Computational Semantics
Day 1: Getting Started with Haskell + Inference Engine for NL

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

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

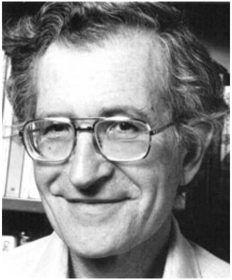
ESSLLI 2011, Ljubljana

The formal study of natural language

1916 Ferdinand de Saussure proposes that natural language may be analyzed as a formal system.

1957 Noam Chomsky proposes to define natural languages as sets of grammatical sentences, and to study their structure with formal means.

- The ability of language users to recognize members of this set is called competence.
- Goal: Build a model of our linguistic knowledge, abstracting from language performance (speech disabilities, memory limitations, errors, etc). Such a model is called grammar.



1970 Richard Montague proposes to extend the Chomskyan program to semantics and pragmatics.

The birth of formal semantics

There is in my opinion no important theoretical difference between natural languages and the artificial languages of logicians; indeed, I consider it possible to comprehend the syntax and semantics of both kinds of languages within a single natural and mathematically precise theory.

(Richard Montague, 1930–1971)

In fact, when we descibe grammars of fragments of natural languages in a formal way, we are doing the same as when describing formal languages. (And this allows for a relatively straightforward implementation.)

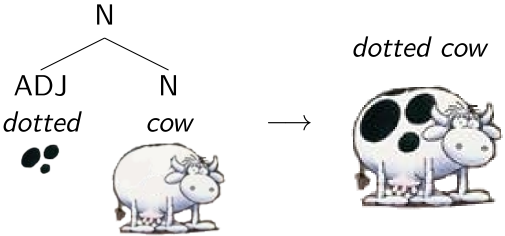
Organization of grammar

- **Phonology** investigates the smallest meaning-distinguishing units (speech sounds) and how they are combined into the smallest meaning-carrying units (morphemes).
- **Morphology** is concerned with how morphemes are combined into words.
- **Syntax** studies how words are combined into phrases and sentences.
- **Semantics** investigates the meanings of words, phrases and sentences, and how the meaning of a complex expression can be constructed from the meanings of its parts.

Our focus

We will concentrate on meaning and form at the level of phrases and sentences, i.e. start with words as basic building blocks.

Example:



A short history of Haskell

A short history of Haskell

In the 80s, efforts of researchers working on functional programming were scattered across many languages (Lisp, SASL, Miranda, ML,...).

In 1987 a dozen functional programmers decided to meet in order to reduce unnecessary diversity in functional programming languages by **designing a common language** that is

- based on ideas that enjoy a wide consensus
- suitable for further language research as well as applications, including building large systems
- freely available

A short history of Haskell

In 1990, they published the first **Haskell** specification, named after the logician and mathematician Haskell B. Curry (1900-1982).

Haskell is functional

A program consists entirely of functions.

- The main program itself is a function with the program's input as argument and the program's output as result.
- Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives.

Running a Haskell program consists in evaluating expressions (basically functions applied to arguments).

A shift in thinking

Imperative thinking:

- Variables are pointers to storage locations whose value can be updated all the time.
- You give a sequence of commands telling the computer what to do step by step.

Examples:

- initialize a variable `examplelist` of type integer list, then add 1, then add 2, then add 3
- in order to compute the factorial of n, initialize an integer variable `f` as 1, then for all `i` from 1 to n, set `f` to `f`$\times$`i`

A shift in thinking

Functional thinking:

- Variables are identifiers for an immutable, persistent value.
- You tell the computer what things are.

Examples:

- `examplelist` is a list of integers containing the elements 1, 2, and 3
- the factorial of n is the product of all integers from 1 to n

A shift in thinking

Functional thinking:

- Variables are identifiers for an immutable, persistent value.
- You tell the computer what things are.

Examples:

- `examplelist` is a list of integers containing the elements 1, 2, and 3
- the factorial of n is the product of all integers from 1 to n

```
factorial :: Int -> Int
factorial n = product [1..n]
```

A shift in thinking

Stop thinking in variable assignments, sequences and loops.

Start thinking in functions, immutable values and recursion.

Why use Haskell?

- Haskell allows for abstract, high order programming.
  (Ideally, more thinking and less writing and debugging.)

Why use Haskell?

- Haskell allows for abstract, high order programming.
  (Ideally, more thinking and less writing and debugging.)
- Haskell is based on the lambda calculus, therefore the step from formal semantics theory to implementation is very small.

Why use Haskell?

- Haskell allows for abstract, high order programming.
  (Ideally, more thinking and less writing and debugging.)
- Haskell is based on the lambda calculus, therefore the step from formal semantics theory to implementation is very small.
- Haskell offers you a new perspective on programming, it is powerful, and it is fun.

Why use Haskell?

- Haskell allows for abstract, high order programming.
  (Ideally, more thinking and less writing and debugging.)
- Haskell is based on the lambda calculus, therefore the step from formal semantics theory to implementation is very small.
- Haskell offers you a new perspective on programming, it is powerful, and it is fun.
- The type system behind Haskell is close related to the type system behind Montague grammar.

Why use Haskell?

- Haskell allows for abstract, high order programming.
  (Ideally, more thinking and less writing and debugging.)
- Haskell is based on the lambda calculus, therefore the step from formal semantics theory to implementation is very small.
- Haskell offers you a new perspective on programming, it is powerful, and it is fun.
- The type system behind Haskell is close related to the type system behind Montague grammar.
- Your Haskell understanding will influence the way you understand natural language semantics.

Why use Haskell?

- Haskell allows for abstract, high order programming. (Ideally, more thinking and less writing and debugging.)
- Haskell is based on the lambda calculus, therefore the step from formal semantics theory to implementation is very small.
- Haskell offers you a new perspective on programming, it is powerful, and it is fun.
- The type system behind Haskell is close related to the type system behind Montague grammar.
- Your Haskell understanding will influence the way you understand natural language semantics.

Haskell is rich enough to be useful. But above all, Haskell is a language in which people play. In the end, we want to infect your brain, not your hard drive.

(Simon Peyton-Jones)

Resources

- **For everything Haskell-related:** `haskell.org`.
- **Tutorials:**
  - Chapter 3 of our book
  - Real World Haskell
    `book.realworldhaskell.org/read/`
  - Learn you a Haskell for great good
    `learnyouahaskell.com`
  - A gentle introduction to Haskell
    `haskell.org/tutorial`

Getting started

Get the Haskell Platform:

- `http://hackage.haskell.org/platform/`

This includes the Glasgow Haskell Compiler (GHC) together with standard libraries and the interactive environment GHCi.

Haskell as a Calculator

Start the interpreter:

Haskell as a Calculator

Start the interpreter:

```
lucht:cmpsem jve$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

Haskell as a Calculator

Start the interpreter:

```
lucht:cmpsem jve$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

GHCi can be used to interactively evaluate expressions.

```
Prelude> 2 + 3
Prelude> 2 + 3 * 4
Prelude> 2^10
Prelude> (42 - 10) / 2
```

Your first Haskell program

1. Write the following code to a text file and save it as `first.hs`:

```
double :: Int -> Int
double n = 2 * n
```

Your first Haskell program

1. Write the following code to a text file and save it as `first.hs`:

```
double :: Int -> Int
double n = 2 * n
```

2. Inside GHCi, you can load the program with `:l first.hs` (or by running `ghci first.hs`).
   With `:r` you can reload it if you change something.

Your first Haskell program

1. Write the following code to a text file and save it as `first.hs`:

   ```
   double :: Int -> Int
   double n = 2 * n
   ```

2. Inside GHCi, you can load the program with `:l first.hs` (or by running `ghci first.hs`).
   With `:r` you can reload it if you change something.

3. Now you can evaluate expressions like `double 5`, `double (2+3)`, and `double (double 5)`.

Your first Haskell program

1. Write the following code to a text file and save it as `first.hs`:

   ```
   double :: Int -> Int
   double n = 2 * n
   ```

2. Inside GHCi, you can load the program with `:l first.hs` (or by running `ghci first.hs`).
   With `:r` you can reload it if you change something.
3. Now you can evaluate expressions like `double 5`, `double (2+3)`, and `double (double 5)`.
4. With `:t` you can ask GHCi about the type of an expression.

Your first Haskell program

1. Write the following code to a text file and save it as `first.hs`:

   ```
   double :: Int -> Int
   double n = 2 * n
   ```

2. Inside GHCi, you can load the program with `:l first.hs` (or by running `ghci first.hs`).
   With `:r` you can reload it if you change something.
3. Now you can evaluate expressions like `double 5`, `double (2+3)`, and `double (double 5)`.
4. With `:t` you can ask GHCi about the type of an expression.
5. Leave the interactive environment with `:q`.

Examples from Chapter 3 of the Book

```
module Day1

where

import Data.List
import Data.Char
```

Sentences can go on . . .

Sentences can go on

Sentences can go on . . .

Sentences can go on and on

Sentences can go on . . .

Sentences can go on and on and on

Sentences can go on . . .

Sentences can go on and on and on and on

Sentences can go on . . .

Sentences can go on and on and on and on and on

Sentences can go on . . .

Sentences can go on and on and on and on and on and on and on

Sentences can go on . . .

Sentences can go on and on and on and on and on and on and on

Sentences can go on . . .

Sentences can go on and on and on and on and on and on and on

```
gen :: Int -> String
gen 0 = "Sentences can go on"
gen n = gen (n-1) ++ " and on"

genS :: Int -> String
genS n = gen n ++ "."
```

A lazy list

```
sentences = "Sentences can go " ++ onAndOn

onAndOn = "on and " ++ onAndOn
```

Lambda Abstraction in Haskell

In Haskell, `\ x` expresses lambda abstraction over variable `x`.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

Lambda Abstraction in Haskell

In Haskell, \ x expresses lambda abstraction over variable x.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

- The intention is that variabele x stands proxy for a number of type Int.

Lambda Abstraction in Haskell

In Haskell, `\ x` expresses lambda abstraction over variable `x`.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

- The intention is that variabele `x` stands proxy for a number of type `Int`.
- The result, the squared number, also has type `Int`.

Lambda Abstraction in Haskell

In Haskell, `\ x` expresses lambda abstraction over variable `x`.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

- The intention is that variabele `x` stands proxy for a number of type `Int`.
- The result, the squared number, also has type `Int`.
- The function `sqr` is a function that, when combined with an argument of type `Int`, yields a value of type `Int`.

Lambda Abstraction in Haskell

In Haskell, `\ x` expresses lambda abstraction over variable `x`.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

- The intention is that variabele `x` stands proxy for a number of type `Int`.
- The result, the squared number, also has type `Int`.
- The function `sqr` is a function that, when combined with an argument of type `Int`, yields a value of type `Int`.
- This is precisely what the type-indication `Int -> Int` expresses.

String Functions in Haskell

```
Prelude> (\ x -> x ++ " emeritus") "professor"
"professor emeritus"
```

String Functions in Haskell

```
Prelude> (\ x -> x ++ " emeritus") "professor"
"professor emeritus"
```

This combines **lambda abstraction** and **concatenation**.

String Functions in Haskell

```
Prelude> (\ x -> x ++ " emeritus") "professor"
"professor emeritus"
```

This combines **lambda abstraction** and **concatenation**.

The types:

```
Prelude> :t (\ x -> x ++ " emeritus")
\x -> x ++ " emeritus" :: [Char] -> [Char]
Prelude> :t "professor"
"professor" :: String
Prelude> :t (\ x -> x ++ " emeritus") "professor"
(\x -> x ++ " emeritus") "professor" :: [Char]
```

Concatenation

The type of the concatenation function:

```
Prelude> :t (++)
(++) :: forall a. [a] -> [a] -> [a]
```

Concatenation

The type of the concatenation function:

```
Prelude> :t (++)
(++) :: forall a. [a] -> [a] -> [a]
```

The type indicates that (++) not only concatenates strings. It works for lists in general.

More String Functions in Haskell

```
Prelude> (\ x -> "nice " ++ x) "guy"
"nice guy"
Prelude> (\ f -> \ x -> "very " ++ (f x))
              (\ x -> "nice " ++ x) "guy"
"very nice guy"
```

More String Functions in Haskell

```
Prelude> (\ x -> "nice " ++ x) "guy"
"nice guy"
Prelude> (\ f -> \ x -> "very " ++ (f x))
               (\ x -> "nice " ++ x) "guy"
"very nice guy"
```

The types:

```
Prelude> :t "guy"
"guy" :: [Char]
Prelude> :t (\ x -> "nice " ++ x)
(\ x -> "nice " ++ x) :: [Char] -> [Char]
Prelude> :t (\ f -> \ x -> "very " ++ (f x))
(\ f -> \ x -> "very " ++ (f x))
  :: forall t. (t -> [Char]) -> t -> [Char]
```

Characters and Strings

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.
- Similarly, lists of integers have type `[Int]`.

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.
- Similarly, lists of integers have type `[Int]`.
- The empty string (or the empty list) is `[]`.

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.
- Similarly, lists of integers have type `[Int]`.
- The empty string (or the empty list) is `[]`.
- The type `[Char]` is abbreviated as `String`.

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.
- Similarly, lists of integers have type `[Int]`.
- The empty string (or the empty list) is `[]`.
- The type `[Char]` is abbreviated as `String`.
- Examples of characters are `'a'`, `'b'` (note the single quotes).

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.
- Similarly, lists of integers have type `[Int]`.
- The empty string (or the empty list) is `[]`.
- The type `[Char]` is abbreviated as `String`.
- Examples of characters are `'a'`, `'b'` (note the single quotes).
- Examples of strings are `"Montague"` and `"Chomsky"` (note the double quotes).

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.
- Similarly, lists of integers have type `[Int]`.
- The empty string (or the empty list) is `[]`.
- The type `[Char]` is abbreviated as `String`.
- Examples of characters are `'a'`, `'b'` (note the single quotes).
- Examples of strings are `"Montague"` and `"Chomsky"` (note the double quotes).
- In fact, `"Chomsky"` can be seen as an abbreviation of the following character list:

  `['C','h','o','m','s','k','y']`.

Properties of Strings

Properties of Strings

- If strings have type `[Char]` (or `String`), properties of strings have type `[Char] -> Bool`.

Properties of Strings

- If strings have type [Char] (or String), properties of strings have type [Char] -> Bool.
- Here is a simple property:

```
aword :: [Char] -> Bool
aword [] = False
aword (x:xs) = (x == 'a') || (aword xs)
```

Properties of Strings

- If strings have type `[Char]` (or `String`), properties of strings have type `[Char] -> Bool`.
- Here is a simple property:

```
aword :: [Char] -> Bool
aword [] = False
aword (x:xs) = (x == 'a') || (aword xs)
```

- This definition uses *pattern* matching: `(x:xs)` is the prototypical non-empty list.

Properties of Strings

- If strings have type `[Char]` (or `String`), properties of strings have type `[Char] -> Bool`.
- Here is a simple property:

```
aword :: [Char] -> Bool
aword [] = False
aword (x:xs) = (x == 'a') || (aword xs)
```

- This definition uses *pattern* matching: `(x:xs)` is the prototypical non-empty list.
- The `head` of `(x:xs)` is `x`, the `tail` is `xs`.

Properties of Strings

- If strings have type `[Char]` (or `String`), properties of strings have type `[Char] -> Bool`.
- Here is a simple property:

```
aword :: [Char] -> Bool
aword [] = False
aword (x:xs) = (x == 'a') || (aword xs)
```

- This definition uses *pattern* matching: `(x:xs)` is the prototypical non-empty list.
- The `head` of `(x:xs)` is `x`, the `tail` is `xs`.
- The head and tail are glued together by means of the operation `:`, of type `a -> [a] -> [a]`.

Properties of Strings

- If strings have type `[Char]` (or `String`), properties of strings have type `[Char] -> Bool`.
- Here is a simple property:

```
aword :: [Char] -> Bool
aword [] = False
aword (x:xs) = (x == 'a') || (aword xs)
```

- This definition uses *pattern* matching: `(x:xs)` is the prototypical non-empty list.
- The `head` of `(x:xs)` is `x`, the `tail` is `xs`.
- The head and tail are glued together by means of the operation `:`, of type `a -> [a] -> [a]`.
- The operation combines an object of type a with a list of objects of the same type to a new list of objects, again of the same type.

List Patterns

List Patterns

- It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.

List Patterns

- It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.
- Note that the function `aword` is called again from the body of its own definition. We will encounter such **recursive** function definitions again and again.

List Patterns

- It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.
- Note that the function `aword` is called again from the body of its own definition. We will encounter such **recursive** function definitions again and again.
- What the definition of `aword` says is that the empty string is not an `aword`, and a non-empty string is an `aword` if either the head of the string is the character `a`, or the tail of the sring is an `aword`.

List Patterns

- It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.
- Note that the function `aword` is called again from the body of its own definition. We will encounter such **recursive** function definitions again and again.
- What the definition of `aword` says is that the empty string is not an `aword`, and a non-empty string is an `aword` if either the head of the string is the character `a`, or the tail of the sring is an `aword`.
- The list pattern `[]` matches only the empty list,

List Patterns

- It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.
- Note that the function `aword` is called again from the body of its own definition. We will encounter such **recursive** function definitions again and again.
- What the definition of `aword` says is that the empty string is not an `aword`, and a non-empty string is an `aword` if either the head of the string is the character `a`, or the tail of the sring is an `aword`.
- The list pattern `[]` matches only the empty list,
- the list pattern `[x]` matches any singleton list,

List Patterns

- It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.
- Note that the function `aword` is called again from the body of its own definition. We will encounter such **recursive** function definitions again and again.
- What the definition of `aword` says is that the empty string is not an `aword`, and a non-empty string is an `aword` if either the head of the string is the character `a`, or the tail of the sring is an `aword`.
- The list pattern `[]` matches only the empty list,
- the list pattern `[x]` matches any singleton list,
- the list pattern `(x:xs)` matches any non-empty list.

List Reversal

CHOMSKY

EUGATNOM

List Reversal

CHOMSKY YKSMOHC

EUGATNOM

List Reversal

CHOMSKY YKSMOHC

EUGATNOM MONTAGUE

List Reversal

CHOMSKY YKSMOHC

EUGATNOM MONTAGUE

```
reversal :: [a] -> [a]
reversal []     = []
reversal (x:t) = reversal t ++ [x]
```

List Reversal

CHOMSKY YKSMOHC

EUGATNOM MONTAGUE

```
reversal :: [a] -> [a]
reversal []     = []
reversal (x:t) = reversal t ++ [x]
```

Reversal works for any list, not just for strings.

Haskell Basic Types

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.
- `Float` and `Double` represent floating point numbers. The elements of `Double` have higher precision.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.
- `Float` and `Double` represent floating point numbers. The elements of `Double` have higher precision.
- `Bool` is the type of Booleans.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.
- `Float` and `Double` represent floating point numbers. The elements of `Double` have higher precision.
- `Bool` is the type of Booleans.
- `Char` is the type of characters.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.
- `Float` and `Double` represent floating point numbers. The elements of `Double` have higher precision.
- `Bool` is the type of Booleans.
- `Char` is the type of characters.

Note that the name of a type always starts with a capital letter.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.
- `Float` and `Double` represent floating point numbers. The elements of `Double` have higher precision.
- `Bool` is the type of Booleans.
- `Char` is the type of characters.

Note that the name of a type always starts with a capital letter.

To denote arbitrary types, Haskell allows the use of *type variables*. For these, `a`, `b`, . . . , are used.

Haskell Derived Types

Haskell Derived Types

- By list-formation: if a is a type, `[a]` is the type of lists over a. Examples: `[Int]` is the type of lists of integers; `[Char]` is the type of lists of characters, or strings.

Haskell Derived Types

- By list-formation: if `a` is a type, `[a]` is the type of lists over `a`. Examples: `[Int]` is the type of lists of integers; `[Char]` is the type of lists of characters, or strings.
- By pair- or tuple-formation: if `a` and `b` are types, then `(a,b)` is the type of pairs with an object of type `a` as their first component, and an object of type `b` as their second component. If `a`, `b` and `c` are types, then `(a,b,c)` is the type of triples with an object of type `a` as their first component, an object of type `b` as their second component, and an object of type `c` as their third component . . .

Haskell Derived Types

- By list-formation: if `a` is a type, `[a]` is the type of lists over `a`. Examples: `[Int]` is the type of lists of integers; `[Char]` is the type of lists of characters, or strings.
- By pair- or tuple-formation: if `a` and `b` are types, then `(a,b)` is the type of pairs with an object of type `a` as their first component, and an object of type `b` as their second component. If `a`, `b` and `c` are types, then `(a,b,c)` is the type of triples with an object of type `a` as their first component, an object of type `b` as their second component, and an object of type `c` as their third component . . .
- By function definition: `a -> b` is the type of a function that takes arguments of type `a` and returns values of type `b`.

Haskell Derived Types

- By list-formation: if `a` is a type, `[a]` is the type of lists over `a`. Examples: `[Int]` is the type of lists of integers; `[Char]` is the type of lists of characters, or strings.
- By pair- or tuple-formation: if `a` and `b` are types, then `(a,b)` is the type of pairs with an object of type `a` as their first component, and an object of type `b` as their second component. If `a`, `b` and `c` are types, then `(a,b,c)` is the type of triples with an object of type `a` as their first component, an object of type `b` as their second component, and an object of type `c` as their third component . . .
- By function definition: `a -> b` is the type of a function that takes arguments of type `a` and returns values of type `b`.
- By defining your own datatype from scratch, with a `data` type declaration. More about this in due course.

Mapping

If you use the Hugs command `:t` to find the types of the function `map`, you get the following:

```
Prelude> :t map
map :: forall a b. (a -> b) -> [a] -> [b]
```

Mapping

If you use the Hugs command `:t` to find the types of the function `map`, you get the following:

```
Prelude> :t map
map :: forall a b. (a -> b) -> [a] -> [b]
```

The function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members.

Mapping

If you use the Hugs command `:t` to find the types of the function `map`, you get the following:

```
Prelude> :t map
map :: forall a b. (a -> b) -> [a] -> [b]
```

The function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members.

If `f` is a function of type `a -> b` and `xs` is a list of type `[a]`, then `map f xs` will return a list of type `[b]`. E.g., `map (^2) [1..9]` will produce the list of squares

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Sections

Sections

- In general, if `op` is an infix operator, `(op x)` is the operation resulting from applying `op` to its righthand side argument.

Sections

- In general, if op is an infix operator, `(op x)` is the operation resulting from applying op to its righthand side argument.
- `(x op)` is the operation resulting from applying op to its lefthand side argument.

Sections

- In general, if op is an infix operator, `(op x)` is the operation resulting from applying op to its righthand side argument.
- `(x op)` is the operation resulting from applying op to its lefthand side argument.
- `(op)` is the prefix version of the operator.

Sections

- In general, if op is an infix operator, `(op x)` is the operation resulting from applying op to its righthand side argument.
- `(x op)` is the operation resulting from applying op to its lefthand side argument.
- `(op)` is the prefix version of the operator.
- Thus `(2^)` is the operation that computes powers of 2, and `map (2^) [1..10]` will yield

  `[2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]`

Sections

- In general, if op is an infix operator, `(op x)` is the operation resulting from applying op to its righthand side argument.
- `(x op)` is the operation resulting from applying op to its lefthand side argument.
- `(op)` is the prefix version of the operator.
- Thus `(2^)` is the operation that computes powers of 2, and `map (2^) [1..10]` will yield

  `[2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]`
- Similarly, `(>3)` denotes the property of being greater than 3, and `(3>)` the property of being smaller than 3.

Map

If *p* is a property (an operation of type `a -> Bool`) and `l` is a list of type `[a]`, then `map p l` will produce a list of type `Bool` (a list of truth values), like this:

```
Prelude> map (>3) [1..6]
[False, False, False, True, True, True]
Prelude>
```

Map

If *p* is a property (an operation of type `a -> Bool`) and `l` is a list of type `[a]`, then `map p l` will produce a list of type `Bool` (a list of truth values), like this:

```
Prelude> map (>3) [1..6]
[False, False, False, True, True, True]
Prelude>
```

```
map :: (a -> b) -> [a] -> [b]
```

Map

If *p* is a property (an operation of type `a -> Bool`) and `l` is a list of type `[a]`, then `map p l` will produce a list of type `Bool` (a list of truth values), like this:

```
Prelude> map (>3) [1..6]
[False, False, False, True, True, True]
Prelude>
```

```
map :: (a -> b) -> [a] -> [b]
```

```
map f [] = []
map f (x:xs) = (f x) : map f xs
```

Filter

A function for filtering out the elements from a list that satisfy a given property.

Filter

A function for filtering out the elements from a list that satisfy a given property.

```
Prelude> filter (>3) [1..10]
[4,5,6,7,8,9,10]
```

Filter

A function for filtering out the elements from a list that satisfy a given property.

```
Prelude> filter (>3) [1..10]
[4,5,6,7,8,9,10]
```

```
filter :: (a -> Bool) -> [a] -> [a]
```

Filter

A function for filtering out the elements from a list that satisfy a given property.

```
Prelude> filter (>3) [1..10]
[4,5,6,7,8,9,10]
```

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
filter p [] = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise =     filter p xs
```

List comprehension

List comprehension is defining lists by the following method:

```
[ x | x <- xs, property x ]
```

This defines the sublist of `xs` of all items satisfying `property`. It is equivalent to:

```
filter property xs
```

Examples

```
someEvens     = [ x | x <- [1..1000], even x ]

evensUntil n = [ x | x <- [1..n], even x ]

allEvens     = [ x | x <- [1..], even x ]
```

Examples

```
someEvens     = [ x | x <- [1..1000], even x ]

evensUntil n = [ x | x <- [1..n], even x ]

allEvens      = [ x | x <- [1..], even x ]
```

Equivalently:

```
someEvens     = filter even [1..1000]

evensUntil n = filter even [1..n]

allEvens      = filter even [1..]
```

Nub

`nub` removes duplicates, as follows:

```
nub :: Eq a => [a] -> [a]
nub [] = []
nub (x:xs) = x : nub (filter (/= x) xs)
```

Function Composition

Function Composition

- The composition of two functions f and g, pronounced 'f after g' is the function that results from first applying g and next f.

Function Composition

- The composition of two functions f and g, pronounced 'f after g' is the function that results from first applying g and next f.
- Standard notation for this: $f \cdot g$.

Function Composition

- The composition of two functions f and g, pronounced 'f after g' is the function that results from first applying g and next f.
- Standard notation for this: $f \cdot g$.
- This is pronounced as "f after g".

Function Composition

- The composition of two functions f and g, pronounced 'f after g' is the function that results from first applying g and next f.
- Standard notation for this: $f \cdot g$.
- This is pronounced as "f after g".
- Haskell implementation:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
f . g = \ x -> f (g x)
```

Function Composition

- The composition of two functions f and g, pronounced 'f after g' is the function that results from first applying g and next f.
- Standard notation for this: $f \cdot g$.
- This is pronounced as "f after g".
- Haskell implementation:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
f . g = \ x -> f (g x)
```

- Note the types!

elem, all, and

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = x == y || elem x ys
```

elem, all, and

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = x == y || elem x ys

all :: Eq a => (a -> Bool) -> [a] -> Bool
all p = and . map p
```

elem, all, and

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = x == y || elem x ys
```

```
all :: Eq a => (a -> Bool) -> [a] -> Bool
all p = and . map p
```

Note the use of `.` for function composition.

elem, all, and

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = x == y || elem x ys
```

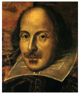
```
all :: Eq a => (a -> Bool) -> [a] -> Bool
all p = and . map p
```

Note the use of . for function composition.

```
and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && and xs
```

Sonnet 73

```
sonnet73 =
 "That time of year thou mayst in me behold\n"
 ++ "When yellow leaves, or none, or few, do hang\n"
 ++ "Upon those boughs which shake against the cold,\n"
 ++ "Bare ruin'd choirs, where late the sweet birds sang.\n"
 ++ "In me thou seest the twilight of such day\n"
 ++ "As after sunset fadeth in the west,\n"
 ++ "Which by and by black night doth take away,\n"
 ++ "Death's second self, that seals up all in rest.\n"
 ++ "In me thou see'st the glowing of such fire\n"
 ++ "That on the ashes of his youth doth lie,\n"
 ++ "As the death-bed whereon it must expire\n"
 ++ "Consumed with that which it was nourish'd by.\n"
 ++ "This thou perceivest, which makes thy love more strong,\n"
 ++ "To love that well which thou must leave ere long."
```



Counting

```
count :: Eq a => a -> [a] -> Int
count x []                 = 0
count x (y:ys) | x == y    = succ (count x ys)
               | otherwise = count x ys
```

Counting

```
count :: Eq a => a -> [a] -> Int
count x []                 = 0
count x (y:ys) | x == y    = succ (count x ys)
               | otherwise = count x ys
```

```
average :: [Int] -> Rational
average [] = error "empty list"
average xs = toRational (sum xs) / toRational (length xs)
```

Some Commands to Try Out

Some Commands to Try Out

- `putStrLn sonnet73`

Some Commands to Try Out

- `putStrLn sonnet73`
- `map toLower sonnet73`

Some Commands to Try Out

- `putStrLn sonnet73`
- `map toLower sonnet73`
- `map toUpper sonnet73`

Some Commands to Try Out

- `putStrLn sonnet73`
- `map toLower sonnet73`
- `map toUpper sonnet73`
- `filter (‘elem‘ "aeiou") sonnet73`

Some Commands to Try Out

- `putStrLn sonnet73`
- `map toLower sonnet73`
- `map toUpper sonnet73`
- `filter (‘elem‘ "aeiou") sonnet73`
- `count ’t’ sonnet73`

Some Commands to Try Out

- `putStrLn sonnet73`
- `map toLower sonnet73`
- `map toUpper sonnet73`
- `filter (‘elem‘ "aeiou") sonnet73`
- `count ’t’ sonnet73`
- `count ’t’ (map toLower sonnet73)`

Some Commands to Try Out

- `putStrLn sonnet73`
- `map toLower sonnet73`
- `map toUpper sonnet73`
- `filter (‘elem‘ "aeiou") sonnet73`
- `count ’t’ sonnet73`
- `count ’t’ (map toLower sonnet73)`
- `count "thou" (words sonnet73)`

Some Commands to Try Out

- `putStrLn sonnet73`
- `map toLower sonnet73`
- `map toUpper sonnet73`
- `` filter (`elem` "aeiou") sonnet73 ``
- `count 't' sonnet73`
- `count 't' (map toLower sonnet73)`
- `count "thou" (words sonnet73)`
- `count "thou" (words (map toLower sonnet73))`

Some Commands to Try Out

- `putStrLn sonnet73`
- `map toLower sonnet73`
- `map toUpper sonnet73`
- `filter (‘elem‘ "aeiou") sonnet73`
- `count ’t’ sonnet73`
- `count ’t’ (map toLower sonnet73)`
- `count "thou" (words sonnet73)`
- `count "thou" (words (map toLower sonnet73))`

Next, attempt exercise 3.16 on page 51 of the book.

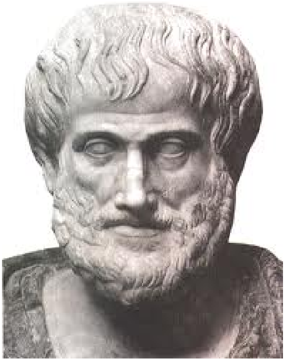
Example

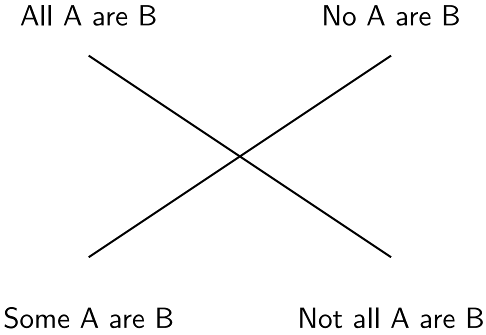
An inference engine with a natural language interface

Overview

- The Aristotelian quantifiers
- A natural language engine for talking about classes.
- Demo
- A tentative connection with cognitive realities.

The Aristotelian quantifiers





Aristotle interprets his quantifiers with existential import: *All A are B* and *No A are B* are taken to imply that there are *A*.

What can we ask or state with the Aristotelian quantifiers?

Questions and Statements (PN for plural nouns):

$$
\begin{array}{lll}
Q & ::= & \text{Are all PN PN?} \\
 & | & \text{Are no PN PN?} \\
 & | & \text{Are any PN PN?} \\
 & | & \text{Are any PN not PN?} \\
 & | & \text{What about PN?}
\end{array}
$$

$$
\begin{array}{lll}
S & ::= & \text{All PN are PN.} \\
 & | & \text{No PN are PN.} \\
 & | & \text{Some PN are PN.} \\
 & | & \text{Some PN are not PN.}
\end{array}
$$

Example interaction

```
user@home:~/courses/esslli2011$ ./Main
```

Example interaction

```
user@home:~/courses/esslli2011$ ./Main
Welcome to the Knowledge Base.
Update or query the KB:
```

Example interaction

```
user@home:~/courses/esslli2011$ ./Main
Welcome to the Knowledge Base.
Update or query the KB:
How about women?
```

Example interaction

```
user@home:~/courses/esslli2011$ ./Main
Welcome to the Knowledge Base.
Update or query the KB:
How about women?
All women are humans.
No women are men.

Update or query the KB:
```

Example interaction

```
user@home:~/courses/esslli2011$ ./Main
Welcome to the Knowledge Base.
Update or query the KB:
How about women?
All women are humans.
No women are men.

Update or query the KB:
All mammals are animals.
```

Example interaction

```
user@home:~/courses/esslli2011$ ./Main
Welcome to the Knowledge Base.
Update or query the KB:
How about women?
All women are humans.
No women are men.

Update or query the KB:
All mammals are animals.
I knew that already.

Update or query the KB:
```

Example interaction

```
user@home:~/courses/esslli2011$ ./Main
Welcome to the Knowledge Base.
Update or query the KB:
How about women?
All women are humans.
No women are men.

Update or query the KB:
All mammals are animals.
I knew that already.

Update or query the KB:
No mammals are birds.
```

Example interaction

```
user@home:~/courses/esslli2011$ ./Main
Welcome to the Knowledge Base.
Update or query the KB:
How about women?
All women are humans.
No women are men.

Update or query the KB:
All mammals are animals.
I knew that already.

Update or query the KB:
No mammals are birds.
OK.
```

Example interaction

```
Update or query the KB:
```

Example interaction

```
Update or query the KB:
How about women?
```

Example interaction

```
Update or query the KB:
How about women?
All women are humans.
No women are men.

Update or query the KB:
```

Example interaction

```
Update or query the KB:
How about women?
All women are humans.
No women are men.

Update or query the KB:
All humans are mammals.
```

Example interaction

```
Update or query the KB:
How about women?
All women are humans.
No women are men.

Update or query the KB:
All humans are mammals.
OK.

Update or query the KB:
```

Example interaction

```
Update or query the KB:
How about women?
All women are humans.
No women are men.

Update or query the KB:
All humans are mammals.
OK.

Update or query the KB:
How about women?
```

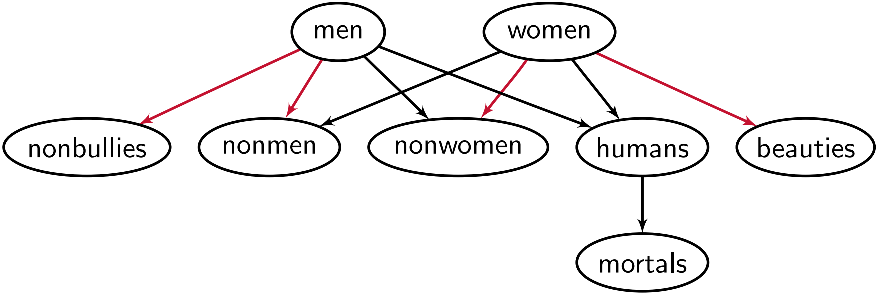
Example interaction

```
Update or query the KB:
How about women?
All women are humans.
No women are men.

Update or query the KB:
All humans are mammals.
OK.

Update or query the KB:
How about women?
All women are animals.
All women are humans.
All women are mammals.
No women are birds.
No women are men.
No women are owls.
```

Example knowledge base



The meanings of the Aristotelean quantifiers

- **ALL**: Set inclusion
- **SOME**: Non-empty set intersection
- **NOT ALL**: Non-inclusion
- **NO**: Empty intersection

Set inclusion:

- $A \subseteq B$ holds if and only if every element of A is an element of B.
- $A \not\subseteq B$ holds if and only if some element of A is not an element of B.

Complementation:

- Fix a universe U. $\overline{A} = U - A$ denotes the set of things in the universe that are not elements of A.

Implementation (in Haskell)

One possible implementation is given in Sections 4.3 and 5.7 of The Book.

Implementation (in Haskell)

One possible implementation is given in Sections 4.3 and 5.7 of The Book.

This code can be found at

`htpp://www.computational-semantics.eu/InfEng.hs`

Implementation (in Haskell)

One possible implementation is given in Sections 4.3 and 5.7 of The Book.

This code can be found at

`htpp://www.computational-semantics.eu/InfEng.hs`

Method: compute the relations $\subseteq$ and $\not\subseteq$ from the KB, using a fixpoint operation.

Implementation (in Haskell)

One possible implementation is given in Sections 4.3 and 5.7 of The Book.

This code can be found at

`htpp://www.computational-semantics.eu/InfEng.hs`

Method: compute the relations $\subseteq$ and $\not\subseteq$ from the KB, using a fixpoint operation.

Here we present a different method, based on **reduction to propositional logic**.

Implementation (in Haskell)

One possible implementation is given in Sections 4.3 and 5.7 of The Book.

This code can be found at

`htpp://www.computational-semantics.eu/InfEng.hs`

Method: compute the relations $\subseteq$ and $\not\subseteq$ from the KB, using a fixpoint operation.

Here we present a different method, based on **reduction to propositional logic**.

Homework for you: compare the performance of the two versions.

Syllogistics and Propositional Logic

Key fact: A finite set of syllogistic forms Σ is unsatisable if and only if there exists an existential form ψ such that ψ taken together with the universal forms from Σ is unsatiable.

Syllogistics and Propositional Logic

Key fact: A finite set of syllogistic forms Σ is unsatisable if and only if there exists an existential form ψ such that ψ taken together with the universal forms from Σ is unsatiable.

This restricted form of satisability can easily be tested with propositional logic.

Talking about the Properties of a Single Object

Talking about the Properties of a Single Object

- Suppose we talk about the properties of a single object x.

Talking about the Properties of a Single Object

- Suppose we talk about the properties of a single object x.
- Let proposition letter a express that object x has property A.

Talking about the Properties of a Single Object

- Suppose we talk about the properties of a single object x.
- Let proposition letter a express that object x has property A.
- Then a universal statement "All A are B" gets translated as $a \to b$.

Talking about the Properties of a Single Object

- Suppose we talk about the properties of a single object x.
- Let proposition letter a express that object x has property A.
- Then a universal statement "All A are B" gets translated as $a \rightarrow b$.
- An existential statement "Some A is B" gets translated as $a \wedge b$.

Talking about the Properties of a Single Object

- Suppose we talk about the properties of a single object x.
- Let proposition letter a express that object x has property A.
- Then a universal statement "All A are B" gets translated as $a \rightarrow b$.
- An existential statement "Some A is B" gets translated as $a \wedge b$.
- For each property A we use a single proposition letter a.

Talking about the Properties of a Single Object

- Suppose we talk about the properties of a single object x.
- Let proposition letter a express that object x has property A.
- Then a universal statement "All A are B" gets translated as $a \to b$.
- An existential statement "Some A is B" gets translated as $a \wedge b$.
- For each property A we use a single proposition letter a.
- We have to check for *each* existential statement whether it is satisfiable when taken together with all universal statements.

Talking about the Properties of a Single Object

- Suppose we talk about the properties of a single object x.
- Let proposition letter a express that object x has property A.
- Then a universal statement "All A are B" gets translated as $a \rightarrow b$.
- An existential statement "Some A is B" gets translated as $a \wedge b$.
- For each property A we use a single proposition letter a.
- We have to check for *each* existential statement whether it is satisfiable when taken together with all universal statements.
- To test the satisfiability of a set of syllogistic statements with n existential statements we need n checks.

Literals, Clauses, Clause Sets

Literals, Clauses, Clause Sets

literals a literal is a propositional letter or its negation.

Literals, Clauses, Clause Sets

literals a literal is a propositional letter or its negation.

clauses a clause is a set of literals

Literals, Clauses, Clause Sets

literals a literal is a propositional letter or its negation.

clauses a clause is a set of literals

clause sets a clause set is a set of clauses

Literals, Clauses, Clause Sets

literals a literal is a propositional letter or its negation.

clauses a clause is a set of literals

clause sets a clause set is a set of clauses

Read a clause as a *disjunction* of its literals, and a clause set as a *conjunction* of its clauses.

Literals, Clauses, Clause Sets

literals a literal is a propositional letter or its negation.

clauses a clause is a set of literals

clause sets a clause set is a set of clauses

Read a clause as a *disjunction* of its literals, and a clause set as a *conjunction* of its clauses.

$$(p \to q) \wedge (q \to r)$$

Literals, Clauses, Clause Sets

literals a literal is a propositional letter or its negation.

clauses a clause is a set of literals

clause sets a clause set is a set of clauses

Read a clause as a *disjunction* of its literals, and a clause set as a *conjunction* of its clauses.

$$(p \to q) \wedge (q \to r)$$

$$\{\{\neg p, q\}, \{\neg q, r\}\}.$$

Inference Rule: Unit Propagation

If one member of a clause set is a singleton $\{l\}$, then:

Inference Rule: Unit Propagation

If one member of a clause set is a singleton $\{l\}$, then:

- remove every other clause containing l from the clause set;

Inference Rule: Unit Propagation

If one member of a clause set is a singleton $\{l\}$, then:

- remove every other clause containing l from the clause set;
- remove $\bar{l}$ from every clause in which it occurs.

Inference Rule: Unit Propagation

If one member of a clause set is a singleton $\{l\}$, then:

- remove every other clause containing l from the clause set;
- remove $\bar{l}$ from every clause in which it occurs.

The result of applying this rule is a simplified equivalent clause set.

Inference Rule: Unit Propagation

If one member of a clause set is a singleton $\{l\}$, then:

- remove every other clause containing l from the clause set;
- remove $\bar{l}$ from every clause in which it occurs.

The result of applying this rule is a simplified equivalent clause set.

Unit propagation for $\{p\}$ to

$$\{\{p\},\{\neg p,q\},\{\neg q,r\},\{p,s\}\}$$

yields

Inference Rule: Unit Propagation

If one member of a clause set is a singleton $\{l\}$, then:

- remove every other clause containing l from the clause set;
- remove $\bar{l}$ from every clause in which it occurs.

The result of applying this rule is a simplified equivalent clause set.

Unit propagation for $\{p\}$ to

$$\{\{p\}, \{\neg p, q\}, \{\neg q, r\}, \{p, s\}\}$$

yields

$$\{\{p\}, \{q\}, \{\neg q, r\}\}.$$

Inference Rule: Unit Propagation

If one member of a clause set is a singleton $\{l\}$, then:

- remove every other clause containing l from the clause set;
- remove $\bar{l}$ from every clause in which it occurs.

The result of applying this rule is a simplified equivalent clause set.

Unit propagation for $\{p\}$ to

$$\{\{p\},\{\neg p, q\},\{\neg q, r\},\{p, s\}\}$$

yields

$$\{\{p\},\{q\},\{\neg q, r\}\}.$$

Unit propagation for $\{q\}$ to this yields:

Inference Rule: Unit Propagation

If one member of a clause set is a singleton $\{l\}$, then:

- remove every other clause containing l from the clause set;
- remove $\bar{l}$ from every clause in which it occurs.

The result of applying this rule is a simplified equivalent clause set.

Unit propagation for $\{p\}$ to

$$\{\{p\}, \{\neg p, q\}, \{\neg q, r\}, \{p, s\}\}$$

yields

$$\{\{p\}, \{q\}, \{\neg q, r\}\}.$$

Unit propagation for $\{q\}$ to this yields:

$$\{\{p\}, \{q\}, \{r\}\}.$$

HORNSAT

HORNSAT

- The *Horn fragment* of propositional logic consists of all clause sets where every clause has *at most one positive literal*.

HORNSAT

- The *Horn fragment* of propositional logic consists of all clause sets where every clause has *at most one positive literal*.
- Satisfiability for syllogistic forms containing exactly one existental statement translates to the Horn fragment of propositional logic.

HORNSAT

- The *Horn fragment* of propositional logic consists of all clause sets where every clause has *at most one positive literal*.
- Satisfiability for syllogistic forms containing exactly one existental statement translates to the Horn fragment of propositional logic.
- HORNSAT is the problem of testing Horn clause sets for satisfiability.

HORNSAT

- The *Horn fragment* of propositional logic consists of all clause sets where every clause has *at most one positive literal*.
- Satisfiability for syllogistic forms containing exactly one existental statement translates to the Horn fragment of propositional logic.
- HORNSAT is the problem of testing Horn clause sets for satisfiability.
- If unit propagation yields a clause set in which units $\{l\}, \{\bar{l}\}$ occur, the original clause set is unsatisfiable.

HORNSAT

- The *Horn fragment* of propositional logic consists of all clause sets where every clause has *at most one positive literal*.
- Satisfiability for syllogistic forms containing exactly one existental statement translates to the Horn fragment of propositional logic.
- HORNSAT is the problem of testing Horn clause sets for satisfiability.
- If unit propagation yields a clause set in which units $\{l\}, \{\bar{l}\}$ occur, the original clause set is unsatisfiable.
- Otherwise the units in the result determine a satisfying valuation.

HORNSAT

- The *Horn fragment* of propositional logic consists of all clause sets where every clause has *at most one positive literal*.
- Satisfiability for syllogistic forms containing exactly one existental statement translates to the Horn fragment of propositional logic.
- HORNSAT is the problem of testing Horn clause sets for satisfiability.
- If unit propagation yields a clause set in which units $\{l\}, \{\bar{l}\}$ occur, the original clause set is unsatisfiable.
- Otherwise the units in the result determine a satisfying valuation.
- Recipe: for all units $\{l\}$ occurring in the final clause set, map their proposition letter to the truth value that makes l true. Map all other proposition letters to false.

Module Declaration

```
module Syll where

import Data.List
import Data.Char
import System.IO
```

Literals, Clauses

```
data Lit = Pos String | Neg String deriving Eq

instance Show Lit where
  show (Pos x) = x
  show (Neg x) = '-': x

neg :: Lit -> Lit
neg (Pos x) = Neg x
neg (Neg x) = Pos x

type Clause = [Lit]

names :: [Clause] -> [String]
names = sort . nub . map nm . concat
  where nm (Pos x) = x
        nm (Neg x) = x
```

Unit Propagation (1)

```
unitProp :: Lit -> [Clause] -> [Clause]
unitProp x cs = concat (map (unitP x) cs)

unitP :: Lit -> Clause -> [Clause]
unitP x ys = if elem x ys then []
              else
               if elem (neg x) ys
                then [delete (neg x) ys]
                else [ys]

unit :: Clause -> Bool
unit [x] = True
unit  _  = False
```

Unit Propagation (2)

```
propagate :: [Clause] -> Maybe ([Lit],[Clause])
```

Unit Propagation (2)

```
propagate :: [Clause] -> Maybe ([Lit],[Clause])
```

```
propagate cls =
  prop [] (concat (filter unit cls)) (filter (not.unit) cls)
  where
    prop :: [Lit] -> [Lit] -> [Clause]
            -> Maybe ([Lit],[Clause])
    prop xs [] clauses = Just (xs,clauses)
    prop xs (y:ys) clauses =
      if elem (neg y) xs
       then Nothing
       else prop (y:xs)(ys++newlits) clauses' where
        newclauses = unitProp y clauses
        zs         = filter unit newclauses
        clauses'   = newclauses \\ zs
        newlits    = concat zs
```

KBs, Statements

```
type KB = ([Clause],[[Clause]])
-- first element: universal statements
-- second element: one clause list per existential statement

domain :: KB -> [Lit]
domain (xs,yss) =
   map (\ x -> Pos x) zs ++ map (\ x -> Neg x) zs
  where zs = names (xs ++ concat yss)

type Class = Lit

data Statement =
     All    Class    Class | No      Class  Class
   | Some   Class    Class | SomeNot Class  Class
   | AreAll Class Class    | AreNo   Class  Class
   | AreAny Class Class    | AnyNot  Class  Class
   | What   Class
  deriving Eq
```

Statement Display

```
instance Show Statement where
  show (All as bs)      =
    "All "  ++ show as ++ " are " ++ show bs ++ "."
  show (No as bs)       =
    "No "   ++ show as ++ " are " ++ show bs ++ "."
  show (Some as bs)     =
    "Some " ++ show as ++ " are " ++ show bs ++ "."
  show (SomeNot as bs) =
    "Some " ++ show as ++ " are not " ++ show bs ++ "."
  show (AreAll as bs)   =
    "Are all " ++ show as ++ show bs ++ "?"
  show (AreNo as bs)    =
    "Are no "  ++ show as ++ show bs ++ "?"
  show (AreAny as bs)   =
    "Are any " ++ show as ++ show bs ++ "?"
  show (AnyNot as bs)   =
    "Are any " ++ show as ++ " not " ++ show bs ++ "?"
  show (What as)        = "What about " ++ show as ++ "?"
```

Statement Classification, Query Negation

```
isQuery :: Statement -> Bool
isQuery (AreAll _ _)  = True
isQuery (AreNo _ _)   = True
isQuery (AreAny _ _)  = True
isQuery (AnyNot _ _)  = True
isQuery (What _)      = True
isQuery  _            = False

negat :: Statement -> Statement
negat (AreAll as bs) = AnyNot as bs
negat (AreNo  as bs) = AreAny as bs
negat (AreAny as bs) = AreNo  as bs
negat (AnyNot as bs) = AreAll as bs
```

The ⊂ Relation

```
subsetRel :: KB -> [(Class,Class)]
subsetRel kb =
  [(x,y) | x <- classes, y <- classes,
     propagate ([x]:[neg y]: fst kb) == Nothing ]
    where classes = domain kb
```

The ⊂ Relation

```
subsetRel :: KB -> [(Class,Class)]
subsetRel kb =
  [(x,y) | x <- classes, y <- classes,
     propagate ([x]:[neg y]: fst kb) == Nothing ]
    where classes = domain kb
```

If $R \subseteq A^2$ and $x \in A$, then $xR := \{y \mid (x, y) \in R\}$. This is called a *right section of a relation*.

```
rSection :: Eq a => a -> [(a,a)] -> [a]
rSection x r = [ y | (z,y) <- r, x == z ]
```

The $\subset$ Relation

```
subsetRel :: KB -> [(Class,Class)]
subsetRel kb =
  [(x,y) | x <- classes, y <- classes,
     propagate ([x]:[neg y]: fst kb) == Nothing ]
    where classes = domain kb
```

If $R \subseteq A^2$ and $x \in A$, then $xR := \{y \mid (x, y) \in R\}$. This is called a *right section of a relation*.

```
rSection :: Eq a => a -> [(a,a)] -> [a]
rSection x r = [ y | (z,y) <- r, x == z ]
```

The supersets of a class are given by a right section of the subset relation. I.e. the supersets of a class are all classes of which it is a subset.

```
supersets :: Class -> KB -> [Class]
supersets cl kb = rSection cl (subsetRel kb)
```

The Relation of Having an Non-empty Intersection

```
intersectRel :: KB -> [(Class,Class)]
intersectRel kb@(xs,yys) =
 nub [(x,y) | x <- classes, y <- classes, lits <- litsList,
    elem x lits && elem y lits    ]
     where
       classes = domain kb
       litsList =
         [ maybe [] fst (propagate (ys++xs)) | ys <- yys ]
```

The Relation of Having an Non-empty Intersection

```
intersectRel :: KB -> [(Class,Class)]
intersectRel kb@(xs,yys) =
 nub [(x,y) | x <- classes, y <- classes, lits <- litsList,
    elem x lits && elem y lits   ]
     where
       classes = domain kb
       litsList =
         [ maybe [] fst (propagate (ys++xs)) | ys <- yys ]
```

```
intersectionsets :: Class -> KB -> [Class]
intersectionsets cl kb = rSection cl (intersectRel kb)
```

Caution about KB Query

There are three possibilities:

Caution about KB Query

There are three possibilities:

- `derive kb stmt` is true. This means that the statement is derivable, hence true.

Caution about KB Query

There are three possibilities:

- `derive kb stmt` is true. This means that the statement is derivable, hence true.
- `derive kb (neg stmt)` is true. This means that the negation of `stmt` is derivable, hence true. So `stmt` is false.

Caution about KB Query

There are three possibilities:

- `derive kb stmt` is true. This means that the statement is derivable, hence true.
- `derive kb (neg stmt)` is true. This means that the negation of `stmt` is derivable, hence true. So `stmt` is false.
- neither `derive kb stmt` nor `derive kb (neg stmt)` is true. This means that the knowledge base has no information about `stmt`.

Derivability

```
derive :: KB -> Statement -> Bool
derive kb (AreAll as bs) = bs `elem` (supersets as kb)
derive kb (AreNo as bs)  = (neg bs) `elem` (supersets as kb)
derive kb (AreAny as bs) = bs `elem` (intersectionsets as kb)
derive kb (AnyNot as bs) = (neg bs) `elem`
                                (intersectionsets as kb)
```

Building a KB

Building a KB

- To *build* a knowledge base we need a function for updating an existing knowledge base with a statement.

Building a KB

- To *build* a knowledge base we need a function for updating an existing knowledge base with a statement.
- If the update is successful, we want an updated knowledge base.

Building a KB

- To *build* a knowledge base we need a function for updating an existing knowledge base with a statement.
- If the update is successful, we want an updated knowledge base.
- If the update is not successful, we want to get an indication of failure.

Example: Update with an 'All' statement

The update function checks for possible inconsistencies. E.g., a request to add an $A \subseteq B$ fact to the knowledge base leads to an inconsistency if $A \not\subseteq B$ is already derivable.

```
update  :: Statement -> KB -> Maybe (KB,Bool)

update (All as bs) kb@(xs,yss)
  | bs' `elem` (intersectionsets as kb) = Nothing
  | bs `elem` (supersets  as kb) = Just (kb,False)
  | otherwise = Just (([as',bs]:xs,yss),True)
 where
   as' = neg as
   bs' = neg bs
```

Demo

...

Conclusions

Conclusions

- Mini-case of computational semantics. What is the use of this?

Conclusions

- Mini-case of computational semantics. What is the use of this?
- Cognitive research focusses on this kind of quantifier reasoning ...

Conclusions

- Mini-case of computational semantics. What is the use of this?
- Cognitive research focusses on this kind of quantifier reasoning ...
- Can this be used to meet cognitive realities? Links with cognition by refinement of this calculus ...

Conclusions

- Mini-case of computational semantics. What is the use of this?
- Cognitive research focusses on this kind of quantifier reasoning ...
- Can this be used to meet cognitive realities? Links with cognition by refinement of this calculus ...
- The "natural logic for natural language" enterprise ...

Conclusions

- Mini-case of computational semantics. What is the use of this?
- Cognitive research focusses on this kind of quantifier reasoning ...
- Can this be used to meet cognitive realities? Links with cognition by refinement of this calculus ...
- The "natural logic for natural language" enterprise ...
- Towards Rational Reconstruction of Cognitive Processing