LING 581: Advanced Computational Linguistics

Lecture Notes

April 16th
Administrivia

• Factoid Question Answering homework
  – did you submit your simulation?
Semantics

• New topic!
  – We want computers to be able to understand sentences,
  – model the world,
  – compute meaning, truth values, entailments etc.
Meaning

• What is a meaning and how do we represent it?
  – difficult to pin down precisely for computers
  – even difficult for humans sometimes...

• Example: word *dog*
  – by reference to other words
    • Merriam-Webster: *a highly variable domestic mammal (Canis familiaris) closely related to the gray wolf*
  – translation
    • 犬 (*INU*, Japanese) = 狗 (*gou*, Chinese) = “dog” (English)
  – Computer:
    • meaning ➔ formal concept (or thought or idea)
    • “dog” maps to DOG
    • <word> maps to <concept>
    • need to provide a concept for every meaningful piece of language?
Understanding

• Suppose we write a computer program to compute the meaning of sentences
• Question: *does it understand sentences?*
• How do you know?
  • Ask questions?

• **Turing test:**
  – converse with a human, convince human the computer is a human

• **Searle’s Chinese room experiment** (adapted)
  – suppose we have a Perl/Python/Prolog program capable of processing Chinese, and we “run” the program manually
  – *i.e. we carry out the instructions of the program*
  – do we understand Chinese?

• Weak AI / Strong AI
Truth Conditions and Values

- What is the meaning of a sentence anyway?
- *What is Meaning?* (Portner 2005)
- Example: *the circle is inside the square*
  - We can draw a picture of scenarios for which the statement is true and the statement is false
- Proposition expressed by a sentence is its truth-conditions
  - “under what conditions a sentence is true”
  - i.e. **sets** of possible worlds (*aka* situations)
  - *truth-conditions* different from *truth-value*
Truth Conditions and Values

• Example:
  – The circle is inside the square and the circle is dark
  – What is the meaning of and here?
  – and = set intersection (of scenarios)
  – [The circle is inside the square] and [the circle is dark]

• Example:
  – Mary is a student and a baseball fan
  – and = set intersection (of ???)
  – Mary is [a student] and [a baseball fan]
Truth Conditions and Values

• Example:
  – Mary and John bought a book
  – Does and = set intersection?
  – Are Mary and John sets anyway?

  – [Mary] and [John] bought a book
  – Set intersection = ∅
  – how about “and = set union” then?
Truth Conditions and Values

• Example:
  – The square is bigger than the circle
  – The circle is smaller than the square

  – Are they synonymous?
  – Are they contradictory?
  – Is there an entailment relationship?
  – Are they tautologies?
More examples

• 1. Does sleep entail snore?
   A. He is sleeping entails He is snoring
   B. He is snoring entails He is sleeping
• 2. Does snore presuppose sleep?
• 3. What does “when did you stop beating your wife?” presuppose?
• 3. Given the statement “All crows are black”,
   give an example of a sentence expressing a tautology involving this statement?
   – Stmt or negation Stmt
Propositional Logic

• Recall the distinction between truth conditions and truth values ...

• Possible world or situation:
  – we can create a possible world in Prolog by asserting (positive) facts into its database
  – Prolog use the closed world assumption
    • i.e. things not explicity stated to be true are assumed to be false
Propositional Logic

Cheat sheet

• Starting SWI Prolog from Terminal/Shell:
  – swipl (if in PATH)
  – /opt/local/bin/swipl (default install location on my mac)

^D (control-D) or halt. to quit
Propositional Logic

Cheat sheet

• Viewing the database:
  – listing.

• Assert (and delete) facts at the command line directly using predicates:
  – assert(\textit{fact}).
  – retract(\textit{fact}).

• Put facts into a file and load file
  – \texttt{[filename]}. (assumed to have extension .pl)
  – (or via pull-down menu in Windows)

• Propositions:
  – named beginning with a lower case letter (not number, not starting with capital letter or underscore: variable – no variables in propositional logic), examples:
    – assert(p). (makes p true in this situation)
    – p. (asks Prolog if p true in this situation)
    – dynamic q. (registers proposition q, prevents error message)
Propositional Logic

• Example:

Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 5.10.4)
Copyright (c) 1990-2011 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- p.
ERROR: toplevel: Undefined procedure: p/0 (DWIM could not correct goal)
?- dynamic q.
true.

?- q.
false.

?- assert(p).
true.

?- p.
true.

?- listing.
:- dynamic q/0.

:- dynamic p/0.
p.
true.

?-
Propositional Logic

• Propositions can be combined using logical connectives and operators
  – Conjunction \( p \land q \).
  – Disjunction \( p \lor q \).
  – Negation \( \neg p \).

• Not directly implemented in Prolog
  – Implication \( p \rightarrow q \). (IS NOT THIS!!!)

Use parentheses ( ) to restrict/clarify scope

can’t add \( p, q \). to the database

needs both \( p \) and \( q \) to be true, see next slide
Propositional Logic

- Help:
  - `?- help(->).`
  - true.

It uses the X11 Window system, which may or may not exist on your system.

The construct `fail` is used when the condition fails. This unusual semantics is part of the ISO and all de-facto Prolog standards.
Propositional Logic

• Also not implemented
  – Logical equality $p = q$.

```prolog
?- p = q.  % false.
false.

?- p = p.  % true.
true.

?- q = q.  % true.
true.
```

Propositional Logic

• Help:

```
1 class
C  = (left, right)

The classes themselves
Identity between two expressions (+)
```

Not quite the right documentation page
= is unifiability in Prolog
Propositional Logic

• Prolog exercise:
  – evaluate formula below for different truth values of A and B

example, using the propositional variables A and B, the binary connectives ∨ and ∧ representing disjunction and conjunction, respectively, and the unary connective ¬ representing negation, the following formula can be obtained:

\[(A \land B) \lor (\neg A) \lor (\neg B)\].

From wikipedia

?- dynamic a.
true.

?- dynamic b.
true.

?- (a, b) ; \+a ; \+ b.
Propositional Logic

• How to demonstrate a propositional formula is a tautology?

• **One answer:** exhaustively enumerate a truth table

http://en.wikipedia.org/wiki/Truth_table
Propositional Logic

• Example:

\[(A \land B) \lor (\neg A) \lor (\neg B)\]

<table>
<thead>
<tr>
<th>(A, B)</th>
<th>(\lor A)</th>
<th>(\lor B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T, T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T, F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F, T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F, F</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Table has \(2^n\) rows, where \(n\) is the number of propositional elements

complexity: exponential in \(n\)
Propositional Logic

- Other connectives (are non-primitive)

### Logical Implication

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$p \rightarrow q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

### Logical Equivalence: $(p \rightarrow q) = (\neg p \lor q)$

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$\neg p$</th>
<th>$\neg p \lor q$</th>
<th>$p \rightarrow q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Propositional Logic

• Other connectives (are non-primitive)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>( p = q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( T )</td>
<td>( F )</td>
<td>( F )</td>
</tr>
<tr>
<td>( F )</td>
<td>( T )</td>
<td>( F )</td>
</tr>
<tr>
<td>( F )</td>
<td>( F )</td>
<td>( T )</td>
</tr>
</tbody>
</table>

aka
\[ p \leftrightarrow q \]

• From 1\(^{st}\) and 4\(^{th}\) line of truth table, we can easily deduce how to simulate \( p \leftrightarrow q \) in Prolog using , ; and \(+\)
Propositional Logic

Let’s prove the law of contraposition

There are infinitely many tautologies. Examples include:

- \((A \lor \neg A)\) ("A or not A"), the law of the excluded middle. This formula has only one propositional variable, \(A\). Any valuation for this formula must, by definition, assign \(A\) one of the truth values true or false, and assign \(\neg A\) the other truth value.
- \((A \rightarrow B) \iff (\neg B \rightarrow \neg A)\) ("if \(A\) implies \(B\) then not-\(B\) implies not-\(A\)", and vice versa), which expresses the law of contraposition.
- \(((\neg A \rightarrow B) \land (\neg A \rightarrow \neg B)) \rightarrow A\) ("if not-\(A\) implies both \(B\) and its negation not-\(B\), then not-\(A\) must be false, then \(A\) must be true"), which is the principle known as reductio ad absurdum.
- \((\neg (A \land B)) \iff (\neg A \lor \neg B)\) ("if not both \(A\) and \(B\), then either not-\(A\) or not-\(B\)", and vice versa), which is known as de Morgan's law.
- \(((A \rightarrow B) \land (B \rightarrow C)) \rightarrow (A \rightarrow C)\) ("if \(A\) implies \(B\) and \(B\) implies \(C\), then \(A\) implies \(C\)"), which is the principle known as syllogism.
- \(((A \lor B) \land (A \rightarrow C) \land (B \rightarrow C)) \rightarrow C\) (if at least one of \(A\) or \(B\) is true, and each implies \(C\), then \(C\) must be true as well), which is the principle known as proof by cases.

A minimal tautology is a tautology that is not the instance of a shorter tautology.

- \((A \lor B) \rightarrow (A \lor B)\) is a tautology, but not a minimal one, because it is an instantiation of \(C \rightarrow C\).

http://en.wikipedia.org/wiki/Tautology_(logic)
Propositional Logic

• Prove both sides of De Morgan’s Laws:

\[
\neg(P \lor Q) \iff (\neg P) \land (\neg Q)
\]

\[
\neg(P \land Q) \iff (\neg P) \lor (\neg Q)
\]

**Note**: De Morgan’s laws tell us we can do without one of conjunction or disjunction. Why?
Propositional Logic

- It’s easy to write a short program in Prolog to automate all this ...

```prolog
try(Vars,Formula) :-
  toggleL(Vars,Xs), eval(Formula,Xs),
  fail.
try(_,_).

% toggles list of propositional variables as true and false
% tries every combination
toggleL([],[]).
toggleL([V|Vs],[X|Xs]) :- toggle(V,X), toggleL(Vs,Xs).
toggle(V,X) :- assert(V), X =\neg V ; retract(V), X = \neg V.
eval(F,Xs) :- call(F) -> format('~w true~n', [Xs]); format('~w false~n', [Xs]).
```

Program: plogic.pl
Propositional Logic

• Example using try/2:

\[(A \land B) \lor (\neg A) \lor (\neg B)\]

?- [plogic].
% plogic compiled 0.00 sec, 520 bytes
true.

?- try([p,q],((p,q);\+p;\+q)).
[p,q] true
[p,-q] true
[-p,q] true
[-p,-q] true
true.

It's a tautology! true under all possible conditions
Propositional Logic

- We can get a bit fancier, support \( \rightarrow \) and \( \leftrightarrow \)

```prolog
:- op(1200, xfx, '->').

try(Vars,Formula) :-
    convert(Formula,PrologFormula),
    toggleL(Vars,Xs), eval(PrologFormula,Xs),
    fail.
try(_,_).

% handles \( \rightarrow \) (implication) and \( \leftrightarrow \) (equivalence)
convert((\+ X),(\+ A)) :- !, convert(X,A).
convert((X,Y),(A,B)) :- !, convert(X,A), convert(Y,B).
convert((X;Y),(A;B)) :- !, convert(X,A), convert(Y,B).
convert((X->Y),(\+A;B)) :- !, convert(X,A), convert(Y,B).
convert((X<->Y),(\+(A;B);\+(\+A,\+B))) :- !, convert(X,A), convert(Y,B).
convert(X,X).

% toggles list of propositional variables as true and false
toggleL([],[]).
toggleL([V|Vs],[X|Xs]) :- toggle(V,X), toggleL(Vs,Xs).
toggle(V,X) :- assert(V), X =\= V ; retract(V), X = \= V.
eval(F,Xs) :- call(F) -> format('~w true~n',[Xs]) ; format('~w false~n',[Xs]).
```

Program: plogic2.pl
Propositional Logic

• We can get even fancier; eliminate having to supply the propositional variables

Program: plogic3.pl
Truth table enumeration

- Parsing the formula:

```
% handles -> (implication) and <--> (equivalence)

convert([-+ X],[-+ A],Vs1,Vs2) :- !, convert(X,A,Vs1,Vs2).
convert([X,Y],(A,B),Vs1,Vs3) :- !, convert(X,A,Vs1,Vs2), convert(Y,B,Vs2,Vs3).
convert([X;Y],(A;B),Vs1,Vs3) :- !, convert(X,A,Vs1,Vs2), convert(Y,B,Vs2,Vs3).
convert([X->Y],[-+A;B],Vs1,Vs3) :- !, convert(X,A,Vs1,Vs2), convert(Y,B,Vs2,Vs3).
convert([X<-->Y],[A,B;[-+A,+-B]],Vs1,Vs3) :- !, convert(X,A,Vs1,Vs2), convert(Y,B,Vs2,Vs3).
convert(X,X,Vs1,Vs2) :- member(X,Vs1) -> Vs2 = Vs1 ; Vs2 = [X|Vs1].
```

11. \+ X converts to \+ A if (subformula) X converts to A
12. X,Y converts to A,B if X converts to A and Y converts to B
13. X;Y converts to A;B if X converts to A and Y converts to B
14. X->Y converts to \+A;B if X converts to A and Y converts to B
15. X<-->Y converts to (A,B) ; (\+A,\+B) if X converts to A and Y converts to B
16. X converts to X and add X to the list of propositional variables if it isn’t already in the list
Propositional Logic

**Program: plogic3.pl**

```prolog
[plogic3].
% plogic3 compiled 0.00 sec, 120 bytes
true.

\[ (A \rightarrow B) \leftrightarrow (\neg B \rightarrow \neg A) \]

?- try(((a->b)<->(\neg b \rightarrow \neg a))).
[a,b] true
[a,-b] true
[-a,b] true
[-a,-b] true
true.

\[ \neg(A \land B) \leftrightarrow (\neg A \lor \neg B) \]

?- try((\neg((a,b))<-(\neg a ; \neg b))).
[a,b] true
[a,-b] true
[-a,b] true
[-a,-b] true
true.

\[ ((A \rightarrow B) \land (B \rightarrow C)) \rightarrow (A \rightarrow C) \]

?- try(((a->b),(b->c))-(a->c))).
[a,b,c] true
[a,b,-c] true
[a,-b,c] true
[a,-b,-c] true
[-a,b,c] true
[-a,b,-c] true
[-a,-b,c] true
[-a,-b,-c] true
true.

\[ ((A \lor B) \land (A \rightarrow C) \land (B \rightarrow C)) \rightarrow C \]

?- try(((a;b),(a->c),(b->c))-(c))).
[a,b,c] true
[a,b,-c] true
[a,-b,c] true
[a,-b,-c] true
[-a,b,c] true
[-a,b,-c] true
[-a,-b,c] true
[-a,-b,-c] true
true.
```
Semantic Grammars

• Use slides from course
  – LING 324 – Introduction to Semantics
  – Simon Frasier University, Prof. F.J. Pelletier

• *Difference is we’re computational linguists... so we’re going to implement the slides*

• We’ll do the syntax part this lecture, and the semantics next time
Syntax

Syntax of a Fragment of English (F1)

(2) a. \( S \rightarrow N \ VP \)
b. \( S \rightarrow S \ conj \ S \)
c. \( S \rightarrow \neg S \)
d. \( VP \rightarrow V_t \ N \)
e. \( VP \rightarrow V_i \)
f. \( N \rightarrow \text{Jack, Sophia, James} \)
g. \( V_i \rightarrow \text{is boring, is hungry, is cute} \)
h. \( V_t \rightarrow \text{likes} \)
i. \( \text{conj} \rightarrow \text{and, or} \)
j. \( \text{neg} \rightarrow \text{it is not the case that} \)
Syntax

• We already know how to build Prolog grammars
• See
  – http://www.swi-prolog.org/pldoc/doc_for?
    object=section(2,'4.12',swi('/doc/Manual/DCG.html'))
    for the executive summary

4.12 DCG Grammar rules

Grammar rules form a comfortable interface to difference-lists. They are designed both to support writing parsers that build a parse tree from a list of characters or tokens as for generating a flat list from a term.

Grammar rules look like ordinary clauses using -->/2 for separating the head and body rather than :--/2. Expanding grammar rules is done by expand_term/2, which adds two additional argument to each term for representing the difference list.

The body of a grammar rule can contain three types of terms. A callable term is interpreted as a reference to a grammar-rule. Code between { ... } is interpreted as plain Prolog code and finally, a list is interpreted as a sequence of literals. The Prolog control-constructs (\+/1, -->/2, ;/2, ,/2 and !0) can be used in grammar rules.
Syntax

- Class exercise
Syntax

- **Step 1**: let’s build the simplest possible Prolog grammar for this

\[(2)\]

\[a. \quad S \rightarrow N \ VP \] \hspace{1cm} \[b. \quad S \rightarrow S \ conj \ S \] \hspace{1cm} \[c. \quad S \rightarrow neg \ S \]
\[d. \quad VP \rightarrow V_t \ N \] \hspace{1cm} \[e. \quad VP \rightarrow V_i \]
\[f. \quad N \rightarrow \text{Jack, Sophia, James} \] \hspace{1cm} \[g. \quad V_i \rightarrow \text{is boring, is hungry, is cute} \]
\[h. \quad V_t \rightarrow \text{likes} \] \hspace{1cm} \[i. \quad \text{conj \rightarrow and, or} \]
\[j. \quad \text{neg \rightarrow it is not the case that} \]

\[(3)\] \hspace{1cm} \[4)\] \hspace{1cm} \[5)\] \hspace{1cm} \[6)\] \hspace{1cm} \[7)\]

\[\text{Jack is hungry.} \] \hspace{1cm} \[\text{Sophia likes James.} \] \hspace{1cm} \[\text{It is not the case that James is cute.} \]
\[\text{It is not the case that James is cute.} \] \hspace{1cm} \[\text{Jack is hungry, and it is not the case that James likes Jack.} \]
\[\text{It is not the case that Jack is hungry or Sophia is boring.} \]

\[fjpslides4.pdf\]
Slide 4
Simplest possible grammar

g1.pl

1 s --> n, vp.
2 s --> neg, s.
3 vp --> vt, n.
4 vp --> vi.
5 n --> [jack].
6 n --> [sophia].
7 n --> [james].
8 vi --> [is,boring].
9 vi --> [is,hungry].
10 vi --> [is,cute].
11 vt --> [likes].
12 conj --> [and].
13 conj --> [or].
14 neg --> [it,is,not,the,case,that].

Excluding (2b) for the time being
Simplest possible grammar

?- [g1].
% g1 compiled 0.00 sec, 5,720 bytes
true.

?- s([jack,is,hungry],[]).
true ;
false.

?- s([sophia,likes,james],[]).
true ;
false.

?- s([it,is,not,the,case,that,james,is,cute],[]).
true ;
false.

?-
Syntax

• **Step 2:** let’s add the parse tree component to our grammar ...

\[(2)\]

a. \( S \rightarrow N \ VP \)

b. \( S \rightarrow S \ conj \ S \)

c. \( S \rightarrow neg \ S \)

d. \( VP \rightarrow V_i \ N \)

e. \( VP \rightarrow V_i \)

f. \( N \rightarrow \text{Jack, Sophia, James} \)

g. \( V_i \rightarrow \text{is boring, is hungry, is cute} \)

h. \( V_t \rightarrow \text{likes} \)

i. \( \text{conj} \rightarrow \text{and, or} \)

j. \( \text{neg} \rightarrow \text{it is not the case that} \)

Recall: grammar rules can have extra arguments

(1) Parse tree
(2) Implement agreement etc.
Syntax

Note: on handling left recursion in Prolog grammar rules

• techniques:
  1. use a bottom-up parser
  2. rewrite grammar  
     (left recursive -> right recursive)
  3. or use lookahead  
     (today’s lecture)

b. \[ S \rightarrow S \text{conj} S \]
\[ s(s(S_1,C,S_2)) \rightarrow \text{lookahead}, s(S_1), \text{conj}(C), s(S_2). \]

lookahead is a dummy nonterminal that does not contribute to the parse, it is a “guard” that prevents rule from firing unless appropriate

lookahead succeeds if it can find a conjunction in the input and marks it (so it can’t find it twice)
Grammar: version 2

g2.pl

1 s(s(N,VP)) --> n(N), vp(VP).
2 s(s(S1,Conj,S2)) --> lookahead, s(S1), conj(Conj), s(S2).
3 s(s(Neg,S)) --> neg(Neg), s(S).
4 vp(vp(VT,N)) --> vt(VT), n(N).
5 vp(vp(VI)) --> vi(VI).
6 n(n(jack)) --> [jack].
7 n(n(sophia)) --> [sophia].
8 n(n(james)) --> [james].
9 vi(v(v(is),ap(boring))) --> [is,boring].
10 vi(v(v(is),ap(hungry))) --> [is,hungry].
11 vi(v(v(is),ap(cute))) --> [is,cute].
12 vt(v(likes)) --> [likes].
13 conj(conj(and)) --> [and1].
14 conj(conj(or)) --> [or1].
15 neg(neg(not)) --> [it,is,not,the,case,that].
Grammar: version 2

16
17 % lookahead/2 succeeds when it can find some conjunct in the input
18 % it marks the conjunct and returns the modified list
19 lookahead(L1,L2) :-
20     append(Left,[Conj\Right],L1),
21     conj(Conj,MarkedConj),
22     !, % commit
23     append(Left,[MarkedConj\Right],L2).
24
25 conj(and,and1).
26 conj(or,or1).
Grammar: version 2

?- [g2].
Warning: /Users/sandiway/Desktop/g2.pl:25:
  Redefined static procedure conj/2
% g2 compiled 0.00 sec, 8,024 bytes
true.

?- s(X,[jack,is,hungry],[]).
X = s(n(jack), vp(v(v(is), ap(hungry)))) ;
false.

?- s(X,[sophia,likes,james],[]).
X = s(n(sophia), vp(v(likes), n(james))) ;
false.

?- s(X,[it,is,not,the,case,that,james,is,cute],[]).
X = s(neg(not), s(n(james), vp(v(v(is), ap(cute)))))) ;
false.
Grammar: version 2

Examples (6) and (7) from slide 9

?- s(X,[jack,is,hungry,and,it,is,not,the,case,that,james,likes,jack],[]).
X = s(s(n(jack), vp(v(is), ap(hungry)))) , conj(and) , s(neg(not) , s(n(james), vp(v(likes) , n(jack))))) ; false.

?- s(X,[it,is,not,the,case,that,jack,is,hungry,or,sophia,is,boring],[]).
X = s(s(neg(not) , s(n(jack), vp(v(is), ap(hungry))))) , conj(or) , s(n(sophia), vp(v(is) , ap(boring)))) ;
X = s(neg(not) , s(s(n(jack), vp(v(is), ap(hungry))))) , conj(or) , s(n(sophia), vp(v(is) , ap(boring)))) ;
false.

?-
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


- SWI Prolog built-in:

  \[
  \text{setof(+Template, +Goal, -Set)}
  \]

  Equivalent to \text{bagof/3}, but sorts the result using \text{sort/2} to get a sorted list of alternatives without duplicates.

  \[
  \text{bagof(+Template, :Goal, -Bag)}
  \]

  Unify \text{Bag} with the alternatives of \text{Template}, if \text{Goal} has free variables besides the one sharing with \text{Template} \text{bagof} will backtrack over the alternatives of these free variables, unifying \text{Bag} with the corresponding alternatives of \text{Template}. The construct \text{+Var^Goal} tells \text{bagof} not to bind \text{Var} in \text{Goal}. \text{bagof/3} fails if \text{Goal} has no solutions.

### 4.29 Finding all Solutions to a Goal

\[
\text{findall(+Template, :Goal, -Bag)}
\]

Creates a list of the instantiations \text{Template} gets successively on backtracking over \text{Goal} and unifies the result with \text{Bag}. Succeeds with an empty list if \text{Goal} has no solutions. \text{findall/3} is equivalent to \text{bagof/3} with all free variables bound with the existential operator (\^), except that \text{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).
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)).
true.
?- setof(X, cute(X), Set), member(Y, Set).
Set = [james, sophia],
Y = james ;
Set = [james, sophia],
Y = sophia.

?- listing(cute).
:- dynamic cute/1.
cute(james).
cute(sophia).
```
(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. $[[\text{it is not the case}]]^V = \begin{bmatrix} 1 \to 0 \\ 0 \to 1 \end{bmatrix}$

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

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

Interpretive Rules for Each Syntactic Rule

- \([A B C] \) is equivalent to

\[ \begin{array}{c}
  A \\
  \hline
  B \quad C
\end{array} \]

- \( [[A B C]] \) stands for the semantic value of

\[ \begin{array}{c}
  A \\
  \hline
  B \quad C
\end{array} \]

- 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. \( [[VP \ Vt \ N]]^V = \{x: \langle x, [[N]]^V \rangle \in [[Vt]]^V \} \)

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.

Semantics

\[ \text{S} \]
\[ \text{N} \quad \text{VP} \]
\[ \text{Jack} \quad \text{V}_i \]
\[ \text{is hungry} \]

\[ 1 \text{ iff } \text{Jack}' \in \{ x : x \text{ is hungry in V} \} \]

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

- Desired implementation:

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

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

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

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

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

*Imagine if the Penn Treebank was labeled using a semantic representation*
Semantics: Implementation

• 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. $n(n(jack)) \rightarrow \text{[jack]}$.
4. $vi(v(v(is),ap(hungry)))) \rightarrow \text{[is,hungry]}$.

$$\text{setof}(X,\text{hungry}(X),S), \ \text{member}(jack,S)$$

1. iff $\text{Jack'} \in \{x: x \text{ is hungry in } V\}$

```
jack
  Jack'
    {x: x is hungry in V}
  Jack'
    {x: x is hungry in V}
  {x: x is hungry in V}
```

```
setof(X, hungry(X), S)
```
Semantics: Implementation

- Semantic grammar:

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

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

• Semantic grammar:

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

?- \text{dynamic hungry/1.}
true.

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

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

?- s(X,[jack,is,hungry],[]), call(X).
X = (\text{setof}(_G14, \text{hungry}(_G14), [jack]), \text{member}(jack, [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.

?- s(X,[sophia,likes,james],[]).
X = (setof(_G315, likes(_G315, james), _G0307), member(sophia, _G0307)) ;

1 s((Fn,member(N,X))) --> n(N), vp(lambda(X,Fn)).
2 vp(VI) --> vi(VI).
3 vp(Fn) --> vt(lambda(X,Fn)), n(X), {X=Y}.
4 n(james) --> [james].
5 n(sophie) --> [sophie].
6 vi(lambda(S,setof(X,hungry(X),S))) --> [is,hungry].
7 vt(lambda(Y,lambda(S,setof(X,likes(X,Y),S)))) --> [likes].
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].

\[
\begin{align*}
[S4]^V & = [Neg]^V(([S3]^V) = 1 \text{ iff } J \not\in \{x: x \text{ is hungry in } \mathcal{V}\} \\
& \quad \text{ and } S' \not\in \{x: x \text{ is boring in } \mathcal{V}\}\end{align*}
\]

\[
[\text{it is not the case that}]^V =
\begin{cases}
1 & \text{if } J \not\in \{x: x \text{ is hungry in } \mathcal{V}\} \\
0 & \text{otherwise}
\end{cases}
\]

\[
[S3]^V = [\text{Conj}]^V(<[S2]^V,[S1]^V>) = 0 \text{ iff } J \not\in \{x: x \text{ is hungry in } \mathcal{V}\} \\
& \quad \text{ and } S' \not\in \{x: x \text{ is boring in } \mathcal{V}\}
\]

\[
[S2]^V = 1 \text{ iff } [N2]^V \in [V \text{ or } 2]^V = 1 \text{ iff } J' \in \{x: x \text{ is hungry in } \mathcal{V}\}
\]

\[
[V2]^V =\begin{cases}
1 & \text{if } \text{Jack} \not\in \{x: x \text{ is hungry in } \mathcal{V}\} \\
0 & \text{otherwise}
\end{cases}
\]

\[
[J\text{ack}]^V = J'
\]

\[
[S1]^V = 1 \text{ iff } [N1]^V \in [V \text{ or } 1]^V = 1 \text{ iff } S' \in \{x: x \text{ is boring in } \mathcal{V}\}
\]

\[
[N1]^V = S' 
\]

\[
[S0]^V = [\text{is hungry}]^V = \begin{cases}
1 & \text{if } J' \not\in \{x: x \text{ is hungry in } \mathcal{V}\} \\
0 & \text{otherwise}
\end{cases}
\]

\[
[\text{is hunger}]^V = \begin{cases}
1 & \text{if } J' \not\in \{x: x \text{ is hungry in } \mathcal{V}\} \\
0 & \text{otherwise}
\end{cases}
\]

\[
[S1]^V = [\text{is boring}]^V = \begin{cases}
1 & \text{if } S' \not\in \{x: x \text{ is boring in } \mathcal{V}\} \\
0 & \text{otherwise}
\end{cases}
\]

\[
[S1]^V = [\text{is boring}]^V = \begin{cases}
1 & \text{if } S' \not\in \{x: x \text{ is boring in } \mathcal{V}\} \\
0 & \text{otherwise}
\end{cases}
\]
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: version 3

g3.pl

1 s((Formula, member(E, Set))) -> n(E), vp(lambda(Set, Formula)).
2 s(S3) -> lookahead, s(S1), conj(S1, S2, S3), s(S2).
3 s(NegS) -> neg(S, NegS), s(S).
4 vp(Formula) -> vt(lambda(X, Formula)), n(Y), \{X\=Y\}.
5 vp(Formula) -> vi(Formula).
6 n(jack) -> [jack].
7 n(sophia) -> [sophia].
8 n(james) -> [james].
9 vi(lambda(Set, setof(X, boring(X), Set))) -> [is, boring].
10 vi(lambda(Set, setof(X, hungry(X), Set))) -> [is, hungry].
11 vi(lambda(Set, setof(X, cute(X), Set))) -> [is, cute].
12 vt(lambda(Y, lambda(Set, setof(X, likes(X, Y), Set)))) -> [likes].
13 conj(S1, S2, (S1, S2)) -> [and1].
14 conj(S1, S2, (S1; S2)) -> [or1].
15 neg(P, (\+ P)) -> [it, is, not, the, case, that].
Grammar: version 3

16
17 % lookahead/2 succeeds when it can find some conjunct in the input
18 % it marks the conjunct and returns the modified list
19 lookahead(L1,L2) :-
20     append(Left,[Conj\Right],L1),
21     conj(Conj,MarkedConj),
22     !,                             % commit
23     append(Left,[MarkedConj\Right],L2).
24
25 conj(and,and1).
26 conj(or,or1).
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.