Computational Semantics

CS224N 2003
Christopher Manning

(Borrows slides from Mary Dalrymple, Jason Eisner (extensively), and Jim Martin)

Why study computational semantics?

- Because everyone has been wanting me to talk about this all course!?
- Obvious applications
  - Summarization
  - Translation
  - Question answering
  - Information access
  - Talking to your pet robot
  - Speech user interfaces
- The next generation of intelligent applications need deeper semantics

Shallow vs. deep semantics

- We want to get from syntax to meaning!
- We can do more than we would have thought without deep linguistic analysis
- But we can't do everything we would like:
  - Not all tasks can ignore higher structure
  - Unsuitable if new text must be generated
  - Unsuitable if aim is not just to hand the user part of a document, relying on the author of the document and the user to make sense of the result

An early example: Chat-80

- Developed between 1979 and 1982 by Fernando Pereira and David Warren; became Pereira's dissertation
- Proof-of-concept natural language interface to database system
- Used in projects: e.g. Shoptalk (Cohen et al. 1989), a natural language and graphical interface for decision support in manufacturing
- Even in an ANLP-2000 conference paper!
- Available in src directory
  - Need sicstus prolog: /afs/ir/class/symbsys139p/bin/sicstus

The CHAT-80 Database

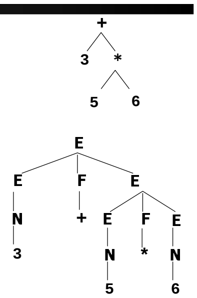
```
% Facts about countries.
% country(Country,Region,Latitude,Longitude,
%     Area (sqmiles), Population, Capital,Currency)
country(andorra,southern_europe,42,-1,179,
   25000,andorra_la_villa,franc_peseta).
country(angola,southern_africa,-12,-18,481351,
   5810000,luanda,?).
country(argentina,south_america,-35,66, 1072067,
   23920000,buenos_aires,peso).

capital(C,Cap) :- country(C,_,_,_,_,_,Cap,_).
```

Chat-80 trace (illegibly small)

```
Question: What is the capital of Australia?
Parse: 0.0sec.
whq
  $VAR
    1
  s
    np
      3+sin
      wh(B)
      []
    verb(be,active,pres+fin,[],pos)
    arg
      dir
      np
        3+sin
        np_head
          det(the(sin))
          []
          capital
        pp
          prep(of)
          np
            3+sin
            name(australia)
            []
    []
Semantics: 0.0sec.
answer([B]) :-
  capital(australia,B)
canberra.
```

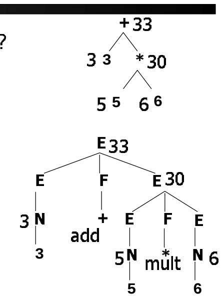
Getting semantics: programming language interpreter

- What is meaning of 3+5*6?
- First parse it into 3+(5*6)



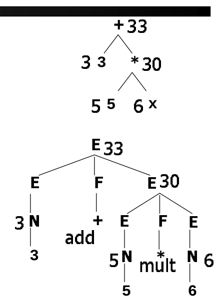
Programming Language Interpreter

- What is meaning of 3+5*6?
- First parse it into 3+(5*6)
- Now give a meaning to each node in the tree (bottom-up)



Interpreting in an Environment

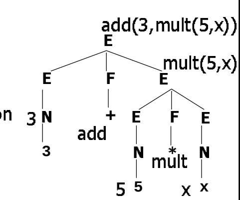
- How about 3+5*x?
- Same thing: the meaning of x is found from the environment (it's 6)
- Analogies in language?



Compiling

- How about 3+5*x?
- Don't know x at compile time
- "Meaning" at a node is a piece of code, not a number

5*(x+1)-2 is a different expression that produces *equivalent* code
(can be converted to the previous code by optimization)
Analogies in language?



What Counts as Understanding? some notions

- We understand if we can respond appropriately
  - ok for commands, questions (these demand response)
  - "Computer, warp speed 5"
  - "throw axe at dwarf"
  - "put all of my blocks in the red box"
  - imperative programming languages
  - database queries and other questions
- We understand statement if we can determine its truth
  - ok, but if you knew whether it was true, why did anyone bother telling it to you?
  - comparable notion for understanding NP is to compute what the NP refers to, which might be useful

What Counts as Understanding? some notions

- We understand statement if we know *how* to determine its truth
  - What are exact conditions under which it would be true?
    - necessary + sufficient
  - Equivalently, derive all its consequences
    - what else must be true if we accept the statement?
  - Philosophers tend to use this definition
- We understand statement if we can use it to answer questions [very similar to above – requires reasoning]
  - **Easy:** John ate pizza. What was eaten by John?
  - **Hard:** White's first move is P-Q4. Can Black checkmate?
  - Constructing a *procedure* to get the answer is enough

What Counts as Understanding? some notions

- Be able to translate
  - Depends on target language
  - English to English? bah humbug!
  - English to French? reasonable
  - English to Chinese? requires deeper understanding
  - English to **logic**? deepest - the definition we'll use!
    - all humans are mortal = $\forall x\ [\text{human}(x) \Rightarrow \text{mortal}(x)]$
- Assume we have logic-manipulating rules to tell us how to act, draw conclusions, answer questions ...

Lecture Plan

- Today:
  - Lookook at some sentences and phrases
  - What would be reasonable logical representations for them?
  - Get some idea of compositional semantics
- Wednesday:
  - How can we build those representations?
- Another course (somewhere in AI, hopefully):
  - How can we reason with those representations?

Logic: Some Preliminaries

Three major kinds of objects

1. Booleans
   - Roughly, the semantic values of sentences
2. Entities
   - Values of NPs, i.e., objects
   - Maybe also other types of entities, like times
3. Functions of various types
   - A function returning a boolean is called a "predicate" – e.g., frog(x), green(x)
   - Functions might return other functions!
   - Function might take other functions as arguments!

Logic: Lambda Terms

- Lambda terms:
  - A way of writing "anonymous functions"
    - No function header or function name
    - But defines the key thing: **behavior** of the function
    - Just as we can talk about 3 without naming it "x"
  - Let square = $\lambda p\ p*p$
  - Equivalent to int square(p) { return p*p; }
  - But we can talk about $\lambda p\ p*p$ without naming it
  - Format of a lambda term: λ variable expression

Logic: Lambda Terms

- Lambda terms:
  - Let square = λp p*p
  - Then square(3) = (λp p*p)(3) = 3*3
  - Note: square(x) isn't a function! It's just the value x*x.
  - But **λx** square(x) = λx x*x = λp p*p = square
    (proving that these functions are equal – and indeed they are, as they act the same on all arguments: what is (λx square(x))(y)?)

  - Let even = λp (p mod 2 == 0) a <u>predicate:</u> returns true/false
  - even(x) is true if x is even
  - How about even(square(x))?
  - λx even(square(x)) is true of numbers with even squares
    - Just apply rules to get λx (even(x*x)) = λx (x*x mod 2 == 0)
    - This happens to denote the same predicate as even does

Logic: Multiple Arguments

- All lambda terms have one argument
- But we can fake multiple arguments ...
- Suppose we want to write times(5,6)
- Remember: square can be written as λx square(x)
- Similarly, times is equivalent to λx λy times(x,y)
- Claim that times(5)(6) means same as times(5,6)
  - times(5) = (λx λy times(x,y)) (5) = λy times(5,y)
    - If this function weren't anonymous, what would we call it?
  - times(5)(6) = (λy times(5,y))(6) = times(5,6)

Logic: Multiple Arguments

- All lambda terms have one argument
- But we can fake multiple arguments ...
- Claim that times(5)(6) means same as times(5,6)
  - times(5) = (λx λy times(x,y)) (5) = λy times(5,y)
    - If this function weren't anonymous, what would we call it?
  - times(5)(6) = (λy times(5,y))(6) = times(5,6)
- So we can always get away with 1-arg functions ...
  - ... which might return a function to take the next argument. Whoa.
  - We'll still allow times(x,y) as syntactic sugar, though

Grounding out

- So what does times actually mean???
- How do we get from times(5,6) to 30 ?
  - Whether times(5,6) = 30 depends on whether symbol times actually denotes the multiplication function!
- Well, maybe times was defined as another lambda term, so substitute to get times(5,6) = (blah blah blah)(5)(6)
- But we can't keep doing substitutions forever!
  - Eventually we have to ground out in a primitive term
  - Primitive terms are bound to object code
- Maybe times(5,6) just executes a multiplication function
- What is executed by loves(john, mary) ?

Logic: Interesting Constants

- Thus, have "constants" that name some of the entities and functions (e.g., times):
  - GeorgeWBush - an entity
  - red – a predicate on entities
    - holds of just the red entities: red(x) is true if x is red!
  - loves – a predicate on 2 entities
    - loves(GeorgeWBush, LauraBush)
    - *Question:* What does loves(LauraBush) denote?
- Constants used to define meanings of words
- Meanings of phrases will be built from the constants

Logic: Interesting Constants

- most – a predicate on 2 predicates on entities
  - most(pig, big) = "most pigs are big"
    - Equivalently, most(λx pig(x), λx big(x))
  - returns true if most of the things satisfying the first predicate also satisfy the second predicate
- similarly for other quantifiers
  - all(pig,big) (equivalent to ∀x pig(x) ⇒ big(x))
  - exists(pig,big) (equivalent to ∃x pig(x) AND big(x))
  - can even build complex quantifiers from English phrases:
    - "between 12 and 75"; "a majority of"; "all but the smallest 2"

A reasonable representation?

- `Gilly swallowed a goldfish`
- First attempt: swallowed(Gilly, goldfish)
- Returns true or false. Analogous to
  - prime(17)
  - equal(4,2+2)
  - loves(GeorgeWBush, LauraBush)
  - swallowed(Gilly, Jilly)
- ... or is it analogous?

A reasonable representation?

- `Gilly swallowed a goldfish`
  - First attempt: swallowed(Gilly, goldfish)
- But we're not paying attention to a!
- `goldfish` isn't the name of a unique object the way `Gilly` is
- In particular, don't want
  `Gilly swallowed a goldfish and Milly swallowed a goldfish`
  to translate as
  swallowed(Gilly, goldfish) AND swallowed(Milly, goldfish)
  since probably not the same goldfish ...

Use a Quantifier

- `Gilly swallowed a goldfish`
  - First attempt: swallowed(Gilly, goldfish)
- Better: ∃g goldfish(g) AND swallowed(Gilly, g)
- Or using one of our quantifier predicates:
  - exists(λg goldfish(g), λg swallowed(Gilly,g))
  - Equivalently: exists(goldfish, swallowed(Gilly))
    - "In the set of goldfish there exists one swallowed by Gilly"
- Here goldfish is a predicate on entities
  - This is the same semantic type as red
  - But goldfish is noun and red is adjective .. #@!?

Tense

- `Gilly swallowed a goldfish`
  - Previous attempt: exists(goldfish, λg swallowed(Gilly,g))
- Improve to use tense:
  - Instead of the 2-arg predicate swallowed(Gilly,g) try a 3-arg version swallow(t,Gilly,g) where t is a time
  - Now we can write:
    ∃t past(t) AND exists(goldfish, λg swallow(t,Gilly,g))
  - "There was some time in the past such that a goldfish was among the objects swallowed by Gilly at that time"

(Simplify Notation)

- `Gilly swallowed a goldfish`
  - Previous attempt: exists(goldfish, swallowed(Gilly))
- Improve to use tense:
  - Instead of the 2-arg predicate swallowed(Gilly,g) try a 3-arg version swallow(time,Gilly,g)
  - Now we can write:
    ∃t past(t) AND exists(goldfish, swallow(t,Gilly))
  - "There was some time in the past such that a goldfish was among the objects swallowed by Gilly at that time"

Event Properties

- `Gilly swallowed a goldfish`
  - Previous: ∃t past(t) AND exists(goldfish, swallow(t,Gilly))
- Why stop at time? An event has other properties:
  - `[Gilly] swallowed [a goldfish] [on a dare] [in a telephone booth] [with 30 other freshmen] [after many bottles of vodka had been consumed].`
  - Specifies who what why when ...
- Replace time variable t with an event variable e
  - ∃e past(e), act(e,swallowing), swallower(e,Gilly), exists(goldfish, swallowee(e)), exists(booth, location(e)), ...
    - As with probability notation, a comma represents AND
    - Could define past as λe ∃t before(t,now), ended-at(e,t)

Compositional Semantics

- We've discussed what semantic representations should look like.
- **But how do we get them from sentences???**
- First - parse to get a syntax tree.
- Second - look up the semantics for each word.
- Third - build the semantics for each constituent
  - Work from the bottom up
  - The syntax tree is a "recipe" for how to do it
- Principle of Compositionality
  - The meaning of a whole is derived from the meanings of the parts, via composition rules

A simple grammar of English

(in Definite Clause Grammar, DCG, form – as in Prolog)

sentence --> noun_phrase, verb_phrase.
noun_phrase --> proper_noun.
noun_phrase --> determiner, noun.
verb_phrase --> verb, noun_phrase.

Proper_noun --> [John]	verb --> [ate]
Proper_noun --> [Mary]	verb --> [kissed]
determiner --> [the]	noun --> [cake]
determiner--> [a]	noun --> [lion]

Extending the grammar to check number agreement between subjects and verbs

S --> NP(Num), VP(Num).
NP(Num) --> Proper_noun(Num).
NP(Num) --> det(Num), noun(Num).
VP(Num) --> verb(Num), noun_phrase(_).

Proper_noun(s) --> [Mary].	noun(s) --> [lion].
det(s) --> [the].	noun(p) --> [lions].
det(p) --> [the].	verb(s) --> [eats].
	verb(p) --> [eat].

A simple grammar with semantics

sentence(SMeaning) --> noun_phrase(NPMeaning), verb_phrase(VPMeaning), combine (NPMeaning, VPMeaning, SMeaning).
noun_phrase(NPMeaning) --> name(NPMeaning).
verb_phrase(VPMeaning) --> verb(VPMeaning).

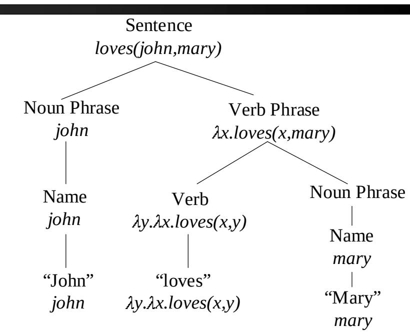
name(john) --> [john].
verb(jumps(X)) --> [jumps].

A simple grammar with semantics 2

sentence(SMeaning) --> noun_phrase(NPMeaning), verb_phrase(VPMeaning), {combine (NPMeaning, VPMeaning, SMeaning)}.
verb_phrase(VPMeaning) --> verb(Vmeaning), noun_phrase(NPMeaning), {combine (NPMeaning, VMeaning, VPMeaning)}.
noun_phrase (NPMeaning) --> name(NPMeaning).

name(john) --> [john].	verb(λx.jumps(x)) --> [jumps]
name(mary) --> [mary].	verb(λy.λx.loves(x,y)) -->[loves]

Parse tree with associated semantics



Augmented CFG Rules

- We can also accomplish this just by attaching semantic formation rules to our syntactic CFG rules

$$A \rightarrow \alpha_1 \ldots \alpha_n \quad \{f(\alpha_1.sem, \ldots \alpha_n.sem)\}$$

- This should be read as the semantics we attach to A can be computed from some function applied to the semantics of A's parts.
- The functions/operations permitted in the semantic rules are restricted, falling into two classes
  - Pass the semantics of a daughter up unchanged to the mother
  - Apply (as a function) the semantics of one of the daughters of a node to the semantics of the other daughters

Example

- S -> NP VP
- VP -> Verb NP
- Verb -> serves

- {VP.sem(NP.sem)}
- {Verb.sem(NP.sem)

$$\lambda x \lambda y \; \exists e \; Serving(e) \wedge Server(e, y) \wedge Served(e, x)$$

- Is it really necessary to specify these attachments?

- No, in each rule there's a daughter whose semantics is a function
- Might just be able to use free type-driven semantic construction