  (admiring, defeating, etc.)
- maybe **contextual parameters** like time and place

Modeltheoretic semantics

We say that natural language expressions *denote* objects in the model.

Expression	Modeltheoretic object
sentence	truth value
proper name	entity
nouns	unary predicates (properties)
adjectives	unary predicates (properties)
intransitive verbs	unary predicates (properties)
transitive verbs	binary predicates (relations)

We will demonstrate the workings of a compositional modeltheoretic semantics using the example of **first-order predicate logic** (FOL), which we will need to know anyway as we are going to use it as formal metalanguage for meaning representations.

First-order predicate logic

Sentences denote propositions (linguistic entities that can be ascribed a truth value, i.e. something like a statement).

In order to be able to talk about the internal structure of propositions, first-order predicate logic provides us with names of objects, predicates for attributing properties to objects, and quantifiers for quantifying over objects.

Vocabulary of FOL

- variables $x, \ldots$
- individual constants $c, \ldots$
- predicate constants $R, \ldots$ of different arities
- logical constants:
    - unary connective $\neg$
    - binary connectives $\wedge, \vee, \rightarrow$
    - quantifiers $\forall, \exists$
- auxiliary symbols (brackets)

Syntax of FOL

- Variables and constants are terms.
- If $t_1, \ldots, t_n$ are terms and R is an n-place predicate constant, then $R(t_1, \ldots, t_n)$ is a formula.
- For x a variable and F a formula, $\forall x.F$ and $\exists x.F$ are formulas.
- If F_1 and F_2 are formulas, then so are $(F_1 \wedge F_2)$, $(F_1 \vee F_2)$, $\neg F_1$, and $(F_1 \rightarrow F_2)$.
- Nothing else is a formula.

Examples

- $R(x, y)$
- $\forall x.P(x)$
- $R(x, y, z) \wedge \exists y.(P(y) \rightarrow Q(y))$

Notational conventions:

- In $\exists x.F$ and $\forall x.F$, the dot (and thus the scope of the quantifier) extends as far to the right as possible.
- We use $x, y, z \ldots$ for variables, $P, Q, R, \ldots$ for relation symbols, and omit brackets when they are not necessary.

A grammar for FOL

$$
\begin{aligned}
\textbf{Const} &::= c \mid \textbf{Const}\,' \\
\textbf{Var} &::= x \mid \textbf{Var}\,' \\
\textbf{Rel} &::= R \mid \textbf{Rel}\,' \\
\textbf{Term} &::= \textbf{Const} \mid \textbf{Var} \\
\textbf{Formula} &::= \textbf{Rel}_n(\textbf{Term}_1, \dots, \textbf{Term}_n) \\
&\mid \forall\, \textbf{Var}.\textbf{Formula} \mid \exists\, \textbf{Var}.\textbf{Formula} \\
&\mid (\textbf{Formula} \wedge \textbf{Formula}) \mid (\textbf{Formula} \vee \textbf{Formula}) \\
&\mid \neg\, \textbf{Formula} \mid (\textbf{Formula} \rightarrow \textbf{Formula})
\end{aligned}
$$

Where $n \in \mathbb{N}$ encodes the arity of a relation symbol.

Implementation

```
data Term = Var Int
          | Const String
          deriving Eq

data Formula = Atom String [Term]
             | Neg Formula
             | Conj [Formula]
             | Disj [Formula]
             | Impl Formula Formula
             | Forall Int Formula
             | Exists Int Formula
             deriving Eq
```

Implementation

```
instance Show Term where
  show (Var n)   = show n
  show (Const s) = s

instance Show Formula where
  show (Atom s [])  = s
  show (Atom s ts)  = s ++"("++ showLst "," ts ++")"
  show (Neg  f)     = "~" ++ show f
  show (Conj fs)    = showLst " AND " fs
  show (Disj fs)    = showLst " OR " fs
  show (Impl f1 f2) = show f1 ++ " -> " ++ show f2
  show (Forall n f) = "FORALL " ++ show n ++ "." ++ show f
  show (Exists n f) = "EXISTS " ++ show n ++ "." ++ show f
```

Implementation

```
showLst :: Show a => String -> [a] -> String
showLst _ [] = []
showLst s (x:xs) | null xs   = show x
                 | otherwise = show x ++ s ++ showLst s xs
```

Bound and free variables

An instance of a variable v is **bound** if is in the scope of an instance of the quantifier $\forall v$ or $\exists v$ (i.e. occurs in a formula F in $\forall v.F$ or $\exists v.F$), otherwise it is **free**.

Feel free to try implementing corresponding functions `bound` and `free` in Haskell!

Semantics of FOL

Formulas are interpreted with respect to a **model** $M = \langle \mathcal{D}, \mathcal{I} \rangle$, where

- $\mathcal{D}$ is a **domain** of entities
- $\mathcal{I}$ is an **interpretation function** that specifies an appropriate semantic value for each constant of the language:
  - individual constants are interpreted as elements of $\mathcal{D}$ (i.e. objects in the domain)
  - *n*-place predicate constants R are interpreted as *n*-ary relations on $\mathcal{D}$ (i.e. relations over objects in the domain)

and an **assignment function** g that maps each variable to an element of the domain $\mathcal{D}$.

Truth of formulas

We write $[\![F]\!]^{M,g}$ for the interpretation of formula F relative to M, g.

$$
\begin{aligned}
[\![R^n(t_1, \dots, t_n)]\!]^{M,g} &= 1 \text{ iff } (\mathcal{I}(t_1), \dots, \mathcal{I}(t_n)) \in \mathcal{I}(R) \\
[\![F_1 \wedge F_2]\!]^{M,g} &= 1 \text{ iff } [\![F_1]\!]^{M,g} = 1 \text{ and } [\![F_2]\!]^{M,g} = 1 \\
[\![F_1 \vee F_2]\!]^{M,g} &= 1 \text{ iff } [\![F_1]\!]^{M,g} = 1 \text{ or } [\![F_2]\!]^{M,g} = 1 \\
[\![\neg F]\!]^{M,g} &= 1 \text{ iff } [\![F]\!]^{M,g} = 0 \\
[\![F_1 \rightarrow F_2]\!]^{M,g} &= 1 \text{ iff not both } [\![F_1]\!]^{M,g} = 1 \text{ and } [\![F_2]\!]^{M,g} = 0 \\
[\![\forall v.F]\!]^{M,g} &= 1 \text{ iff for all } d \in \mathcal{D} \text{ it holds that } [\![F]\!]^{M,g[v:=d]} = 1 \\
[\![\exists v.F]\!]^{M,g} &= 1 \text{ iff there is some } d \in \mathcal{D} \text{ such that } [\![F]\!]^{M,g[v:=d]} = 1
\end{aligned}
$$

Where $g[v := d]$ is the variable assigment which assigns d to the variable v, and is identical to g otherwise.

Example

$$[\![P(x) \rightarrow \exists y.Q(y) \wedge R(a,y)]\!]^{M,g} = 1$$

Example

$$[\![P(x) \rightarrow \exists y.Q(y) \land R(a,y)]\!]^{M,g} = 1$$

- iff not both $[\![P(x)]\!]^{M,g} = 1$ and $[\![\exists y.Q(y) \land R(a,y)]\!]^{M,g} = 0$

Example

$[\![P(x) \rightarrow \exists y.Q(y) \wedge R(a,y)]\!]^{M,g} = 1$

- iff not both $[\![P(x)]\!]^{M,g} = 1$ and $[\![\exists y.Q(y) \wedge R(a,y)]\!]^{M,g} = 0$
- iff not both $g(x) \in \mathcal{I}(P)$ and there is no $d \in \mathcal{D}$ such that $[\![Q(y) \wedge R(a,y)]\!]^{M,g[y:=d]} = 1$

Example

$[\![P(x) \rightarrow \exists y.Q(y) \wedge R(a,y)]\!]^{M,g} = 1$

- iff not both $[\![P(x)]\!]^{M,g} = 1$ and $[\![\exists y.Q(y) \wedge R(a,y)]\!]^{M,g} = 0$
- iff not both $g(x) \in \mathcal{I}(P)$ and there is no $d \in \mathcal{D}$ such that $[\![Q(y) \wedge R(a,y)]\!]^{M,g[y:=d]} = 1$
- iff not both $g(x) \in \mathcal{I}(P)$ and there is no $d \in \mathcal{D}$ such that $[\![Q(y)]\!]^{M,g[y:=d]} = 1$ and $[\![R(a,y)]\!]^{M,g[y:=d]} = 1$

Example

$[\![P(x) \rightarrow \exists y.Q(y) \wedge R(a,y)]\!]^{M,g} = 1$

- iff not both $[\![P(x)]\!]^{M,g} = 1$ and $[\![\exists y.Q(y) \wedge R(a,y)]\!]^{M,g} = 0$
- iff not both $g(x) \in \mathcal{I}(P)$ and there is no $d \in \mathcal{D}$ such that $[\![Q(y) \wedge R(a,y)]\!]^{M,g[y:=d]} = 1$
- iff not both $g(x) \in \mathcal{I}(P)$ and there is no $d \in \mathcal{D}$ such that $[\![Q(y)]\!]^{M,g[y:=d]} = 1$ and $[\![R(a,y)]\!]^{M,g[y:=d]} = 1$
- iff not both $g(x) \in \mathcal{I}(P)$ and there is no $d \in \mathcal{D}$ such that $g[y := d](y) \in \mathcal{I}(Q)$ and $(\mathcal{I}(a), g[y := d](y)) \in \mathcal{I}(R)$

Reformulation using Lists

$$
\begin{aligned}
[\![R^n[t_1,\ldots,t_n]]\!]^{M,g} &= 1 \text{ iff } [\mathcal{I}(t_1),\ldots,\mathcal{I}(t_n)] \in \mathcal{I}(R)\\
[\![\neg F]\!]^{M,g} &= 1 \text{ iff } [\![F]\!]^{M,g} = 0\\
[\![\wedge[F_1,\ldots,F_n]]\!]^{M,g} &= 1 \text{ iff } [\![F_1]\!]^{M,g} = 1 \text{ and } \ldots \text{and } [\![F_n]\!]^{M,g} = 1\\
[\![\vee[F_1,\ldots,F_n]]\!]^{M,g} &= 1 \text{ iff } [\![F_1]\!]^{M,g} = 1 \text{ or } \ldots \text{or } [\![F_n]\!]^{M,g} = 1\\
[\![\forall v.F]\!]^{M,g} &= 1 \text{ iff for all } d \in \mathcal{D} \text{ it holds that } [\![F]\!]^{M,g[v:=d]} = 1\\
[\![\exists v.F]\!]^{M,g} &= 1 \text{ iff there is some } d \in \mathcal{D} \text{ such that } [\![F]\!]^{M,g[v:=d]} = 1
\end{aligned}
$$

Implementation

```
data Model = Model { domain  :: Domain,
                     mapping :: ConstantMapping,
                     interpretation :: Interpretation }

type Domain = [Entity]

type ConstantMapping = String -> Entity
type Interpretation  = String -> [Entity] -> Bool
type Assignment = Int -> Entity
```

Implementation of $g[v := d]$

The implementation of model checking for predicate logic is straightforward, once we have captured the notion $g[v := d]$.

Implementation of $g[v := d]$

The implementation of model checking for predicate logic is straightforward, once we have captured the notion $g[v := d]$.

```
change :: (Int -> a) -> Int -> a -> Int -> a
change s x d = \ v -> if x == v then d else s v
```

Now `change g x d` is the implementation of $g[x := d]$.

Implementation of $[\![\phi]\!]^{M,g}$

```
eval :: Model -> Assignment -> Formula -> Bool
eval m g (Atom s ts)  = (interpretation m) s
                          (map (intTerm m g) ts)
eval m g (Neg f)      = not $ eval m g f
eval m g (Conj fs)    = and $ map (eval m g) fs
eval m g (Disj fs)    = or  $ map (eval m g) fs
eval m g (Impl f1 f2) = not ((eval m g f1)
                      && not (eval m g f2))
eval m g (Forall n f) =
  all (\d -> eval m (change g n d) f) (domain m)
eval m g (Exists n f) =
  any (\d -> eval m (change g n d) f) (domain m)
```

```
intTerm :: Model -> Assignment -> Term -> Entity
intTerm m g (Var n)   = g n
intTerm m g (Const s) = (mapping m) s
```

Satisfiability

The question whether a predicate logical formula is **satisfiable**, i.e. whether there is a model and an assignment that make it true, is **not decidable**, i.e. there is no method that tells us for an any formula in a finite number of steps, whether the answer is yes or no.

Satisfiability

The question whether a predicate logical formula is **satisfiable**, i.e. whether there is a model and an assignment that make it true, is **not decidable**, i.e. there is no method that tells us for an any formula in a finite number of steps, whether the answer is yes or no.

Predicate logic is expressive enough to formulate undecidable queries. It can specify the actions of Turing machines, and it can formalize statements about Turing machines that are undecidable, such as the **Halting Problem**.

Satisfiability

The question whether a predicate logical formula is **satisfiable**, i.e. whether there is a model and an assignment that make it true, is **not decidable**, i.e. there is no method that tells us for an any formula in a finite number of steps, whether the answer is yes or no.

Predicate logic is expressive enough to formulate undecidable queries. It can specify the actions of Turing machines, and it can formalize statements about Turing machines that are undecidable, such as the **Halting Problem**.

The **semantic tableaux method** is a systematic hunt for a conterexample to the validity of a formula.

Satisfiability

The question whether a predicate logical formula is **satisfiable**, i.e. whether there is a model and an assignment that make it true, is **not decidable**, i.e. there is no method that tells us for an any formula in a finite number of steps, whether the answer is yes or no.

Predicate logic is expressive enough to formulate undecidable queries. It can specify the actions of Turing machines, and it can formalize statements about Turing machines that are undecidable, such as the **Halting Problem**.

The **semantic tableaux method** is a systematic hunt for a conterexample to the validity of a formula. But this is **not a decision method**, for there are formulas for which the tableau construction process does not terminate.

Satisfiability

The question whether a predicate logical formula is **satisfiable**, i.e. whether there is a model and an assignment that make it true, is **not decidable**, i.e. there is no method that tells us for an any formula in a finite number of steps, whether the answer is yes or no.

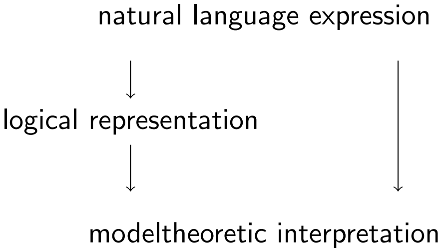
Predicate logic is expressive enough to formulate undecidable queries. It can specify the actions of Turing machines, and it can formalize statements about Turing machines that are undecidable, such as the **Halting Problem**.

The **semantic tableaux method** is a systematic hunt for a conterexample to the validity of a formula. But this is **not a decision method**, for there are formulas for which the tableau construction process does not terminate.

Note: However, there are fragments of first-order predicate logic that are decidable, e.g. the so-called $\exists^*\forall^*$*-prefix class* and *monadic predicate logic*.

Logical formulas as meaning representations

Direct vs indirect interpretation



natural language expression

logical representation

modeltheoretic interpretation

Indirect interpretation

Using FOL as language for meaning representations reduces the semantics of natural language to the semantics of FOL (which we know). Thus our task is to find a procedure to systematically map natural language expressions to expressions of FOL.

Digression on Direct Interpretation: Predicate Logic in Haskell

Digression on Direct Interpretation: Predicate Logic in Haskell

- The domain of discourse is some Haskell type. Let us say the type of Integers.

Digression on Direct Interpretation: Predicate Logic in Haskell

- The domain of discourse is some Haskell type. Let us say the type of Integers.
- Predicates are properties of integers, such as `odd`, `even`, `threefold`, `(>0)`, and relations such as `(>)`, `(<=)`.

Digression on Direct Interpretation: Predicate Logic in Haskell

- The domain of discourse is some Haskell type. Let us say the type of Integers.
- Predicates are properties of integers, such as `odd`, `even`, `threefold`, `(>0)`, and relations such as `(>)`, `(<=)`.
- Logical operations on predicates are negation, conjunction, disjunction.

Digression on Direct Interpretation: Predicate Logic in Haskell

- The domain of discourse is some Haskell type. Let us say the type of Integers.
- Predicates are properties of integers, such as `odd`, `even`, `threefold`, `(>0)`, and relations such as `(>)`, `(<=)`.
- Logical operations on predicates are negation, conjunction, disjunction.
- 'even or threefold' becomes `\ x -> even x || rem x 3 == 0`.

Digression on Direct Interpretation: Predicate Logic in Haskell

- The domain of discourse is some Haskell type. Let us say the type of Integers.
- Predicates are properties of integers, such as `odd`, `even`, `threefold`, `(>0)`, and relations such as `(>)`, `(<=)`.
- Logical operations on predicates are negation, conjunction, disjunction.
- ‘even or threefold’ becomes `\ x -> even x || rem x 3 == 0`.
- ‘not even’ becomes `not . even` or `\ x -> not (even x)`.

Digression on Direct Interpretation: Predicate Logic in Haskell

- The domain of discourse is some Haskell type. Let us say the type of Integers.
- Predicates are properties of integers, such as `odd`, `even`, `threefold`, `(>0)`, and relations such as `(>)`, `(<=)`.
- Logical operations on predicates are negation, conjunction, disjunction.
- 'even or threefold' becomes `\ x -> even x || rem x 3 == 0`.
- 'not even' becomes `not . even` or `\ x -> not (even x)`.
- Quantifications are 'some integers in [1..100] are even', or 'all integers in [1..] are positive'.

Examples of Quantifications in Haskell

Examples of Quantifications in Haskell

- ‘some integers in [1..100] are even’

Examples of Quantifications in Haskell

- ‘some integers in [1..100] are even’
-
  ```
  Day2> any even [1..100]
  True
  ```

Examples of Quantifications in Haskell

- ‘some integers in [1..100] are even’
- `Day2> any even [1..100]`
  `True`
- ‘all integers in [1..100] are positive’:

Examples of Quantifications in Haskell

- ‘some integers in [1..100] are even’

-
  ```
  Day2> any even [1..100]
  True
  ```
- ‘all integers in [1..100] are positive’:

-
  ```
  Day2> all (>0) [1..100]
  True
  Day2> all (>0) [1..]
  {Interrupted!}
  ```

Examples of Quantifications in Haskell

- 'some integers in [1..100] are even'

-
  ```
  Day2> any even [1..100]
  True
  ```
- 'all integers in [1..100] are positive':

-
  ```
  Day2> all (>0) [1..100]
  True
  Day2> all (>0) [1..]
  {Interrupted!}
  ```
- Question: does a quantification over an infinite list (like `[1..]`) always run forever?

Example vocabulary

- individual constant a, b, c
- predicate constants
  - for one-place predicates: *boy, princess, dwarf, giant, wizard, happy, evil, cheer, laugh*
  - for two-place predicates: *admire, defeat, find*
  - for three-place predicates: *give*

As well as variables and the logical constants $\forall, \exists, \wedge, \vee, \neg, \rightarrow$.

Example formulas

- $wizard(b)$
- $evil(x) \wedge admire(x, c)$
- $\forall x.happy(x)$
- $\exists y.\exists x.\neg find(x, y)$

Example translation

Lexical item	Logical constant	
Atreyu	*a*	(individual constant)
boy	*boy*	(one-place predicate)
princess	*princess*	(one-place predicate)
wizard	*wizard*	(one-place predicate)
cheered	*cheer*	(one-place predicate)
laughed	*laugh*	(one-place predicate)
happy	*happy*	(one-place predicate)
drunken	*drunken*	(one-place predicate)
admired	*admire*	(two-place predicate)
defeated	*defeat*	(two-place predicate)
found	*find*	(two-place predicate)
gave	*give*	(three-place predicate)

Example logical forms

- Atreyu laughed
- Everyone cheered
- Atreyu admired a princess
- Every dwarf defeated some giant

Example logical forms

- Atreyu laughed
  $laugh(a)$
- Everyone cheered
- Atreyu admired a princess
- Every dwarf defeated some giant

Example logical forms

- Atreyu laughed
  $laugh(a)$
- Everyone cheered
  $\forall x.cheer(x)$
- Atreyu admired a princess
- Every dwarf defeated some giant

Example logical forms

- Atreyu laughed
  $laugh(a)$
- Everyone cheered
  $\forall x.cheer(x)$
- Atreyu admired a princess
  $\exists x.princess(x) \wedge admire(a, x)$
- Every dwarf defeated some giant

Example logical forms

- Atreyu laughed
  $laugh(a)$
- Everyone cheered
  $\forall x. cheer(x)$
- Atreyu admired a princess
  $\exists x. princess(x) \wedge admire(a, x)$
- Every dwarf defeated some giant
  $\forall x. dwarf(x) \rightarrow \exists y. giant(y) \wedge defeat(x, y)$

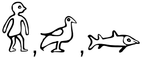
Example model

Assume a model $M = \langle \mathcal{D}, \mathcal{I} \rangle$, where

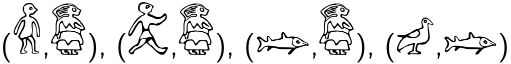
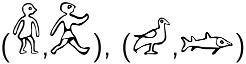
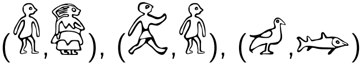
- $\mathcal{D} = \{$,,,,,,$\}$

- $\mathcal{I}(a) =$ 
- $\mathcal{I}(b) =$ 
- $\mathcal{I}(c) =$ 
- $\mathcal{I}(d) =$ 
- $\mathcal{I}(e) =$ 
- $\mathcal{I}(f) =$ 

Example model

- $\mathcal{I}(\mathit{boy}) = \{$  $\}$
- $\mathcal{I}(\mathit{princess}) = \{$  $\}$
- $\mathcal{I}(\mathit{wizard}) = \{$  $\}$
- $\mathcal{I}(\mathit{sword}) = \{$  $\}$
- $\mathcal{I}(\mathit{happy}) = \{$  $\}$
- $\mathcal{I}(\mathit{evil}) = \{$  $\}$
- $\mathcal{I}(\mathit{laugh}) = \{$  $\}$
- $\mathcal{I}(\mathit{cheer}) = \{$  $\}$

Example model

- $\mathcal{I}(admire) = \{$  $\}$
- $\mathcal{I}(defeat) = \{$  $\}$
- $\mathcal{I}(find) = \{$  $\}$
- $\mathcal{I}(give) = \{$  $\}$

Examples

- $\exists x.boy(x) \wedge admire(x,c)$ is true relative to M,g iff for some entity d it holds that both
  - $d \in \mathcal{I}(boy)$ and
  - $(d, \mathcal{I}(c)) \in \mathcal{I}(admire)$
- $\forall x.((wizard(x) \wedge \neg evil(x)) \rightarrow \exists y.admire(x,y))$ is true relative to M,g iff for some entity d it holds that
  - if $d \in \mathcal{I}(wizard)$ and $d \notin \mathcal{I}(evil)$, then
  - for some entity d' it holds that $(d,d') \in \mathcal{I}(admire)$

Implementation

```
data Entity = A | B | C | D | E | F deriving (Eq,Show)

model :: Model
model = Model dom intConst int

dom :: Domain
dom = [A,B,C,D,E,F]

intConst :: ConstantMapping
intConst "a" = A
intConst "b" = B
intConst "c" = C
intConst "d" = D
intConst "e" = E
intConst "f" = F
intConst  _  = error "unknown constant"
```

Implementation

```
int :: Interpretation
int "boy"      = \ [x] -> x ‘elem‘ [A]
int "princess" = \ [x] -> x ‘elem‘ [C]
int "wizard"   = \ [x] -> x ‘elem‘ [B]
int "sword"    = \ [x] -> x ‘elem‘ [D]
int "happy"    = \ [x] -> x ‘elem‘ [A,E,F]
int "evil"     = \ [x] -> x ‘elem‘ [B,D]
int "laugh"    = \ [x] -> x ‘elem‘ [B,E]
int "cheer"    = \ [x] -> x ‘elem‘ [A]
int "admire"   = \ [x,y] -> (x,y) ‘elem‘ [(A,C),(B,C),(E,C),
                                     (F,C)]
int "defeat"   = \ [x,y] -> (x,y) ‘elem‘ [(A,B),(E,F)]
int "find"     = \ [x,y] -> (x,y) ‘elem‘ [(A,C),(B,A),(E,F)]
int "give"     = \ [x,y,z] -> (x,y,z) ‘elem‘
                                        [(B,C,F),(C,A,D)]
int  _         = error "unknown constant"
```

Loose end

Loose end

- We do not yet know how to recursively construct FOL expressions corresponding to natural language expressions.

Loose end

- We do not yet know how to recursively construct FOL expressions corresponding to natural language expressions.
- Which kind of FOL expressions correspond to intermediate constituents like *admires every princess*?

Loose end

- We do not yet know how to recursively construct FOL expressions corresponding to natural language expressions.
- Which kind of FOL expressions correspond to intermediate constituents like *admires every princess*?

Difficulty: The structure of FOL expressions is quite different from the semantic structure of natural language expressions.

Loose end

- We do not yet know how to recursively construct FOL expressions corresponding to natural language expressions.
- Which kind of FOL expressions correspond to intermediate constituents like *admires every princess*?

Difficulty: The structure of FOL expressions is quite different from the semantic structure of natural language expressions.

Solution: In order to devise a syntax-directed translation of natural language to FOL, i.e. to compositionally build meaning representations in tandem with a syntactic analysis, we will interpret natural language expressions as expressions of a typed lambda calculus (subsuming FOL as fragment).

Course overview

Day 2:
Meaning representations and (predicate) logic

- **Day 3:**
  Lambda calculus and the composition of meanings
- **Day 4:**
  Extensionality and intensionality
- **Day 5:**
  From strings to truth conditions and beyond