LING 581: Advanced Computational Linguistics

Lecture Notes

May 4th
Last Time

\[
\begin{align*}
1 & \text{s}(s(N, VP)) \rightarrow n(N), \text{vp}(VP).
2 & \text{s}(s(S_1, CONJ, S_2)) \rightarrow \text{lookahead}, \text{s}(S_1), \text{conj}(CONJ), \text{s}(S_2).
3 & \text{s}(s(\text{NEG}, S)) \rightarrow \text{neg}(\text{NEG}), \text{s}(S).
4 & \text{vp}(\text{vp}(VT, N)) \rightarrow \text{vt}(VT), \text{n}(N).
5 & \text{vp}(\text{vp}(VI)) \rightarrow \text{vi}(VI).
6 & \text{vt}(\text{vt}(\text{likes})) \rightarrow \text{[likes]}.
7 & \text{vi}(\text{vi}(\text{v(is), ap(boring)})) \rightarrow \text{[is, boring]}.
8 & \text{vi}(\text{vi}(\text{v(is), ap(hungry)})) \rightarrow \text{[is, hungry]}.
9 & \text{vi}(\text{vi}(\text{v(is), ap(cute)})) \rightarrow \text{[is, cute]}.
10 & \text{n}(\text{n(james)}) \rightarrow \text{[james]}.
11 & \text{n}(\text{n(sophia)}) \rightarrow \text{[sophia]}.
12 & \text{n}(\text{n(jack)}) \rightarrow \text{[jack]}.
13 & \text{conj}(\text{conj}(\text{and})) \rightarrow \text{[and1]}.
14 & \text{conj}(\text{conj}(\text{or})) \rightarrow \text{[or1]}.
15 & \text{neg}(\text{neg}) \rightarrow \text{[it, is, not, the, case, that]}.
16 & \text{map_conj}(\text{and, and1}).
17 & \text{map_conj}(\text{or, or1}).
18 & \text{lookahead}(\text{List1}, \text{List2}) :-
19 & \quad \text{map_conj}(\text{Conj}, \text{Conj1}),
20 & \quad \text{append}(\text{Left}, [\text{Conj}|\text{Right}], \text{List1}),
21 & \quad \text{append}(\text{Left}, [\text{Conj1}|\text{Right}], \text{List2}),
22 & \quad !. \quad \% \text{cut}
\end{align*}
\]
Semantics

• We want to obtain a semantic parse for our sentences that we can “run” (i.e. evaluate) against the Prolog database (i.e. situation or possible world).

• So the semantic parse should be valid Prolog code (that we can call)

• We’ll need (built-in) member/2 and setof/3 defined in the following 2 slides (a quick review)
setof/3

- See  
  - [http://www.swi-prolog.org/pldoc/doc_for?
  object=section(2,'4.29',swi('/doc/Manual/allsolutions.html')))

- SWI Prolog built-in:

```prolog
setof(+Template, +Goal, -Set)  [ISO]
Equivalent to bagof/3, but sorts the result using sort/2 to get a sorted list of alternatives without duplicates.

bagof(+Template, :Goal, -Bag)  [ISO]
Unify Bag with the alternatives of Template, if Goal has free variables besides the one sharing with Template bagof will backtracking over the alternatives of these free variables, unifying Bag with the corresponding alternatives of Template. The construct +Var^Goal tells bagof not to bind Var in Goal. bagof/3 fails if Goal has no solutions.
```

### 4.29 Finding all Solutions to a Goal

```prolog
findall(+Template, :Goal, -Bag)  [ISO]
Creates a list of the instantiations Template gets successively on backtracking over Goal and unifies the result with Bag.  
Succeeds with an empty list if Goal has no solutions. findall/3 is equivalent to bagof/3 with all free variables bound with the existential operator (^), except that bagof/3 fails when goal has no solutions.
```
setof/3

• Example:

?- listing(cute).
   :- dynamic cute/1.

   cute(sophia).
cute(james).

   true.

?- setof(X,cute(X),Set).
   Set = [james, sophia].

?- retractall(cute(_)).
   true.

?- listing(cute).
   :- dynamic cute/1.

   true.

?- setof(X,cute(X),Set).
   Set = [james, sophia].

   false.

?- bagof(X,cute(X),Set).
   false.

?- findall(X,cute(X),Set).
   Set = [].

?-
member/2

• See

member(?Elem, ?List)
True if Elem is a member of List. The SWI-Prolog definition differs from the classical one. Our definition avoids unpacking each list element twice and provides determinism on the last element. E.g. this is deterministic:

```
member(X, [One]).
?- assert(cute(james)).
true.
?- assert(cute(sophia)).
ture.
?- listing(cute).
:- dynamic cute/1.
cute(james).
cute(sophia).
?- setof(X, cute(X), Set), member(Y, Set).
Set = [james, sophia],
Y = james ;
Set = [james, sophia],
Y = sophia.
```
Basic Lexical Entries

(8) For any situation (or circumstance) $V$,

a. $[\text{Jack}]^V = \text{Jack}'$

b. $[\text{Sophia}]^V = \text{Sophia}'$

c. $[\text{James}]^V = \text{James}'$

d. $[\text{is boring}]^V = \{ x : x \text{ is boring in } V \}$.
   (The set of those individuals that are boring in $V$.)

e. $[\text{is hungry}]^V = \{ x : x \text{ is hungry in } V \}$

f. $[\text{is cute}]^V = \{ x : x \text{ is cute in } V \}$

g. $[\text{likes}]^V = \{ <x, y> : x \text{ likes } y \text{ in } V \}$
   (The set of ordered pairs of individuals such that the first likes the second in $V$.)


Semantics

Logical Connectives

We can think of the semantic values of logical connectives in natural language as functions that map truth values into truth values.

(9) For any situation $V$,

a. $[[\text{it is not the case}]]^V = \begin{bmatrix} 1 \rightarrow 0 \\ 0 \rightarrow 1 \end{bmatrix}$

b. $[[\text{and}]]^V = \begin{bmatrix} < 1, 1 > \rightarrow 1 \\ < 1, 0 > \rightarrow 0 \\ < 0, 1 > \rightarrow 0 \\ < 0, 0 > \rightarrow 0 \end{bmatrix}$

c. $[[\text{or}]]^V = \begin{bmatrix} < 1, 1 > \rightarrow 1 \\ < 1, 0 > \rightarrow 1 \\ < 0, 1 > \rightarrow 1 \\ < 0, 0 > \rightarrow 0 \end{bmatrix}$
Semantics

Interpretive Rules for Each Syntactic Rule

- $[A \ B \ C]$ is equivalent to $\overset{\text{A}}{\begin{array}{c} B \\ C \end{array}}$

- $[[A \ B \ C]]$ stands for the semantic value of $\overset{\text{A}}{\begin{array}{c} B \\ C \end{array}}$

- If $g$ is a function and $u$ is a possible argument for $g$, $g(u)$ indicates the result of applying $g$ to $u$.

(10)  
   a. $[[S \ N \ VP]]^V = 1$ iff $[[N]]^V \in [[VP]]^V$ and 0 otherwise.
   b. $[[S \ S1 \ conj \ S2]]^V = [[\text{conj}]]^V(<[[S1]]^V,[[S2]]^V>)$
   c. $[[S \ neg \ S]]^V = [[\text{neg}]]^V([[S]]^V)$
   d. $[[VP \ V_t \ N]]^V = \{x: <x, [[N]]^V> \in [[V_t]]^V\}$
   e. If $A$ is a category and $a$ is a lexical entry or a lexical category and $\Delta = [A \ a]$, then $[[\Delta]]^V = [[a]]^V$
(11) Jack is hungry.

S
  \[ N \quad VP \]
    \[ Jack \quad V_i \]
      \[ is\ hungry \]

1 iff Jack' ∈ \{ x: x is hungry in V \}

  Jack'
    \{ x: x is hungry in V \}
      \{ x: x is hungry in V \}
        \{ x: x is hungry in V \}
Semantics: Implementation

• Desired implementation:

```prolog
?- s(X,[jack,is,hungry],[]).
X = (setof(_G312, hungry(_G312), _G307), member(jack, _G307)) ;
false.

?- s(X,[jack,is,hungry],[]), call(X).
false.

?- assert(hungry(jack)).
true.

?- s(X,[jack,is,hungry],[]), call(X).
X = (setof(_G359, hungry(_G359), [jack]), member(jack, [jack])) ;
false.
```

Note: we are bypassing the (explicit) construction of the syntax tree

*Imagine if the Penn Treebank was labeled using a semantic representation*
Let’s write the semantic grammar to handle “Jack is hungry”
– first, let’s introduce a bit of notation (lambda calculus)
– \( \lambda = \) function
– \( \lambda x. x + 1 \) denotes a function that takes an argument \( x \) and computes value \( x + 1 \)
  • (a period separates the argument from the function body)
– \( (\lambda x. x + 1)(5) \) means apply 5 to the lambda function
  • substitute 5 in place of \( x \) and evaluate
  • answer = 6
Semantics: Implementation

Syntax:

1. $s(s(N, VP)) \rightarrow n(N), vp(VP)$.
2. $vp(vp(VI)) \rightarrow vi(VI)$.
3. $n(n(jack)) \rightarrow [jack]$.
4. $vi(v(v(is), ap(hungry))) \rightarrow [is, hungry]$.

$setof(X, hungry(X), S), member(jack, S)$

1 if $Jack' \in \{x: x \text{ is hungry in } V\}$
Semantics: Implementation

• Semantic grammar:

1. \( s((\text{Fn, member}(N, X))) \rightarrow n(N), \text{vp}(\text{lambda}(X, \text{Fn})) \).
2. \( \text{vp}(\text{VI}) \rightarrow \text{vi}(\text{VI}) \).
3. \( n(\text{jack}) \rightarrow [\text{jack}] \).
4. \( \text{vi}(\text{lambda}(S, \text{setof}(X, \text{hungry}(X), S))) \rightarrow [\text{is, hungry}] \).

?- \( s(X, [\text{jack, is, hungry}], []) \).
\( X = \text{setof}(_G678, \text{hungry}(_G678), _G673), \text{member}(\text{jack}, _G673)) \).
Semantics: Implementation

- Semantic grammar:
  \[s((\text{Fn}, \text{member}(N,X))) \rightarrow n(N), \text{vp}(\lambda(X,\text{Fn})).\]
  \[\text{vp}(\text{VI}) \rightarrow \text{vi}(\text{VI}).\]
  \[n(\text{jack}) \rightarrow [\text{jack}].\]
  \[\text{vi}(\lambda(X,\text{hungry}(X),S)) \rightarrow [\text{is},\text{hungry}].\]

?- dynamic hungry/1.
true.

?- s(X,[jack,\text{is},\text{hungry}],[]), call(X).
false.

?- assert(hungry(jack)).
true.

?- s(X,[jack,\text{is},\text{hungry}],[]), call(X).
X = (\text{setof}(\text{X}, \text{hungry}(\text{X}), [\text{jack}]), \text{member}(\text{jack}, [\text{jack}]))).
Semantics: Implementation

• More examples of computation:

(12) Sophia likes James.

(13) It is not the case that James is cute.

?- s(X,[sophia,likes,james],[]).
X = (setof(_G315, likes(_G315, james), _G307), member(sophia, _G307)) ;
false.

?- s(X,[it,is,not,the,case,that,james,is,cute],[]).
X = (\+ (setof(_G350, cute(_G350), _G345), member(james, _G345))) ;
false.
Semantics: Implementation

• More examples of computation:

(12) Sophia likes James.

\[\begin{align*}
\text{?- s(X,[sophia,likes,james],[]).}\ \\
\text{X = (setof(_G315, likes(_G315, james), _G307), member(sophia, _G307)) ;}
\end{align*}\]

\begin{align*}
^1 & s((F_n, member(N,X))) \rightarrow n(N), \ vp(\text{lambda}(X,F_n)). \\
^2 & vp(VI) \rightarrow vi(VI). \\
^3 & vp(F_n) \rightarrow vt(\text{lambda}(X,F_n)), n(Y), \ \{X=Y\}. \\
^4 & n(james) \rightarrow [\text{james}]. \\
^5 & n(sophie) \rightarrow [\text{sophie}]. \\
^6 & vi(\text{lambda}(S,\text{setof}(X,\text{hungry}(X),S))) \rightarrow [\text{is, hungry}]. \\
^7 & vt(\text{lambda}(Y,\text{lambda}(S,\text{setof}(X,\text{likes}(X,Y),S)))) \rightarrow [\text{likes}].
\end{align*}
Semantics: Implementation

• More examples of computation:

   (13) It is not the case that James is cute.

   ?- s(X,[it,is,not,the,case,that,james,is,cute],[]).
   X = (\+ (setof(_G350, cute(_G350), _G345), member(james, _G345)))) ;
   false.

1  s((Fn,member(N,X))) --> n(N), vp(lambda(X,Fn)).
2  s(NS) --> neg(S,NS), s(S).
3  vp(VI) --> vi(VI).
4  vp(Fn) --> vt(lambda(X,Fn)), n(Y), \{X=Y\}.
5  n(james) --> [james].
6  n(sophie) --> [sophie].
7  vi(lambda(S,setof(X,hungry(X),S))) --> [is,hungry].
8  vi(lambda(S,setof(X,cute(X),S))) --> [is,cute].
9  vt(lambda(Y,lambda(S,setof(X,likes(X,Y),S)))) --> [likes].
10  neg(P,\(+ P)) --> [it,is,not,the,case,that].
Semantics

Compositional Semantics for F1 (cont.)

(14) It is not the case that [Jack is hungry or Sophia is boring].

\[
[S4]^V = [\text{Neg}]^V ([S3]^V) = \\
\text{1 if } J' \notin \{x: x \text{ is hungry in } V\} \text{ and } S' \notin \{x: x \text{ is boring in } V\}
\]

\[
[Neg]^V = \begin{bmatrix}
1 & 0 \\
0 & 1 \\
\end{bmatrix}
\]

\[
[S3]^V = [\text{Conj}]^V (< [S2]^V, [S1]^V >) = \text{0 if } J' \notin \{x: x \text{ is hungry in } V\} \text{ and } S' \notin \{x: x \text{ is boring in } V\}
\]

\[
[\text{it is not the case that}]^V = \\
\begin{bmatrix}
1 & 0 \\
0 & 1 \\
\end{bmatrix}
\]

\[
[S2]^V = 1 \text{ if } [N2]^V \in [VP2]^V = 1 \text{ if } J' \in \{x: x \text{ is hungry in } V\}
\]

\[
[N2]^V = J' \\
[VP2]^V = \{x: x \text{ is hungry in } V\}
\]

\[
[\text{Jack}]^V = J' \\
[VP2]^V = \{x: x \text{ is hungry in } V\}
\]

\[
[\text{or}]^V = \\
\begin{bmatrix}
< 1, 1 > & 1 \\
< 1, 0 > & 1 \\
< 0, 1 > & 1 \\
< 0, 0 > & 0 \\
\end{bmatrix}
\]

\[
[S1]^V = 1 \text{ if } [N1]^V \in [VP1]^V = 1 \text{ if } S' \in \{x: x \text{ is boring in } V\}
\]

\[
[S1]^V = \begin{bmatrix}
1 & 0 \\
0 & 1 \\
\end{bmatrix}
\]

\[
[N1]^V = S' \\
[VP1]^V = \{x: x \text{ is boring in } V\}
\]

\[
[\text{Sophia}]^V = S' \\
[VP1]^V = \{x: x \text{ is boring in } V\}
\]

\[
[\text{is hungry}]^V = \{x: x \text{ is hungry in } V\}
\]

\[
[\text{is boring}]^V = \{x: x \text{ is boring in } V\}
\]
Semantics: Implementation

• Scope of negation: wide or narrow

(14) It is not the case that [Jack is hungry or Sophia is boring].

?- s(X,[it,is,not,the,case,that,jack,is,hungry,or,sophia,is,boring],[]).
X = (\+ (setof(_G431, hungry(_G431), _G426), member(jack, _G426));setof(_G449, boring(_G449),
_G444), member(sophia, _G444)) ;
X = (\+ (setof(_G395, hungry(_G395), _G390), member(jack, _G390);setof(_G413, boring(_G413), _G408), member(sophia, _G408))) ;
false.

?-
Grammar: revised

```
g2.pl

1 s(s(N,VP), (Stmt, member(SN,Q)))  -->  n(N,SN), vp(VP, lambda(Q, Stmt)).
2 s(s(S1,C,S2), (SS1, SS2))  -->  lookahead, s(S1, SS1), conj(C, and), s(S2, SS2).
3 s(s(S1,C,S2), (SS1; SS2))  -->  lookahead, s(S1, SS1), conj(C, or), s(S2, SS2).
4 s(s(NEG, S), \(+\) SS))  -->  neg(NEG), s(S, SS).
5 vp(vp(VT,N), Stmt)  -->  vt(VT, lambda(SN, Stmt)), n(N, SN).
6 vp(vp(VI), SVI)  -->  vi(VI, SVI).
7 vt(vt(likes), lambda(Y, lambda(L, findall(X, likes(X,Y), L))))  -->  [likes].
8 vi(vi(is_boring), lambda(L, findall(X, boring(X), L)))  -->  [is, boring].
9 vi(vi(is_hungry), lambda(L, findall(X, hungry(X), L)))  -->  [is, hungry].
10 vi(vi(is_cute), lambda(L, findall(X, cute(X), L)))  -->  [is, cute].
11 n(n(james), james)  -->  [james].
12 n(n(sophia), sophia)  -->  [sophia].
13 n(n(jack), jack)  -->  [jack].
14 conj(conj(and), and)  -->  [and1].
15 conj(conj(or), or)  -->  [or1].
16 neg(neg)  -->  [it, is, not, the, case, that].
```
Grammar: revised

```
map_conj(and, and1).
map_conj(or, or1).

lookahead(List1, List2) :-
    map_conj(Conj, Conj1),
    append(Left, [Conj | Right], List1),
    append(Left, [Conj1 | Right], List2),
    !.           % cut
```
Evaluation

• Check our computer implementation on...

(15) Jack is hungry, and it is not the case that James likes Jack.

- Situation $V'$: Jack is hungry, James does not like him.

- Situation $V''$: Jack is hungry, James likes him.

(16) [It is not the case that Jack is hungry] or [Sophia is boring].

- Situation $V'''$: Jack is hungry, Sophia is boring.

- Situation $V''''$: Jack is hungry, Sophia is not boring.
Quan7fiers

• Not all noun phrases (NPs) are (by nature) directly referential like names
• Quantifiers: “something to do with indicating the quantity of something”
• Examples:
  – every child
  – nobody
  – two dogs
  – several animals
  – most people

nobody has seen a unicorn
means roughly (Prolog-style):
?- setof(X,(person(X), seen(X,Y), unicorn(Y)),Set),cardinality(Set,0).
Quantifiers

- Database

```
1  person(a).
2  person(b).
3  person(c).
4  seen(c,u).
5  seen(c,d).
6  seen(c,v).
7  unicorn(u).
8  unicorn(v).
9  unicorn(u).
10 unicorn(v).
```

- setof vs. findall (recall last lecture)

```
?- setof(X,(person(X), seen(X,Y), unicorn(Y)),Set),cardinality(Set,0).
```

Nobody has seen a unicorn means roughly (Prolog-style):

```
?- setof(X,(person(X), seen(X,Y), unicorn(Y)),Set),cardinality(Set,0).
```
Quantifiers

• Semantic compositionality:
  – *elements of a sentence combine in piecewise fashion to form an overall (propositional) meaning for the sentence*

• Example:
  – (4) Every baby cried
    – **Word**               **Meaning**
    – cried                  cried(X).
    – baby                   baby(X).
    – every                  ?
    – every baby cried       *proposition* (True/False)
    – that can be evaluated for a given situation
Quantifiers

• Scenario (Possible World):
  – suppose there are three babies...
    • baby(noah).
    • baby(merrill).
    • baby(dani).
  – all three cried
    • cried(noah).
    • cried(merrill).
    • cried(dani).
  – only Dani jumped
    • jumped(dani).
  – Noah and Dani swam
    • swam(noah).
    • swam(dani).

<table>
<thead>
<tr>
<th>(6)</th>
<th>every baby</th>
<th>exactly one baby</th>
<th>most babies</th>
</tr>
</thead>
<tbody>
<tr>
<td>cried</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>jumped</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>swam</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

• think of quantifiers as “properties-of-properties”
• every_baby(P) is a proposition
• P: property
• every_baby(P) true for P=cried
• every_baby(P) false for P=jumped and P=swam
Quantifiers

• **think of quantifiers as “properties-of-properties”**
  – every_baby(P) **true** for P=cried
  – every_baby(P) **false** for P=jumped and P=swam

• **Generalized Quantifiers**
  – the idea that quantified NPs represent sets of sets
  – *this idea is not as weird as it sounds*
  – we know
    • every_baby(P) is true for certain properties
  – view
    • every_baby(P) = set of all properties P for which this is true
  – in our scenario
    • every_baby(P) = {cried}
  – we know *cried* can also be view as a set itself
    • cried = set of individuals who cried
  – in our scenario
    • cried = {noah, merrill, dani}
Quantifiers

- **how do we define the expression** every_baby(P)?
- **(Montague-style)**
  - every_baby(P) is shorthand for
    - for all individuals X, baby(X) -> P(X)
    - -> : *if-then* (*implication* : logic symbol)
- written another way
  - *(lambda calculus-style):*
    - \( \lambda P.[\forall X.[baby(X) \rightarrow P(X)]] \)
    - \( \forall: *for all* (universal quantifier: logic symbol) \)

- **Example:**
  - every baby walks
    - for all individuals X, baby(X) -> walks(X)
  - [\textit{NP every baby} \textit{VP walks}]
    - \( \lambda P. [\forall X.[baby(X) \rightarrow P(X)]](walks) \)
    - \( \forall X.[baby(X) \rightarrow walks(X)] \)
Quantifiers

• **how do we define this Prolog-style?**
• **Example:**
  - every baby walks
  - \([\text{NP} \text{every baby} \ [\text{VP} \text{walks}]

  \quad \cdot \ \lambda P. (\forall X (\text{baby}(X) \rightarrow P(X)))(\text{walks})
  \quad \cdot \ \forall X (\text{baby}(X) \rightarrow \text{walks}(X))

• **Possible World (Prolog database):**
  - \(\text{:- dynamic baby/1.} \)  \((\text{allows us to modify the baby database online})\)
  - \(\text{baby(a). baby(b).} \)
  - \(\text{walks(a). walks(b). walks(c).} \)
  - \(\text{individual(a). individual(b). individual(c).} \)

• **What kind of query would you write?**

• **One Possible Query (every means there are no exceptions):**
  - \(? \ - \ \downarrow \ (\text{baby}(X), \ \downarrow \ \text{walks}(X)). \)  \((\text{NOTE: may need a space between } \downarrow \text{ and } ( \text{here})\)
  - \(\text{Yes (TRUE)} \)

  - \(? \ - \ \text{baby}(X), \ \downarrow \ \text{walks}(X). \)
  - \(\text{No} \)

  - \(? \ - \ \text{assert(baby(d)).} \)
  - \(? \ - \ \text{baby}(X), \ \downarrow \ \text{walks}(X). \)
  - \(X = d \)
  - \(\text{Yes} \)

**Using no exception idea**
that \(\forall X P(X)\)

is the same as \(\neg \exists X \neg P(X)\)

\(\exists = \text{“there exists” (quantifier)}\)

(implicitly: all Prolog variables

are existentially quantified variables)
Recall: *Truth Tables*

- **De Morgan’s Rule**
- \( \neg(P \lor Q) = \neg P \land \neg Q \)

\[
\begin{array}{c|c|c}
P & Q & \neg(P \lor Q) \\
\hline
T & T & F \\
T & F & T \\
F & T & T \\
F & F & T \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\neg P & \land & \neg Q & \\
\hline
T & F & F & \\
F & T & F & \\
T & F & T & \\
F & F & F & \\
\end{array}
\]

\( \neg(P \lor Q) = T \) only when both \( P \) and \( Q \) are \( F \)

\( \neg P \land \neg Q = T \) only when both \( P \) and \( Q \) are \( F \)

Hence, \( \neg(P \lor Q) \) is equivalent to \( \neg P \land \neg Q \)
Conversion into Prolog

**Note:**
\(+ (\text{baby}(X), +\text{walks}(X))\) is Prolog for \(\forall X (\text{baby}(X) \rightarrow \text{walks}(X))\)

**Steps:**
- \(\forall X (\text{baby}(X) \rightarrow \text{walks}(X))\)
- \(\forall X (\neg\text{baby}(X) v \text{walks}(X))\)
  - (since \(P \rightarrow Q = \neg P v Q\), see truth tables from two lectures ago)
- \(\neg \exists X \neg (\neg\text{baby}(X) v \text{walks}(X))\)
  - (since \(\forall X P(X) = \neg \exists X \neg P(X)\), no exception idea)
- \(\neg \exists X (\text{baby}(X) \land \neg\text{walks}(X))\)
  - (by De Morgan’s rule, see truth table from last slide)
- \(\neg (\text{baby}(X) \land \neg\text{walks}(X))\)
  - (can drop \(\exists X\) since all Prolog variables are basically existentially quantified variables)
- \(+ (\text{baby}(X) \land +\text{walks}(X))\)
  - (+ = Prolog negation symbol)
- \(+ (\text{baby}(X), +\text{walks}(X))\)
  - (, = Prolog conjunction symbol)
Quantifiers

- **how do we define this Prolog-style?**
- **Example:**
  - every baby walks
  - \([_{NP \text{ every baby}}]_{VP \text{ walks}}\)
    - \(\lambda P. [\forall X. [\text{baby}(X) \rightarrow P(X)]](\text{walks})\)
    - \(\forall X. [\text{baby}(X) \rightarrow \text{walks}(X)]\)
- **Another situation (Prolog database):**
  - :- dynamic baby/1.
  - :- dynamic walks/1.
- **Does ?- \(+ (\text{baby}(X), \text{\+ walks}(X)).\)** still work?
- **Yes because**
  - ?- baby(X), \text{\+ walks}(X).
  - No
  cannot be satisfied
Quantifiers

- **how do we define the expression every_baby(P)?**
- **(Montague-style)**
  - every_baby(P) is shorthand for
    - $\lambda P. [\forall X. \text{baby}(X) \rightarrow P(X)]$

- **(Barwise & Cooper-style)**
  - think directly in terms of sets
  - *leads to another way of expressing the Prolog query*

- **Example**: every baby walks
  - $\{X: \text{baby}(X)\}$  *set of all X such that baby(X) is true*
  - $\{X: \text{walks}(X)\}$  *set of all X such that walks(X) is true*

- **Subset relation ($\subseteq$)**
  - $\{X: \text{baby}(X)\} \subseteq \{X: \text{walks}(X)\}$  *the “baby” set must be a subset of the “walks” set*
Quantifiers

(Barwise & Cooper-style)

• think directly in terms of sets
• leads to another way of expressing the Prolog query

• Example: every baby walks
• \{X: baby(X)\} \subseteq \{X: walks(X)\} the “baby” set must be a subset of the “walks” set

• How to express this as a Prolog query?

• Queries:
• ?- setof(X,baby(X),L1). \textit{L1 is the set of all babies in the database}
• ?- setof(X,walks(X),L2). \textit{L2 is the set of all individuals who walk}

Need a Prolog definition of the subset relation. This one, for example:
subset([],_).
subset([X|L1],L2) :- member(X,L2), subset(L1,L2).
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
Quantifiers

- **Example**: every baby walks
- \{X: \text{baby}(X)\} \subseteq \{X: \text{walks}(X)\}  
  *the “baby” set must be a subset of the “walks” set*
- **Assume the following definitions are part of the database**:
  
  ```prolog
  subset([],_).
  subset([X|_ ],L) :- member(X,L).
  member(X,[X|_ ]).  
  member(X,[ _|L]) :- member(X,L).
  ```

- **Prolog Query**:
  
  ```prolog
  ?- setof(X,baby(X),L1), setof(X,walks(X),L2), subset(L1,L2).
  ```
  
  - **True for world**:
    - \text{baby}(a). \text{baby}(b).
    - \text{walks}(a). \text{walks}(b). \text{walks}(c).
    
    \[\begin{align*}
    L_1 &= [a,b] \\
    L_2 &= [a,b,c] \\
    \text{?- subset}(L_1,L_2) &\text{ is true}
    \end{align*}\]

  - **False for world**:
    - \text{baby}(a). \text{baby}(b). \text{baby}(d).
    - \text{walks}(a). \text{walks}(b). \text{walks}(c).
    
    \[\begin{align*}
    L_1 &= [a,b,d] \\
    L_2 &= [a,b,c] \\
    \text{?- subset}(L_1,L_2) &\text{ is false}
    \end{align*}\]

Quantifiers

- **Example:** *every baby walks*
- *(Montague-style)* \( \forall X \ (\text{baby}(X) \rightarrow \text{walks}(X)) \)
- *(Barwise & Cooper-style)* \( \{X: \text{baby}(X)\} \subseteq \{X: \text{walks}(X)\} \)

- **how do we define every_baby(P)?**
  - *(Montague-style)* \( \lambda P. [ \forall X (\text{baby}(X) \rightarrow P(X))] \)
  - *(Barwise & Cooper-style)* \( \{X: \text{baby}(X)\} \subseteq \{X: P(X)\} \)

- **how do we define every?**
  - *(Montague-style)* \( \lambda P_1. [\lambda P_2. [ \forall X (P_1(X) \rightarrow P_2(X))]] \)
  - *(Barwise & Cooper-style)* \( \{X: P_1(X)\} \subseteq \{X: P_2(X)\} \)
Quantifiers

• how do we define the expression *every*?

- (Montague-style) \( \lambda P_1. (\lambda P_2. [\forall X (P_1(X) \rightarrow P_2(X))]) \)

• *Let’s look at computation in the lambda calculus...*

• **Example:** *every man likes John*
  - **Word**
  - *every* \( \lambda P_1. (\lambda P_2. [\forall X (P_1(X) \rightarrow P_2(X))]) \)
  - *man* man
  - *likes* \( \lambda Y. [\lambda X. [X \text{ likes } Y]] \)
  - *John* John

• **Syntax:** \([s \ [np \ [q \ every] \ [n \ man]] \ [vp \ [v \ likes] \ [np \ John]]]\)
Quantifiers

- **Example:** \([ S \ [ N_p \ [ Q \ every ] \ [ N \ man ] ] \ [ V_p \ [ V \ likes ] \ [ N_p \ John ] ] ]\)
  - *Word*  
    - *every* \( \lambda P_1.\lambda P_2. [ \forall X (P_1(X) -> P_2(X))] \)
    - *man* \( \text{man} \)
    - *likes* \( \lambda Y. [\lambda X. [ X \ likes \ Y] ] \)
    - *John* \( \text{John} \)

- **Logic steps:**
  \( [ Q \ every ] [ N \ man ] \) \( \lambda P_1.\lambda P_2. [ \forall X (P_1(X) -> P_2(X)) ](\text{man}) \)
  \( [ Q \ every ] [ N \ man ] \) \( \lambda P_2. [ \forall X (\text{man}(X) -> P_2(X))] \)
  \( [ V_p \ [ V \ likes ] [ N_p \ John ] ] \) \( \lambda Y. [\lambda X. [ X \ likes \ Y]](\text{John}) \)
  \( [ V_p \ [ V \ likes ] [ N_p \ John ] ] \) \( \lambda X. [ X \ likes \ John] \)
  \( [ S \ [ N_p \ [ Q \ every ] [ N \ man ] ] \ [ V_p \ [ V \ likes ] [ N_p \ John ] ] ] \)
  \( \lambda P_2. [ \forall X (\text{man}(X) -> P_2(X))](\lambda X. [ X \ likes \ John]) \)
  \( \forall X (\text{man}(X) -> \lambda X. [ X \ likes \ John ](X) \)
  \( \forall X (\text{man}(X) -> [ X \ likes \ John ] \)
Quantifiers

• Prolog is kinda first order logic ...
  – no quantifier variables
Quantifiers

• Example:
  – $\lambda P_1. [\lambda P_2. [\forall X (P_1(X) \rightarrow P_2(X))]]$
  – $\lambda\lambda (P1,\lambda (P2, (\rightarrow (P1(X), \rightarrow P2(X))))))$

    ```
    |- lambda(P1,lambda(P2,(\rightarrow (P1(X), \rightarrow P2(X))))))
    \l Error: Syntax error: Operator expected
    \l Error: lambda(P1,lambda(P2, (\rightarrow (P1
    \l Error: ** here **
    \l Error: (X), \rightarrow P2(X))))).
    |- 
    ```

• Example:
  – $\{X: P(X)\}$
  – `setof(X,P(X),Set)`

    ```
    |- setof(X,P(X),Set).
    \l Error: Syntax error: Operator expected
    \l Error: setof(X,P
    \l Error: ** here **
    \l Error: (X),Set).
    |- 
    ```
Quantifiers

- **Example:**
  - \{X: P(X)\}
  - **Illegal:** setof(X,P(X),Set)
  - **Alternate:** setof(X,call(P,X),Set)

### Database

<table>
<thead>
<tr>
<th>Person</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>person(a). X = a;</td>
</tr>
<tr>
<td>2</td>
<td>person(b). X = b;</td>
</tr>
<tr>
<td>3</td>
<td>person(c). X = c.</td>
</tr>
</tbody>
</table>

```
call(:Goal)                                   [ISO]
Invoke Goal as a goal. Note that clauses may have variables as
subclauses, which is identical to call/1.

call(:Goal, +ExtraArg1, ...)                  
Append ExtraArg1, ExtraArg2, ... to the argument list of Goal
and call the result. For example, call(plus(1), 2, X) will call
plus(1, 2, X), binding X to 3.

The call/[2-..] construct is handled by the compiler, which implies
that redefinition as a predicate has no effect. The predicates
call/[2-6] are defined as real predicates, so they can be handled
by interpreted code.
```
Quantifiers

• Example:
  \[ \lambda P_1.[\lambda P_2.[\forall X (P_1(X) \rightarrow P_2(X))]] \]

  Illegal: \hspace{1cm} \text{lambda}(P1, \text{lambda}(P2, (\text{\textbackslash + } (P1(X), \text{\textbackslash + } P2(X))))))

  Alternate: \hspace{1cm} \text{lambda}(P1, \text{lambda}(P2, (\text{\textbackslash + } (\text{call}(P1,X), \text{\textbackslash + } \text{call}(P2,X)))))}
Quantifiers

Part 3: Coordination

- Extend the grammars to handle
  - *Every man and every woman likes John*
Other Quantifiers

• Other quantifiers can also be expressed using set relations between two predicates:

Example:

*no:* \( \{X: P_1(X)\} \cap \{Y: P_2(Y)\} = \emptyset \)

\( \cap \) = set intersection
\( \emptyset \) = empty set

*no man smokes*

\( \{X: \text{man}(X)\} \cap \{Y: \text{smokes}(Y)\} = \emptyset \)

should evaluate to true for all possible worlds where there is no overlap between men and smokers
Other Quantifiers

- Other quantifiers can also be expressed using set relations between two predicates:

  Example:

  *some*: \( \{X: P_1(X)\} \cap \{Y: P_2(Y)\} \neq \emptyset \)

  \( \cap = \) set intersection

  \( \emptyset = \) empty set

  *some men smoke*

  \( \{X: \text{man}(X)\} \cap \{Y: \text{smokes}(Y)\} \neq \emptyset \)
Names as Generalized Quantifiers

• we’ve mentioned that names directly refer
  *here is another idea…*

• Conjunction
  – X and Y
    – both X and Y have to be of the same type
    – in particular, semantically...
    – we want them to have the same semantic type

• what is the semantic type of every baby?

Example

- every baby and John likes ice cream
  - \([_{\text{NP}}]_{\text{NP}}\) every baby and \([_{\text{NP}}}\) John] likes ice cream
- every baby likes ice cream
  - \{X: baby(X)\} ⊆ \{Y: likes(Y,ice_cream)\}
- John likes ice cream
  - ??? ⊆ \{Y: likes(Y,ice_cream)\}
- want everything to be a set (to be consistent)
- i.e. want to state something like
  - \(\{X: baby(X)\} \cup \{X: john(X)\}\) ⊆ \{Y: likes(Y,ice_cream)\}
- note: set union (\(\cup\)) is the translation of “and”
Downwards and Upwards Entailment (DE & UE)

- **Quantifier** *every* has semantics
  - \{X: P_1(X)\} \subseteq \{Y: P_2(Y)\}
  - e.g. every woman likes ice cream
  - \{X: woman(X)\} \subseteq \{Y:likes(Y, ice_cream)\}
- **Every** is DE for \(P_1\) and UE for \(P_2\)
- Examples:
  - (25) a. Every dog barks
  - b. Every Keeshond barks (valid)
  - c. Every animal barks (invalid)
    - semantically, “Keeshond” is a sub-property or subset with respect to the set “dog”
Downwards and Upwards Entailment (DE & UE)

• **Quantifier every** has semantics
  - \{X: P_1(X)\} \subseteq \{Y: P_2(Y)\}
  - e.g. every woman likes ice cream
  - \{X: woman(X)\} \subseteq \{Y:likes(Y,ice_cream)\}
• **Every** is DE for P_1 and UE for P_2
• Examples:
  • (25) a. Every dog barks
  • d. **Every dog barks loudly** (invalid)
  • c. Every dog makes noise (valid)
    – semantically, “barks loudly” is a subset with respect to the set “barks”, which (in turn) is a subset of the set “makes noise”