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

What we say to dogs

Okay, Ginger! I've had it! You stay out of the garbage! Understand, Ginger? Stay out of the garbage, or else!

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 sclarus prolog : afs:/clay/umlprobing/impl/usrcathug/sclarus
The CHAT-80 Database

- Facts about countries,
- country(Country, Region, Latitude, Longitude),
- Area (sq miles), 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, Cap).

Chat-80 trace (illegibly small)

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 a dart 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 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)
  - Emy: 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

Lecture Plan

- Today:
  - Look 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. Boolean
   - 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., ∃x human(x) material(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 = λ p p p
  - Equivalent to int square(p) { return p p; }  
- But we can talk about λ p p p without naming it
- Format of a lambda term: λ variable expression
**Logic: Lambda Terms**

- Lambda terms:
  - Let \( \text{square} \equiv \lambda x \, p \, p \)
  - Then \( \text{square}(3) = (\lambda x \, p \, p)(3) = 3^2 \)
  - Note: \( \text{square}(x) \) isn't a function! It's just the value \( x^2 \).
  - But \( \lambda x \, \text{square}(x) = \lambda x \, x^2 = \lambda x \, p \, p \) is \( \text{square} \).
  - (Proving that these functions are equal — and indeed they are, as they act the same on all arguments; what is \( \lambda x \, \text{square}(x)(3) \)?)

- Let \( \text{even} \equiv \lambda p \, (p \mod 2 = 0) \) a predicate; \( \text{even}(x) \) is true if \( x \) is even
  - How about \( \text{even}(\text{square}(x)) \)?
  - \( \lambda x \, \text{even}(\text{square}(x)) \) is true of numbers with even squares
  - Just apply rules to get \( \lambda x \, (\text{even}(\text{square}(x))) = \lambda x \, (p \, p \, p \, p) \)
  - 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 ...

- Claim that \( \text{times}(5)(6) \) means same as \( \text{times}(5,6) \)
  - \( \text{times}(5) = (\lambda x \, y \, \text{times}(x,y))(5) = \lambda y \, \text{times}(5,y) \)
  - If this function weren't anonymous, what would we call it?
  - \( \text{times}(5)(6) = (\lambda x \, \text{times}(x,6))(5) = \text{times}(5,6) \)
  - So we can always get away with 1-arg functions ...
  - ... which might return a function to take the next argument. Why?
  - We'll still allow \( \text{times}(x,y) \) as syntactic sugar, though

---

**Grounding out**

- So what does \( \text{times} \) actually mean???
  - How do we get from \( \text{times}(5,6) \) to 30?
  - Whether \( \text{times}(5,6) = 30 \) depends on whether symbol \( \text{times} \) actually denotes the multiplication function!
  - Well, maybe \( \text{times} \) was defined as another lambda term, so substitute to get \( \text{times}(5,6) = (\text{blah} \, \text{blah} \, \text{blah})(5)(6) \)
  - 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 \( \text{times}(5,6) \) just executes a multiplication function
  - What is executed by \text{loves}(\text{John}, \text{Mary})?

---

**Logic: Interesting Constants**

- Thus, have 'constants' that name some of the entities and functions (e.g., \( \text{times} \)):
  - \( \text{GeorgeBush} \) - an entity
  - \( \text{red} \) - a predicate on entities
    - \( \lambda x \, \text{red}(x) \) is true if \( x \) is red
  - \( \text{loves} \) - a predicate on 2 entities
    - \( \lambda x \, y \, \text{loves}(\text{GeorgeBush}, \text{LauralBush}) \)
  - \( \text{Question: What does loves(\text{GeorgeBush}, \text{LauralBush}) denote?} \)
  - Constants used to define meanings of words
  - Meanings of phrases will be built from the constants

---

**Logic: Interesting Constants**

- \( \text{most} \) - a predicate on 2 predicates on entities
  - \( \text{most}(\text{pig}, \text{big}) \) = "most pigs are big"
    - \( \lambda x \, \text{big}(x) \)
  - \( \text{returns} \) true if most of the things satisfying the first predicate also satisfy the second predicate
  - Similarly for other quantifiers
    - \( \text{all}(\text{pig}, \text{big}) \)
    - \( \text{exists}(\text{pig}, \text{big}) \)
  - Can even build complex quantifiers from English phrases:
    - "between 10 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(GeorgeBush, 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: \( \exists g \) goldfish(h(g)) AND swallowed(Gilly, g)
- Or using one of our quantifier predicates:
  - exists(\( \lambda g \) goldfish(h(g)), \( \lambda g \) swallowed(Gilly,g))
  - Equivalently: \( \exists g \) 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, \( \lambda g \) 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:
  - \( \exists \) past(t) AND exists(goldfish, \( \lambda g \) 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(t, Gilly,g)
- Now we can write:
  - \( \exists \) 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: \( \exists t \) past(t) AND exists(goldfish, swallow(t, Gilly))
- Why stop at time? An event has other properties:
  - \( [\text{Gilly}] \) swallowed \([\text{a goldfish}]\) \([\text{on a dare}]\) \([\text{in a telephone booth}]\) \([\text{with 30 other freshmen}]\) \([\text{after many bottles of vodka were consumed}]\).
  - Specifies who at what when ...
- Replace time variable t with an event variable e
  - \( \text{event}(e), \text{act}(e, \text{swallowing}), \text{swallow}(e, Gilly), \text{exists(goldfish, swallow(e))} \), exists(both, location(e)),
  - As with probability notation, a comma represents AND
  - Could define past as \( \lambda t \) before \( e \), new, event(e) \( \lambda e \)
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

\[ \text{sentence} \rightarrow \text{noun_phrase}, \text{verb_phrase}. \]
\[ \text{noun_phrase} \rightarrow \text{proper_noun}. \]
\[ \text{noun_phrase} \rightarrow \text{determiner}, \text{noun}. \]
\[ \text{verb_phrase} \rightarrow \text{verb}, \text{noun_phrase}. \]

- Proper_noun \rightarrow [John]
- Proper_noun \rightarrow [Mary]
- Determiner \rightarrow [the]
- Noun \rightarrow [cake]
- Noun --&gt; [a]
- Noun --&gt; [lion]

Extending the grammar to check number agreement between subjects and verbs

\[ \text{S} \rightarrow \text{N(Num)}, \text{V(Num)}. \]
\[ \text{N(Num)} \rightarrow \text{Proper_noun(Num)}. \]
\[ \text{N(Num)} \rightarrow \text{det(Num)}, \text{noun(Num)}. \]
\[ \text{V(Num)} \rightarrow \text{verb(Num)}, \text{noun_phrase(\_)}. \]

- Proper_noun(s) \rightarrow [Mary].
- Noun(s) \rightarrow [lion].
- Det(s) \rightarrow [the].
- Noun(p) \rightarrow [lions].
- Det(p) \rightarrow [the].
- Verb(s) \rightarrow [eats].
- Verb(p) \rightarrow [eat].

A simple grammar with semantics

\[ \text{sentence(SMeaning)} \rightarrow \text{noun_phrase(SMeaning)}, \text{verb_phrase(VMeaining)}, \text{combine (SMeaining, VMeaining)}. \]
\[ \text{verb_phrase(VMeaining)} \rightarrow \text{verb(Vmeaining)}, \text{noun_phrase(NMeaining)}, \text{combine (NMeaining, VMeaining)}. \]
\[ \text{noun_phrase(NMeaining)} \rightarrow \text{name(NMeaining)}. \]

- Name(john) \rightarrow [John].
- Verb(jumps(\(x\))) \rightarrow [jumps].
- Verb(\(x\), \(x\), loves(x, y)) \rightarrow [loves].

A simple grammar with semantics 2

\[ \text{sentence(SMeaning)} \rightarrow \text{noun_phrase(SMeaning)}, \text{verb_phrase(VMeaining)}, \text{combine (SMeaining, VMeaining)}. \]
\[ \text{verb_phrase(VMeaining)} \rightarrow \text{verb(Vmeaining)}, \text{noun_phrase(NMeaining)}, \text{combine (NMeaining, VMeaining)}. \]
\[ \text{noun_phrase(NMeaining)} \rightarrow \text{name(NMeaining)}. \]

- Name(john) \rightarrow [John].
- Verb(\(x\), \(x\), loves(x, y)) \rightarrow [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

- \( S \rightarrow NP VP \)
- \( VP \rightarrow \text{Verb} \ NP \)
- \( \text{Verb} \rightarrow \text{serves} \)
- \( \lambda x y \exists e \text{Serving}(e)^e \text{Server}(e, y)^e \text{Served}(e, x) \)

- 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