CS388: Natural Language Processing

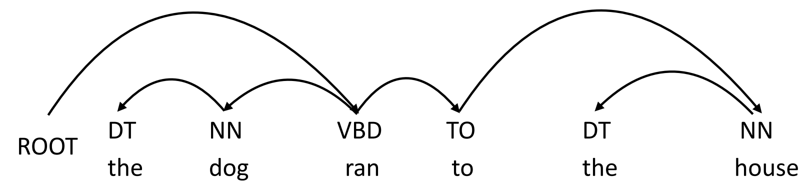
Lecture 14: Semantics I

Greg Durrett

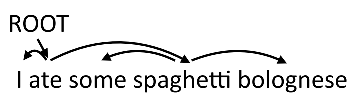
TEXAS
The University of Texas at Austin

Recall: Dependencies

- Dependency syntax: syntactic structure is defined by dependencies
  - Head (parent, governor) connected to dependent (child, modifier)
  - Each word has exactly one parent except for the ROOT symbol
  - Dependencies must form a directed acyclic graph



Recall: Shift-Reduce Parsing



- State: Stack: [ROOT I ate] Buffer: [some spaghetti bolognese]
- Left-arc (reduce operation): Let σ denote the stack
  - "Pop two elements, add an arc, put them back on the stack"

  $\boxed{\sigma | w_{-2}, w_{-1}} \rightarrow \boxed{\sigma | w_{-1}}$, w_{-2} is now a child of w_{-1}
- Train a classifier to make these decisions sequentially — that classifier can parse sentences for you

Where are we now?

- Early in the class: bags of word (classifiers) => sequences of words (sequence modeling)
- Now we can understand sentences in terms of tree structures as well
- Why is this useful? What does this allow us to do?
- We're going to see how parsing can be a stepping stone towards more formal representations of language meaning

Today

- Montague semantics:
  - Model theoretic semantics
  - Compositional semantics with first-order logic
- CCG parsing for database queries
- Lambda-DCS for question answering

Model Theoretic Semantics

Model Theoretic Semantics

- Key idea: can ground out natural language expressions in set-theoretic expressions called *models* of those sentences
- Natural language statement S => interpretation of S that models it

  She likes going to that restaurant

  - Interpretation: defines who *she* and *that restaurant* are, make it able to be concretely evaluated with respect to a *world*
- Entailment (statement A implies statement B) reduces to: in all worlds where A is true, B is true
- Our modeling language is *first-order logic*

First-order Logic

- Powerful logic formalism including things like entities, relations, and quantifications

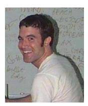
Lady Gaga sings

- sings is a *predicate* (with one argument), function f: entity → true/false
- sings(Lady Gaga) = true or false, have to execute this against some database (*world*)
- [[sings]] = *denotation*, set of entities which sing (found by executing this predicate on the world — we'll come back to this)

Quantification

- Universal quantification: "forall" operator
  - ∀x sings(x) ∨ dances(x) → performs(x)

    "Everyone who sings or dances performs"
- Existential quantification: "there exists" operator
  - ∃x sings(x) *"Someone sings"*
- Source of ambiguity! "*Everyone is friends with someone*"
  - ∀x ∃y friend(x,y)
  - ∃y ∀x friend(x,y)



Logic in NLP

- Question answering:

  Who are all the American singers named Amy?

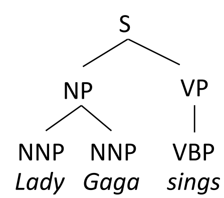
  λx. nationality(x,USA) ∧ sings(x) ∧ firstName(x,Amy)
  - Function that maps from x to true/false, like `filter`. Execute this on the world to answer the question
  - Lambda calculus: powerful system for expressing these functions
- Information extraction: *Lady Gaga and Eminem are both musicians*

  musician(Lady Gaga) ∧ musician(Eminem)
  - Can now do reasoning. Maybe know: ∀x musician(x) => performer(x)

    Then: performer(Lady Gaga) ∧ performer(Eminem)

Compositional Semantics with First-Order Logic

Montague Semantics



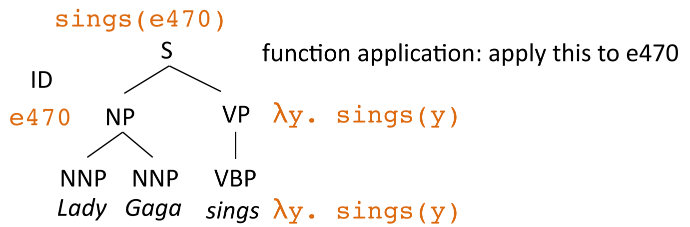
Id	Name	Alias	Birthdate	Sings?
e470	Stefani Germanotta	Lady Gaga	3/28/1986	T
e728	Marshall Mathers	Eminem	10/17/1972	T

- Database containing entities, predicates, etc.
- Sentence expresses something about the world which is either true or false
- Denotation: evaluation of some expression against this database
- `[[`*Lady Gaga*`]] = e470`

  denotation of this string is an entity
- `[[sings(e470)]] = True`

  denotation of this expression is T/F

Montague Semantics

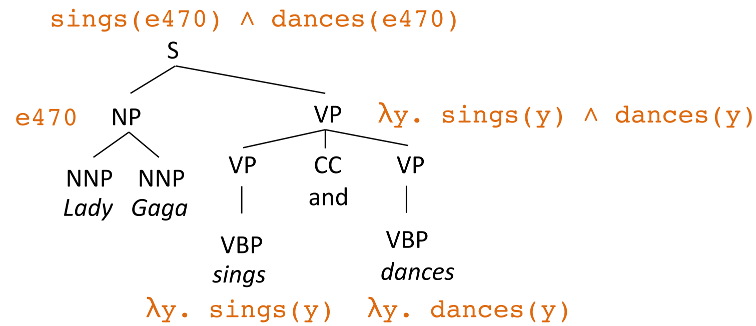


function application: apply this to e470

takes one argument (y, the entity) and returns a logical form `sings(y)`

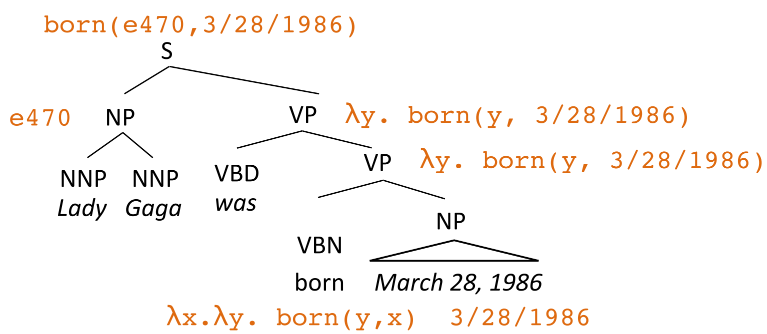
- We can use the syntactic parse as a bridge to the lambda-calculus representation, build up a logical form (our model) *compositionally*

Parses to Logical Forms



- General rules: VP: λy. a(y) ∧ b(y) -> VP: λy. a(y) CC VP: λy. b(y)
  S: f(x) -> NP: x VP: f

Parses to Logical Forms



- Function takes two arguments: first x (date), then y (entity)
- How to handle tense: should we indicate that this happened in the past?

Tricky things

- Adverbs/temporality: *Lady Gaga sang well yesterday*
  `sings(Lady Gaga, time=yesterday, manner=well)`
  - "Neo-Davidsonian" view of events: things with many properties:
    `∃e. type(e,sing) ∧ agent(e,e470) ∧ manner(e,well) ∧ time(e,…)`
- Quantification: *Everyone is friends with someone*
  `∃y ∀x friend(x,y)` (one friend) `∀x ∃y friend(x,y)` (different friends)
  - Same syntactic parse for both! So syntax doesn't resolve all ambiguities
- Indefinite: *Amy ate a waffle* `∃w. waffle(w) ∧ ate(Amy,w)`
- Generic: *Cats eat mice* (all cats eat mice? most cats? some cats?)

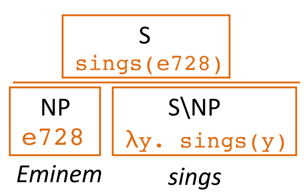
Semantic Parsing

- For question answering, syntactic parsing doesn't tell you everything you want to know, but indicates the right structure
- Solution: *semantic parsing*: many forms of this task depending on semantic formalisms
- Two today: CCG (looks like what we've been doing) and lambda-DCS
- Applications: database querying/question answer: produce lambda-calculus expressions that can be executed in these contexts

CCG Parsing

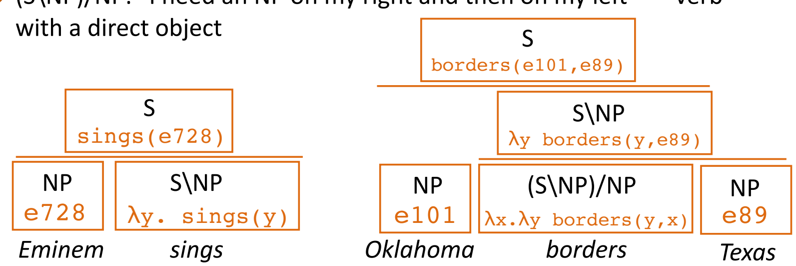
Combinatory Categorial Grammar

- Steedman+Szabolcsi (1980s): formalism bridging syntax and semantics
- Parallel derivations of syntactic parse and lambda calculus expression
- Syntactic categories (for this lecture): S, NP, "slash" categories
- S\NP: "if I combine with an NP on my left side, I form a sentence" — verb
- When you apply this, there has to be a parallel instance of function application on the semantics side



Combinatory Categorial Grammar

- Steedman+Szabolcsi 1980s: formalism bridging syntax and semantics
- Syntactic categories (for this lecture): S, NP, "slash" categories
  - S\NP: "if I combine with an NP on my left side, I form a sentence" — verb
  - (S\NP)/NP: "I need an NP on my right and then on my left" — verb with a direct object



CCG Parsing

What	states	border	Texas
$(S/(S\backslash NP))/N$	N	$(S\backslash NP)/NP$	NP
$\lambda f.\lambda g.\lambda x.f(x) \wedge g(x)$	$\lambda x.state(x)$	$\lambda x.\lambda y.borders(y,x)$	$texas$

border Texas $\Rightarrow$ (>): $(S\backslash NP)$: $\lambda y.borders(y, texas)$

- "What" is a ***very*** complex type: needs a noun and needs a S\NP to form a sentence. S\NP is basically a verb phrase (*border Texas*)

Zettlemoyer and Collins (2005)

CCG Parsing

What	states	border	Texas
$(S/(S\backslash NP))/N$	N	$(S\backslash NP)/NP$	NP
$\lambda f.\lambda g.\lambda x.f(x) \wedge g(x)$	$\lambda x.state(x)$	$\lambda x.\lambda y.borders(y,x)$	$texas$

What states (>): $S/(S\backslash NP)$: $\lambda g.\lambda x.state(x) \wedge g(x)$

border Texas (>): $(S\backslash NP)$: $\lambda y.borders(y, texas)$

(>): S : $\lambda x.state(x) \wedge borders(x, texas)$

- "What" is a ***very*** complex type: needs a noun and needs a S\NP to form a sentence. S\NP is basically a verb phrase (*border Texas*)
- Lexicon is highly ambiguous — all the challenge of CCG parsing is in picking the right lexicon entries

Zettlemoyer and Collins (2005)

CCG Parsing

Show me	**flights**	**to**	**Prague**
S/N	**N**	**(N\N)/NP**	**NP**
$\lambda f.f$	$\lambda x.flight(x)$	$\lambda y.\lambda f.\lambda x.f(y) \wedge to(x,y)$	PRG

to Prague: **N\N** : $\lambda f.\lambda x.f(x) \wedge to(x,PRG)$

flights to Prague: **N** : $\lambda x.flight(x) \wedge to(x,PRG)$

Show me flights to Prague: **S** : $\lambda x.flight(x) \wedge to(x,PRG)$

- "to" needs an NP (destination) and N (parent)

Slide credit: Dan Klein

CCG Parsing

- Many ways to build these parsers
- One approach: run a "supertagger" (tags the sentence with complex labels), then run the parser

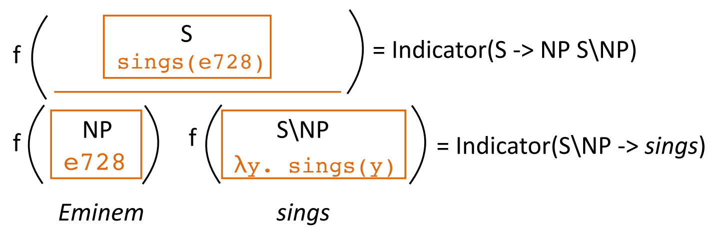
What	states	border	Texas
$(S/(S\backslash NP))/N$	N	$(S\backslash NP)/NP$	NP
$\lambda f.\lambda g.\lambda x.f(x) \wedge g(x)$	$\lambda x.state(x)$	$\lambda x.\lambda y.borders(y,x)$	$texas$

  - Parsing is easy once you have the tags, so we've reduced it to a (hard) tagging problem

Zettlemoyer and Collins (2005)

Building CCG Parsers

- Model: log-linear model over derivations with features on rules: $P(d|x) \propto \exp w^\top \left(\sum_{r \in d} f(r, x) \right)$



- Can parse with a variant of CKY

Zettlemoyer and Collins (2005)

Building CCG Parsers

- Training data looks like pairs of sentences and logical forms

What states border Texas `λx. state(x) ∧ borders(x, e89)`

- Problem: we don't know the derivation
  - *Texas* corresponds to NP | `e89` in the logical form (easy to figure out)
  - *What* corresponds to (S/(S\NP))/N | `λf.λg.λx. f(x) ∧ g(x)`
  - How do we infer that without being told it?

Zettlemoyer and Collins (2005)

Lexicon

- GENLEX: takes sentence S and logical form L. Break up logical form into chunks C(L), assume any substring of S might map to any chunk

What states border Texas `λx. state(x) ∧ borders(x, e89)`

- Chunks inferred from the logic form based on rules:
  - NP: `e89`
  - (S\NP)/NP: `λx. λy. borders(x,y)`
- Any substring can parse to any of these in the lexicon
  - *Texas* -> NP: e89 is correct
  - *border Texas* -> NP: e89
  - *What states border Texas* -> NP: e89

  ...

Zettlemoyer and Collins (2005)

GENLEX

Rules		Categories produced from logical form
Input Trigger	Output Category	$\arg\max(\lambda x.state(x) \wedge borders(x, texas), \lambda x.size(x))$
constant c	$NP : c$	$NP : texas$
arity one predicate p_1	$N : \lambda x.p_1(x)$	$N : \lambda x.state(x)$
arity one predicate p_1	$S\backslash NP : \lambda x.p_1(x)$	$S\backslash NP : \lambda x.state(x)$
arity two predicate p_2	$(S\backslash NP)/NP : \lambda x.\lambda y.p_2(y, x)$	$(S\backslash NP)/NP : \lambda x.\lambda y.borders(y, x)$
arity two predicate p_2	$(S\backslash NP)/NP : \lambda x.\lambda y.p_2(x, y)$	$(S\backslash NP)/NP : \lambda x.\lambda y.borders(x, y)$
arity one predicate p_1	$N/N : \lambda g.\lambda x.p_1(x) \wedge g(x)$	$N/N : \lambda g.\lambda x.state(x) \wedge g(x)$
literal with arity two predicate p_2 and constant second argument c	$N/N : \lambda g.\lambda x.p_2(x, c) \wedge g(x)$	$N/N : \lambda g.\lambda x.borders(x, texas) \wedge g(x)$
arity two predicate p_2	$(N\backslash N)/NP : \lambda x.\lambda g.\lambda y.p_2(x, y) \wedge g(x)$	$(N\backslash N)/NP : \lambda g.\lambda x.\lambda y.borders(x, y) \wedge g(x)$
an arg max / min with second argument arity one function f	$NP/N : \lambda g. \arg\max / \min(g, \lambda x.f(x))$	$NP/N : \lambda g. \arg\max(g, \lambda x.size(x))$
an arity one numeric-ranged function f	$S/NP : \lambda x.f(x)$	$S/NP : \lambda x.size(x)$

- Very complex and hand-engineered way of taking lambda calculus expressions and "backsolving" for the derivation

Zettlemoyer and Collins (2005)

Learning

- Iterative procedure like the EM algorithm: estimate “best” parses that derive each logical form, retrain the parser using these parses with supervised learning
- We’ll talk about a simpler form of this in a few slides

Zettlemoyer and Collins (2005)

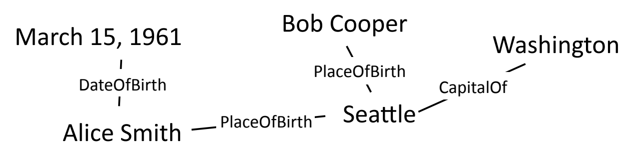
Applications

- GeoQuery: answering questions about states (~80% accuracy)
- Jobs: answering questions about job postings (~80% accuracy)
- ATIS: flight search
- Can do well on all of these tasks if you handcraft systems and use plenty of training data: these domains aren’t that rich
- What about broader QA?

Lambda-DCS

Lambda-DCS

- Dependency-based compositional semantics — original version was less powerful than lambda calculus, lambda-DCS is as powerful
- Designed in the context of building a QA system from Freebase
- Freebase: set of entities and relations



- [[PlaceOfBirth]] = set of pairs of (person, location)

Liang et al. (2011), Liang (2013)

Lambda-DCS

Lambda-DCS	Lambda calculus
`Seattle`	`λx. x = Seattle`
`PlaceOfBirth`	`λx.λy. PlaceOfBirth(x,y)`
`PlaceOfBirth.Seattle`	`λx. PlaceOfBirth(x,Seattle)`

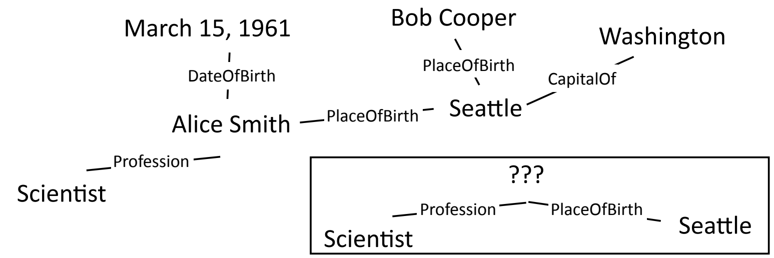
- Looks like a tree fragment over Freebase

??? —PlaceOfBirth– Seattle

`Profession.Scientist ∧ PlaceOfBirth.Seattle`	`λx. Profession(x,Scientist) ∧ PlaceOfBirth(x,Seattle)`

Liang et al. (2011), Liang (2013)

Lambda-DCS



"list of scientists born in Seattle"

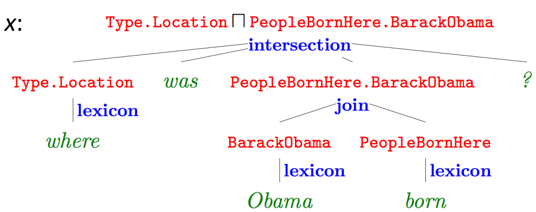
`Profession.Scientist ∧ PlaceOfBirth.Seattle`

- Execute this fragment against Freebase, returns Alice Smith (and others)

Liang et al. (2011), Liang (2013)

Parsing into Lambda-DCS

- Derivation *d* on sentence *x*:
- No more explicit syntax in these derivations like we had in CCG



- Building the lexicon: more sophisticated process than GENLEX, but can handle thousands of predicates
- Log-linear model with features on rules: $P(d|x) \propto \exp w^\top \left(\sum_{r \in d} f(r,x)\right)$
- Similar to CRF parsers

Berant et al. (2013)

Parsing with Lambda-DCS

- Learn just from question-answer pairs: maximize the likelihood of the right denotation *y* with the derivation *d* marginalized out

$$\mathcal{O}(\theta) = \sum_{i=1}^{n} \log \sum_{d \in D(x) : [\![d.z]\!]_{\mathcal{K}} = y_i} p_\theta(d \mid x_i).$$

sum over derivations *d* such that the denotation of *d* on knowledge base *K* is y_i

For each example:

Run beam search to get a set of derivations

Let d = highest-scoring derivation in the beam

Let d* = highest-scoring derivation in the beam *with correct denotation*

Do a structured perceptron update towards d* away from d

Berant et al. (2013)

Takeaways

- Can represent meaning with first order logic and lambda calculus
- Can bridge syntax and semantics and create semantic parsers that can interpret language into lambda-calculus expressions
- Useful for querying databases, question answering, etc.
- Next time: neural net methods for doing this that rely less on having explicit grammars