LING 581: Advanced Computational Linguistics

Lecture Notes
April 23rd
Last Time

• Built a Prolog grammar
  – g.pl on the webpage

• TCE:
  – *We'll do this first* ...
Truth Conditions and Values

• Set-theoretic meaning of "and":
  – The circle is inside the square and the circle is dark
  – Mary is a student and a baseball fan
  – Mary and John bought a book
Truth Conditions and Values

• Terms:
  – synonymous, contradictory, entailment, presupposition, tautology...

• Prolog:
  – Conjunction \( p \land q \).
  – Disjunction \( p \lor q \).
  – Negation \( \neg p \).
  – If-Then \( p \implies q \). (not implication)
  – If-Then-Else \( p \implies q \lor r \).
  – Unification \( p = q \). (not logical equality)
Tautologies

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 \(C \rightarrow C\).
Propositional Logic

Program: plogic3.pl

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

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

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

?- 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

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

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

(3) Jack is hungry.

(4) Sophia likes James.

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

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

(7) It is not the case that Jack is hungry or Sophia is boring.
Syntax

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

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

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)

```prolog
b. S → S conj S

1  s(s(S1,C,S2)) --> lookahead, s(S1), conj(C), s(S2).
```

`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

\[\begin{align*}
1. & \quad s(s(N,VP)) \rightarrow n(N),\ vp(VP). \\
2. & \quad s(s(S1,CONJ,S2)) \rightarrow \text{lookahead, } s(S1),\ \text{conj(CONJ), } s(S2). \\
3. & \quad s(s(NEG,S)) \rightarrow \text{neg(NEG), } s(S). \\
4. & \quad vp(vp(VT,N)) \rightarrow \text{vt(VT), } n(N). \\
5. & \quad vp(vp(VI)) \rightarrow \text{vi(VI).} \\
6. & \quad vt(vt(likes)) \rightarrow [\text{likes}]. \\
7. & \quad vi(vi(is\_boring)) \rightarrow [\text{is,boring}]. \\
8. & \quad vi(vi(is\_hungry)) \rightarrow [\text{is,hungry}]. \\
9. & \quad vi(vi(is\_cute)) \rightarrow [\text{is,cute}]. \\
10. & \quad n(n(james)) \rightarrow [\text{james}]. \\
11. & \quad n(n(sophia)) \rightarrow [\text{sophia}]. \\
12. & \quad n(n(jack)) \rightarrow [\text{jack}]. \\
13. & \quad conj(conj(and)) \rightarrow [\text{and1}]. \\
14. & \quad conj(conj(or)) \rightarrow [\text{or1}]. \\
15. & \quad neg(neg) \rightarrow [\text{it,is,not,the,case,that}].
\end{align*}\]
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
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:

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

bagof(+Template, :Goal, -Bag)
Unify Bag with the alternatives of Template, if Goal has free variables besides the one sharing with Template bagof will backtrack 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

```
findall(+Template, :Goal, -Bag)
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.
   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:

```prolog
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).
```
Semantics

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

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

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

Interpretive Rules for Each Syntactic Rule

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

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

- 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. \(\llbracket [S \ N \ VP] \rrbracket^V = 1\) iff \(\llbracket N \rrbracket^V \in \llbracket VP \rrbracket^V\) and \(0\) otherwise.
b. \(\llbracket [S ; S1 \ conj \ S2] \rrbracket^V = \llbracket \text{conj} \rrbracket^V(\llbracket S1 \rrbracket^V, \llbracket S2 \rrbracket^V)\)
c. \(\llbracket [S \ neg \ S] \rrbracket^V = \llbracket \text{neg} \rrbracket^V(\llbracket S \rrbracket^V)\)
d. \(\llbracket [VP \ V_t \ N] \rrbracket^V = \{x: \langle x, \llbracket N \rrbracket^V \rangle \in \llbracket V_t \rrbracket^V\}\)
e. If \(A\) is a category and \(a\) is a lexical entry or a lexical category and \(\Delta = [A \ a]\), then \(\llbracket \Delta \rrbracket^V = \llbracket a \rrbracket^V\)
Semantics

(11) Jack is hungry.

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

1 iff \( \text{Jack}' \in \{ x : x \text{ is hungry in } V \} \)

\[
\begin{align*}
  \text{Jack}' & \quad \{ x : x \text{ is hungry in } V \} \\
  \text{Jack}' & \quad \{ x : x \text{ is hungry in } V \} \\
  \{ x' : x' \text{ is hungry in } V \}
\end{align*}
\]
Semantics: Implementation

• Desired implementation:

```
?- 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.
```

The extra argument returns a Prolog query that can be evaluated against the database.

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

- Semantic grammar:

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

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

- Semantic grammar:
  
  $s((F_n, member(N, X))) \rightarrow n(N), \ vp(lambda(X, F_n))$.
  $vp(VI) \rightarrow vi(VI)$.
  $n(jack) \rightarrow [jack]$.
  $vi(lambda(S, setof(X, hungry(X), S))) \rightarrow [is, hungry]$.

?- dynamic hungry/1.
true.

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

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

?- s(X, [jack, is, hungry], []), call(X).
X = (setof(_G14, hungry(_G14), [jack]), 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), _G307), member(sophia, _G307));

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(Y), {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.
s((Fn,member(N,X))) -> n(N), vp(lambda(X,Fn)).
s(NS) --> neg(S,NS), s(S).
vp(VI) --> vi(VI).
vp(Fn) --> vt(lambda(X,Fn)), n(Y), \{X=Y\}.
n(james) --> [james].
n(sophie) --> [sophie].
vi(lambda(S,setof(X,hungry(X),S))) --> [is,hungry].
vi(lambda(S,setof(X,cute(X),S))) --> [is,cute].
vt(lambda(Y,lambda(S,setof(X,likes(X,Y),S)))) --> [likes].
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].

\[
\begin{align*}
\llbracket S4 \rrbracket^V &= [\llbracket \text{Neg} \rrbracket^V (\llbracket S3 \rrbracket^V)] = \\
&= 1 \text{ iff } J' \notin \{x: x \text{ is hungry in } V\} \\
&\quad \text{ and } S' \notin \{x: x \text{ is boring in } V\} \\
\llbracket \text{Neg} \rrbracket^V &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
\llbracket S3 \rrbracket^V &= [\llbracket \text{Conj} \rrbracket^V (< \llbracket S2 \rrbracket^V, \llbracket S1 \rrbracket^V >)] = \\
&= 0 \text{ iff } J' \notin \{x: x \text{ is hungry in } V\} \\
&\quad \text{ and } S' \notin \{x: x \text{ is boring in } V\} \\
\llbracket \text{Conj} \rrbracket^V &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\end{align*}
\]

fjpSlides4.pdf
Slide 11
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.