An Introduction to Formal Computational Semantics

CS224N/Ling 280

Christopher Manning

May 23, 2000; revised 2008
A first example

Lexicon

*Kathy*, NP : *kathy*

*Fong*, NP : *fong*

*respects*, V : \( \lambda y.\lambda x.\text{respect}(x, y) \)

*runs*, V : \( \lambda x.\text{run}(x) \)

Grammar

\( S : \beta(\alpha) \rightarrow \text{NP} : \alpha \quad \text{VP} : \beta \)

\( \text{VP} : \beta(\alpha) \rightarrow \text{V} : \beta \quad \text{NP} : \alpha \)

\( \text{VP} : \beta \rightarrow \text{V} : \beta \)
A first example

\[
S : \text{respect}(\text{kathy}, \text{fong})
\]

\[
\begin{align*}
\text{NP} &: \text{kathy} \\
\text{VP} &: \lambda x. \text{respect}(x, \text{fong}) \\
\text{V} &: \lambda y. \lambda x. \text{respect}(x, y) \\
\text{NP} &: \text{fong}
\end{align*}
\]

\[
[\text{VP respects Fong}] : [\lambda y. \lambda x. \text{respect}(x, y)](\text{fong})
\]

\[
= \lambda x. \text{respect}(x, \text{fong}) \quad [\beta \text{ red.}]
\]

\[
[S \text{ Kathy respects Fong}] : [\lambda x. \text{respect}(x, \text{fong})](\text{kathy})
\]

\[
= \text{respect}(\text{kathy}, \text{fong})
\]
Database/knowledgebase interfaces

- Assume that `respect` is a table `Respect` with two fields `respecter` and `respected`
- Assume that `kathy` and `fong` are IDs in the database: `k` and `f`
- If we assert *Kathy respects Fong* we might evaluate the form `respect(fong)(kathy)` by doing an insert operation:

  `insert into Respects(respecter, respected) values (k, f)`
Database/knowledgebase interfaces

• Below we focus on questions like *Does Kathy respect Fong* for which we will use the relation to ask:
  
  select ‘yes’ from Respects where Respects.respecter = k and Respects.respected = f

• We interpret “no rows returned” as ‘no’ = 0.
Typed $\lambda$ calculus (Church 1940)

- Everything has a type (like Java!)
- \textbf{Bool} truth values (0 and 1)
  \textbf{Ind} individuals
  \textbf{Ind $\rightarrow$ Bool} properties
  \textbf{Ind $\rightarrow$ Ind $\rightarrow$ Bool} binary relations
- \textbf{kathy} and \textbf{fong} are \textbf{Ind}
  \textbf{run} is \textbf{Ind $\rightarrow$ Bool}
  \textbf{respect} is \textbf{Ind $\rightarrow$ Ind $\rightarrow$ Bool}
- Types are interpreted right associatively.
  \textbf{respect} is \textbf{Ind $\rightarrow$ (Ind $\rightarrow$ Bool)}
- We convert a several argument function into embedded unary functions. Referred to as \textit{currying}.
Typed $\lambda$ calculus (Church 1940)

- Once we have types, we don’t need $\lambda$ variables just to show what arguments something takes, and so we can introduce another operation of the $\lambda$ calculus:
  
  $\eta$ reduction [abstractions can be contracted]

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

- This means that instead of writing:

  $\lambda y. \lambda x. \text{respect}(x, y)$

  we can just write:

  \text{respect}
Typed $\lambda$ calculus (Church 1940)

- $\lambda$ extraction allowed over any type (not just first-order)
- $\beta$ reduction [application]
  $(\lambda x. P(\cdots, x, \cdots))(Z) \Rightarrow P(\cdots, Z, \cdots)$
- $\eta$ reduction [abstractions can be contracted]
  $\lambda x. (P(x)) \Rightarrow P$
- $\alpha$ reduction [renaming of variables]
Typed λ calculus (Church 1940)

• The first form we introduced is called the β, η long form, and the second more compact representation (which we use quite a bit below) is called the β, η normal form. Here are some examples:

<table>
<thead>
<tr>
<th>β, η normal form</th>
<th>β, η long form</th>
</tr>
</thead>
<tbody>
<tr>
<td>run</td>
<td>λx.run(x)</td>
</tr>
<tr>
<td>every²(kid, run)</td>
<td>every²((λx.kid(x)), (λx.run(x)))</td>
</tr>
<tr>
<td>yesterday(run)</td>
<td>λy.yesterday(λx.run(x))(y)</td>
</tr>
</tbody>
</table>
Types of major syntactic categories

- nouns and verb phrases will be properties \((\text{Ind} \rightarrow \text{Bool})\)
- noun phrases are \text{Ind} – though they are commonly type-raised to \((\text{Ind} \rightarrow \text{Bool}) \rightarrow \text{Bool}\)
- adjectives are \((\text{Ind} \rightarrow \text{Bool}) \rightarrow (\text{Ind} \rightarrow \text{Bool})\)
  
This is because adjectives modify noun meanings, that is properties.

- Intensifiers modify adjectives: e.g, *very* in a *very happy camper*, so they’re \(((\text{Ind} \rightarrow \text{Bool}) \rightarrow (\text{Ind} \rightarrow \text{Bool})) \rightarrow ((\text{Ind} \rightarrow \text{Bool}) \rightarrow (\text{Ind} \rightarrow \text{Bool}))\) [honest!].
A grammar fragment

- $S : \beta(\alpha) \rightarrow NP : \alpha$  
  $VP : \beta$
- $NP : \beta(\alpha) \rightarrow Det : \beta$  
  $N' : \alpha$
- $N' : \beta(\alpha) \rightarrow Adj : \beta$  
  $N' : \alpha$
- $N' : \beta(\alpha) \rightarrow N' : \alpha$  
  $PP : \beta$
- $N' : \beta \rightarrow N : \beta$
- $VP : \beta(\alpha) \rightarrow V : \beta$  
  $NP : \alpha$
- $VP : \beta(\gamma)(\alpha) \rightarrow V : \beta$  
  $NP : \alpha$  
  $NP : \gamma$
- $VP : \beta(\alpha) \rightarrow VP : \alpha$  
  $PP : \beta$
- $VP : \beta \rightarrow V : \beta$
- $PP : \beta(\alpha) \rightarrow P : \beta$  
  $NP : \alpha$
A grammar fragment

- Kathy, NP: kathy$_{\text{Ind}}$
  
  Fong, NP: fong$_{\text{Ind}}$
  
  Palo Alto, NP: paloalto$_{\text{Ind}}$
  
  car, N: car$_{\text{Ind}}$ → Bool
  
  overpriced, Adj: overpriced$_{\text{Ind}}$ → (Ind → Bool) → (Ind → Bool)
  
  outside, PP: outside$_{\text{Ind}}$ → (Ind → Bool) → (Ind → Bool)
  
  red, Adj: λP.(λx.P(x) ∧ red′(x))
  
  in, P: λy.λP.λx.(P(x) ∧ in′(y)(x))
  
  the, Det: ι
  
  a, Det: some$_{\text{Ind}}$ → Bool → (Ind → Bool) → Bool
  
  runs, V: run$_{\text{Ind}}$ → Bool
  
  respects, V: respect$_{\text{Ind}}$ → Ind → Bool
  
  likes, V: like$_{\text{Ind}}$ → Ind → Bool
A grammar fragment

- \( \text{in}' \) is \( \text{Ind} \rightarrow \text{Ind} \rightarrow \text{Bool} \)
- \( \text{in} \overset{\text{def}}{=} \lambda y. \lambda P. \lambda x. (P(x) \land \text{in}'(y)(x)) \) is \( \text{Ind} \rightarrow (\text{Ind} \rightarrow \text{Bool}) \rightarrow (\text{Ind} \rightarrow \text{Bool}) \)
- \( \text{red}' \) is \( \text{Ind} \rightarrow \text{Bool} \)
- \( \text{red} \overset{\text{def}}{=} \lambda P. (\lambda x. (P(x) \land \text{red}'(x)) \) is \( (\text{Ind} \rightarrow \text{Bool}) \rightarrow (\text{Ind} \rightarrow \text{Bool}) \)
Adjective and PP modification

1. \( N' : \lambda x.\text{car}(x) \land \text{in}'(\text{paloalto})(x) \land \text{red}'(x) \)

   \[
   \text{Adj} : \lambda P. (\lambda x. P(x) \land \text{red}'(x))
   \]

   \[
   N' : \lambda x. (\text{car}(x) \land \text{in}'(\text{paloalto})(x))
   \]

   \[
   \text{red}
   \]

   \[
   \text{car}
   \]

   \[
   \text{in}
   \]

   \[
   \text{Palo Alto}
   \]

   \[
   \text{P}: \lambda y. \lambda P. \lambda x. (P(x) \land \text{in}'(y)(x))
   \]

   \[
   \text{NP}: \text{paloalto}
   \]

2. \( N' : \lambda x.\text{car}(x) \land \text{in}'(\text{paloalto})(x) \land \text{red}'(x) \)

   \[
   \text{Adj} : \lambda P. (\lambda x. P(x) \land \text{red}'(x))
   \]

   \[
   N' : \lambda x. (\text{car}(x) \land \text{in}'(\text{paloalto})(x))
   \]

   \[
   \text{red}
   \]

   \[
   \text{car}
   \]

   \[
   \text{in}
   \]

   \[
   \text{NP}: \text{paloalto}
   \]

   \[
   \text{Palo Alto}
   \]
Intersective adjectives

- Syntactic ambiguity is spurious: you get the same semantics either way
- Database evaluation is possible via a table join

Non-intersective adjectives

- For non-intersective adjectives get different semantics depending on what they modify
  - overpriced(in(paloalto)(house))
  - in(paloalto)(overpriced(house))
- But probably won’t be able to evaluate it on database!
Why things get more complex

- When doing predicate logic did you wonder why:
  - *Kathy runs* is \( \text{run}(kathy) \)
  - *no kid runs* is \( \neg(\exists x)(\text{kid}(x) \land \text{run}(x)) \)
- Somehow the NP’s meaning is wrapped around the predicate
- Or consider why this argument doesn’t hold:
  - Nothing is better than a life of peace and prosperity.
    A cold egg salad sandwich is better than nothing.
    A cold egg salad sandwich is better than a life of peace and prosperity.
- The problem is that *nothing* is a quantifier
Generalized Quantifiers

• We have a reasonable semantics for *red car in Palo Alto* as a property from \( \text{Ind} \to \text{Bool} \)

• How do we represent noun phrases like *the red car in Palo Alto* or *every red car in Palo Alto*?

• \( [\iota](P) = a \text{ if } (P(b) = 1 \text{ iff } b = a) \)

  undefined, otherwise

• The semantics for *the* following Bertrand Russell, for whom *the* \( x \) meant the unique item satisfying a certain description
Generalized Quantifiers

- *red car in Palo Alto*

  select Cars.obj from Cars, Locations, Red where
  Cars.obj = Locations.obj AND
  Locations.place = ‘paloalto’ AND Cars.obj = Red.obj

  (here we assume the unary relations have one field, obj).
Generalized Quantifiers

- **the red car in Palo Alto**

- NP : \(\iota(\lambda x. \text{car}(x) \land \text{in'}(\text{paloalto})(x) \land \text{red'}(x))\)

- Det : \(\iota\)

- N' : \(\lambda x. \text{car}(x) \land \text{in'}(\text{paloalto})(x) \land \text{red'}(x)\)

- tree:

  - **the**

  - **red car in Palo Alto**

- **the red car in Palo Alto**

  select Cars.obj from Cars, Locations, Red where
  Cars.obj = Locations.obj AND
  Locations.place = 'paloalto' AND Cars.obj = Red.obj

  having count(*) = 1
Generalized Quantifiers

• What then of every red car in Palo Alto?
• A generalized determiner is a relation between two properties, one contributed by the restriction from the N’, and one contributed by the predicate quantified over:

  \((\text{Ind} \to \text{Bool}) \to (\text{Ind} \to \text{Bool}) \to \text{Bool}\)

• Here are some determiners

  \(\text{some}^2(\text{kid})(\text{run}) \equiv \text{some}(\lambda x.\text{kid}(x) \land \text{run}(x))\)

  \(\text{every}^2(\text{kid})(\text{run}) \equiv \text{every}(\lambda x.\text{kid}(x) \to \text{run}(x))\)
Generalized Quantifiers

- Generalized determiners are implemented via the quantifiers:

  \[
  \text{every}(P) = 1 \text{ iff } (\forall x)P(x) = 1;
  \]
  i.e., if \( P = \text{Dom}_{\text{Ind}} \)

  \[
  \text{some}(P) = 1 \text{ iff } (\exists x)P(x) = 1; \text{ i.e., if } P \neq \emptyset
  \]
Generalized Quantifiers

- Every student likes the red car
  - $S : \text{every}^2(\text{student})(\text{like}(\tau(\lambda x.\text{car}(x) \land \text{red}'(x))))$

  \[
  \begin{array}{c}
  \text{Det} : \text{every}^2 & N' : \text{student} \\
  \text{every} & \text{student} \\
  \end{array}
  \begin{array}{c}
  \text{NP} : \text{every}^2(\text{student}) \\
  \text{VP} : \text{like}(\tau(\lambda x.\text{car}(x) \land \text{red}'(x))) \\
  \end{array}
  \begin{array}{c}
  \text{V} : \text{like} \\
  \text{Det} : \tau \\
  \end{array}
  \begin{array}{c}
  \text{NP} : \tau(\lambda x.\text{car}(x) \land \text{red}'(x)) \\
  \text{N}' : \lambda x.(\text{car}(x) \land \text{red}'(x)) \\
  \end{array}
  \begin{array}{c}
  \text{Adj} : \lambda P.(\lambda x.P(x) \land \text{red}'(x)) \\
  \text{N} : \text{car} \\
  \end{array}
  \begin{array}{c}
  \text{red} \\
  \text{car} \\
  \end{array}
\]
Representing proper nouns with quantifiers

• The central insight of Montague’s PTQ was to treat individuals as of the same type as quantifiers (as type-raised individuals):
  
  • *Kathy:* $\lambda P.P(\text{kathy})$

• Both good and bad

• The main alternative (which we use) is flexible *type shifting* – you raise the type of something when necessary.
Nominal type shifting

- Common patterns of nominal type shifting

\[ \text{Ind} \xleftarrow{\iota} \text{Bool} \xrightarrow{\lambda P.P(x)} \text{Ind} \rightarrow \text{Bool} \]

\[ R : \text{some}^2(\text{Ind} \rightarrow \text{Bool}) \rightarrow \text{Bool} \]

- \( R(x) = \lambda P.P(x) \)
- \( \text{some}^2(P) = \lambda Q. (Q \cap P) \neq \emptyset \)
- \( Q(x) = \lambda y. x = y \)

- In this diagram, \( R \) is exactly this basic type-raising function for individuals.
Noun phrase scope – following Hendriks (1993)

Value raising raises a function that produces an individual to one that produces a quantifier. If $\alpha : \sigma \rightarrow \text{Ind}$ then $\lambda x.\lambda P. P(\alpha(x)) : \sigma \rightarrow (\text{Ind} \rightarrow \text{Bool}) \rightarrow \text{Bool}$

Argument raising replaces an argument of a boolean function with a variable and applies the quantifier semantically binding the replacing variable. If $\alpha : \sigma \rightarrow \text{Ind} \rightarrow \tau \rightarrow \text{Bool}$ then $\lambda x_1.\lambda Q.\lambda x_3. Q(\lambda x_2.\alpha(x_1)(x_2)(x_3)) : \sigma \rightarrow (\text{Ind} \rightarrow \text{Bool}) \rightarrow \text{Bool} \rightarrow \tau \rightarrow \text{Bool}$

Argument lowering replaces a quantifier in a boolean function with an individual argument, where the semantics is calculated by applying the original function to the type raised argument. If $\alpha : \sigma \rightarrow ((\text{Ind} \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow \tau \rightarrow \text{Bool}$ then $\lambda x_1.\lambda x_2.\lambda x_3. \alpha(x_1)(\lambda P. P(x_2))(x_3) : \sigma \rightarrow \text{Ind} \rightarrow \tau \rightarrow \text{Bool}$
Every student runs

- \( S : \text{every}^2(\text{student})(\text{run}) \equiv \text{every}(\lambda x.\text{student}(x) \rightarrow \text{run}(x)) \)

\[
\begin{array}{c}
\text{NP : every}^2(\text{student}) \\
\text{VP : } \lambda Q.Q(\lambda x.\text{run}(x)) \\
\text{Det : every}^2 \\
\text{VP : run} \\
\text{N' : student} \\
\text{V : run} \\
\text{every} \\
\text{N : student} \\
\text{runs} \\
\text{student}
\end{array}
\]
Some kid broke every toy

- $S: \text{every}^2(\text{toy})(\lambda y_0.\text{some}^2(\text{kid})(\lambda x_5.\text{break}(y_o)(x_s))))$

- NP: some$^2$(kid)
  - Det: some$^2$
  - N: kid
    - some
    - kid

- VP: $\lambda S'.\text{every}^2(\text{toy})(\lambda y_0.S'(\lambda x_5.\text{break}(y_o)(x_s))))$
  - V: $\lambda O.\lambda S'.O(\lambda y_0.S'(\lambda x_5.\text{break}(y_o)(x_s))))$
  - V: $\lambda x_5.\lambda S.S(\lambda x_5.\text{break}(x_o)(x_s))$
    - V: $\lambda y.\lambda x.\text{break}(y)(x)$
      - broke
      - kid

- NP: every$^2$(toy)
  - Det: every$^2$
  - N: toy
    - every
    - toy
Some kid broke every toy

- $S : \text{some}^2(\text{kid})(\lambda y.\text{every}^2(\text{toy})(\lambda x.\text{break}(x)(y)))$

  NP : $\text{some}^2(\text{kid})$

    Det : $\text{some}^2$
    N' : kid
    some

    N : kid
    kid

  VP : $\lambda S.S(\lambda y.\text{every}^2(\text{toy})(\lambda x.\text{break}(x)(y)))$

    V : $\lambda y.\lambda S.S(\lambda y.\text{every}^2(\text{toy})(\lambda x.\text{break}(x)(y)))$

      V : $\lambda x.\text{break}(x)(y)$
      V : $\lambda y.\lambda x.\text{break}(y)(x)$

    NP : $\text{every}^2(\text{toy})$

      Det : $\text{every}^2$
      N' : toy
      every

      N : toy
      toy
Questions with answers!

- A yes/no question \((\text{Is Kathy running?})\) will be something of type \textbf{Bool}, checked on database.
- A content question \((\text{Who likes Kathy?})\) will be an \emph{open proposition}, that is something semantically of the type \textit{property} \((\textbf{Ind} \rightarrow \textbf{Bool})\), and operationally we will consult the database to see what individuals will make the statement true.
- We use a grammar with a simple form of gap-threading for question words.
Syntax/semantics for questions

- $S' : \beta(\alpha) \rightarrow \text{NP}[wh] : \beta$  $\text{Aux} 
  S : \alpha$
- $S' : \alpha \rightarrow \text{Aux} 
  S : \alpha$
- $\text{NP/NP}_z : z \rightarrow e$
- $S : \lambda z. F(\ldots z \ldots) \rightarrow S/\text{NP}_z : F(\ldots z \ldots)$
Syntax/semantics for questions

- **who**, NP[wh] : \( \lambda U.\lambda x. U(x) \land \text{human}(x) \)
- **what**, NP[wh] : \( \lambda U.U \)
- **which**, Det[wh] : \( \lambda P.\lambda V.\lambda x. P(x) \land V(x) \)
- **how_many**, Det[wh] : \( \lambda P.\lambda V.|\lambda x.P(x) \land V(x)| \)

- Where \( | \cdot | \) is the operation that returns the cardinality of a set (count).
Question examples

• \( S' : \lambda z. \text{like}(z)(\text{kathy}) \)

  \( \text{NP[wh]} : \lambda U.U \quad \text{Aux} \quad \text{S} : \lambda z. \text{like}(z)(\text{kathy}) \)
  \( \quad \text{S/NP}_z : \text{like}(z)(\text{kathy}) \)

  \( \text{NP} : \text{kathy} \quad \text{VP/NP}_z : \text{like}(z) \)
  \( \quad \text{Kathy} \quad \text{V} : \text{like} \quad \text{NP/NP}_z : z \)
  \( \quad \text{like} \quad \text{e} \)

• select liked from Likes where Likes.liker='Kathy'
Question examples

- $S' : \lambda x. \text{like}(x)(\text{kathy}) \land \text{human}(x)$

  NP[$wh$] : $\lambda U. \lambda x. U(x) \land \text{human}(x)$
  - Who
  - does

  $S : \lambda z. \text{like}(z)(\text{kathy})$

  S/NP$_Z$ : $\text{like}(z)(\text{kathy})$

  VP/NP$_Z$ : $\text{like}(z)$

  NP : $\text{kathy}$

  V : $\text{like}$

  NP/NP$_Z$ : $z$

  $\text{like}$

- select liked from Likes, Humans where Likes.liker='Kathy' AND Humans.obj = Likes.liked
Question examples

- \( S' : \lambda x.\text{car}(x) \land \text{like}(x)(kathy) \)

\[
\begin{align*}
\text{NP[wh]} : & \lambda V.\lambda x.\text{car}(x) \land V(x) \\
\text{Aux} : & \lambda z.\text{like}(z)(kathy) \\
\text{S/np} : & \text{like}(z)(kathy) \\
\text{np} : & \text{like}(z) (kathy) \\
\text{vp/np}_{z} : & \text{like}(z) \\
\text{np/np}_{z} : & \text{like} \\
\text{e} : & \text{like} \\
\text{kathy} : & \text{kathy} \\
\text{car} : & \text{car} \\
\text{n' : car} : & \text{car} \\
\text{np : kathy} : & \text{kathy} \\
\text{which} : & \text{which} \\
\text{n : car} : & \text{car} \\
\text{det :} & \lambda P.\lambda V.\lambda x.\text{P}(x) \land V(x) \\
\text{np : cars} : & \text{cars} \\
\text{np : like} : & \text{like} \\
\text{v : like} : & \text{like} \\
\end{align*}
\]

- select liked from Cars,Likes where Cars.obj=Likes.liked AND Likes.liker='Kathy'
Question examples

• $S' \equiv \lambda x. \text{car}(x) \land \text{every}^2(\text{student})(\text{like}(x))$

  \begin{align*}
    & \text{NP[wh]} \equiv \lambda V. \lambda x. \text{car}(x) \land V(x) \\
    & \text{Det} \equiv \lambda P. \lambda V. \lambda x. P(x) \land V(x) \\
    & \text{N'} \equiv \text{car} \\
    & \text{Aux} \equiv \lambda z. \text{every}^2(\text{student})(\text{like}(z)) \\
    & \text{S/NP} \equiv \text{every}^2(\text{student})(\text{like}(z)) \\
    & \text{NP} \equiv \text{every}^2(\text{student}) \\
    & \text{Det} \equiv \text{every}^2 \\
    & \text{N'} \equiv \text{student} \\
    & \text{VP} \\
    & \text{NP}_z \equiv \text{like}(z) \\
    & \text{V} \equiv \text{like} \\
    & \text{NP}_z \equiv z \\
    & \text{e}
  \end{align*}

• ???
Question examples

• *How many red cars in Palo Alto does Kathy like?*
  
  select count(*) from Likes, Cars, Locations, Reds where
  Cars.obj = Likes.liked AND Likes.liker = 'Kathy' AND
  Red.obj = Likes.liked AND Locations.place = 'Palo Alto'
  AND Locations.obj = Likes.liked

• *Did Kathy see the red car in Palo Alto?*
  
  select ‘yes’ where Seeings.seer = k AND Seeings.seen
  = (select Cars.obj from Cars, Locations, Red where
  Cars.obj = Locations.obj AND Locations.place = ‘paloalto’
  AND Cars.obj = Red.obj having count(*) = 1)
How many red cars in Palo Alto does Kathy like?
Did Kathy see the red car in Palo Alto?

\[ S' : \text{see}(\lambda x.\text{car}(x) \land \text{in'}(\text{paloalto})(x) \land \text{red'}(x))(kathy) \]
How could we learn such representations?

- After disengagement for many years, there has started to be very interesting work in this area:
How could we learn such representations?

- General approach (ZC05): Start with initial lexicon, category templates, and paired sentences and meanings:

  What states border Texas?

  \[ \lambda x. \text{state}(x) \land \text{borders}(x, \text{texas}) \]

- Learn lexical syntax/semantics for other words and learn to parse to logical form (parse structure is hidden).

- They successfully do iterative refinement of a lexicon and maxent parser
How can we reason with such representations?

• Logical reasoning is practical for certain domains (business rules, legal code, etc.) and has been used (see Blackburn and Bos 2005 for background).
• But our knowledge of the world is in general incomplete and uncertain.
• There is various recent work on handling restricted fragments of first order logic in probabilistic models
How can we reason with such representations?

- Undirected model:

- A recent attempt to apply this to natural language inference:

- Logical formulae are given weights which are grounded out in an undirected markov network