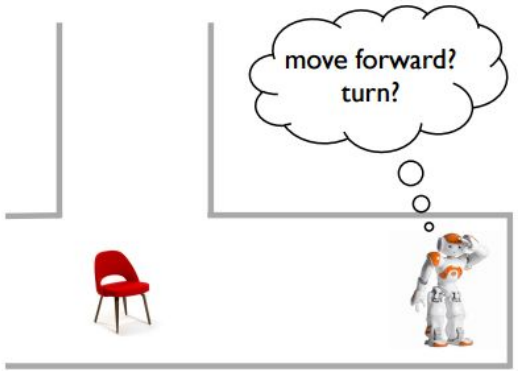
Language Modeling using CCG and Lambda DCS

Advanced Topics in NLP Seminar
December 2016
Hila Weisman

Talk Outline

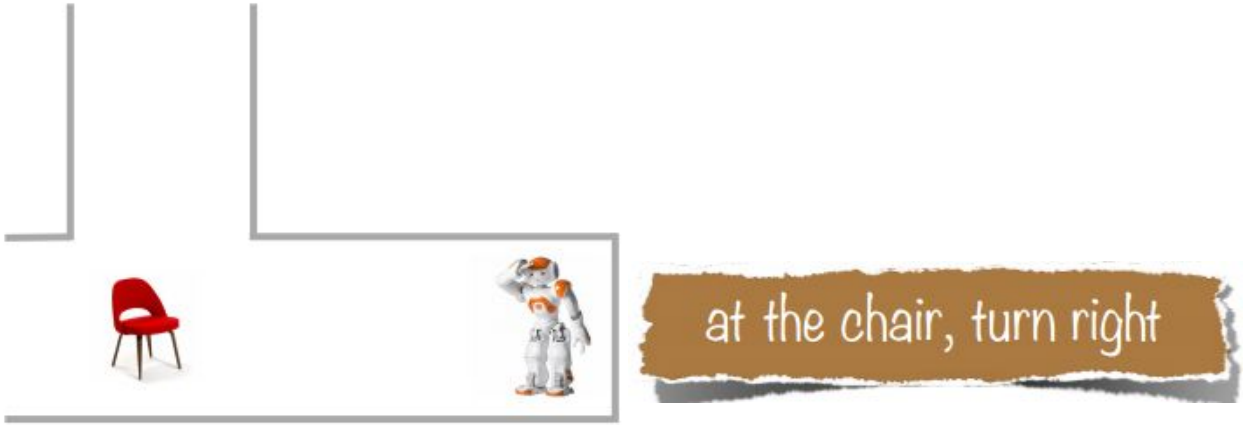
- Learning of Semantic Parsers for Mapping Instructions to Actions
  - Task definition
- CCG Definition & Examples
- CCG Semantics
  - Spatial Language Modeling
  - How to model instructions
- Parsing with CCG
  - Type-raising rules
- Lambda-DCS Representation



Weakly Supervised Learning of Semantic Parsers for Mapping Instructions to Actions (Artzi & Zettlemoyer, 2013)

- Task: robotic interpretation of navigational directions
  - language understanding in situated environments (“grounded learning”)

- Approach: parsing and executing at the same time with minimal supervision



Task Definition

- Let S be the set of possible environment states, A be the set of possible actions
  - State is a triple (x,y,o): x & y are integer grid coordinates, o $\in$ {0, 90, 180, 270} is orientation
  - Possible actions are {LEFT, RIGHT, MOVE, NULL}

- Given a start state s $\in$ S and a natural language instruction x

- Generate a sequence of actions a' = {a1, . . . , an}, with each ai $\in$ A, that performs the steps described in x
  - Actions change state of world according to transition function T : A × S → S

.

Central Problems

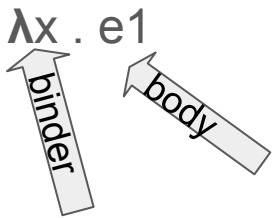
- In this talk we will focus on the Parsing and Modeling tasks

Parsing & Modeling

- Recap on Lambda-calculus
- Definition of Combinatory Categorial Grammar
- Spatial Language Modeling
- Modeling Instructions

Recap: Lambda Calculus

- Introduce variables ranging over values, define functions by (lambda-) abstracting over variables, apply functions to values



  - occurrences of x in the body e1 are bound.

- to evaluate (λx.e1) e2 means to evaluate e1 with x bound to e2

Recap: Beta Reduction

Let's beta-reduce (**λ**x.**λ**z.x z) y

→ (**λ**x.(**λ**z.(x z))) y // since **λ** extends to right

// apply (λx.e1) e2 → e1[x/e2]
// where e1 = λz.(x z), e2 = y

→ **λ**z.(y z) // final result

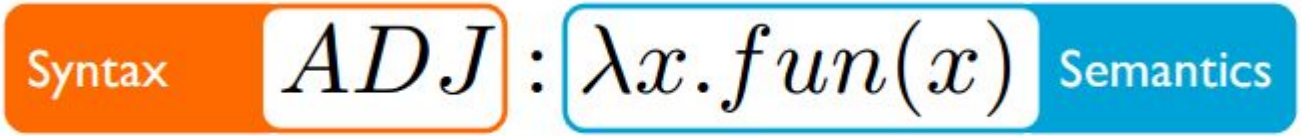
Combinatory Categorial Grammar (CCG)

- CCG is an established linguistic formalism (Steedman, 1996, 2000)
  - Based on Categorial grammar (CG) (Ajdukiewicz, 1935; Bar-Hillel, 1953; Lambek 1958)
- Explains a wide range of linguistic phenomena
  - coordination, long distance dependencies, free word order, etc.
- Jointly models syntax and semantics
  - fun --| ADJ : λ(x). fun(x)

CCG - Lexical Entries

- Pair words and phrases with meaning
- Meaning captured by a CCG category

CCG Categories

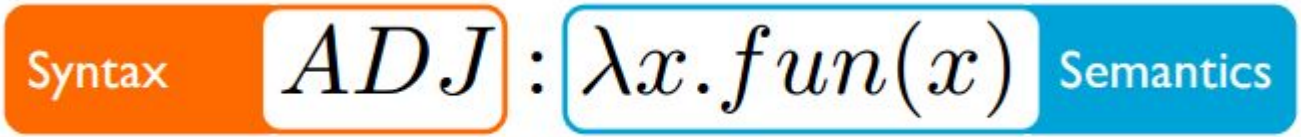


- Basic building block
- Jointly capture syntactic and semantic information

CCG Categories - Syntax side

- Primitive syntactic symbols: S, N , NP, PP, ADJ, AP
- Complex symbols are built recursively from primitives and slashes (application)
- Example complex syntactic symbols for verbs:
  - intransitive verb: S\NP — *he walks*
  - transitive verb: (S\NP)/NP — *he respects her*
  - ditransitive verb: ((S\NP)/NP)/NP — *he gave her the key*

CCG Categories - Semantics side



- λ-calculus expression
- Determined by modeling (more on that later..)

CCG vs. CFG

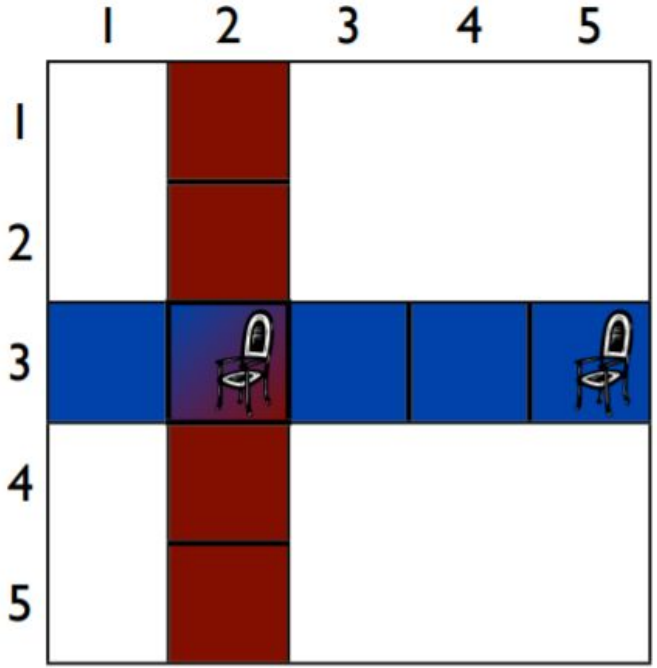
	CFGs	CCGs
Combination operations	Many	Few
Parse tree nodes	Non-terminals	Categories
Syntactic symbols	Few dozen	Handful, but can combine
Paired with words	POS tags	Categories

Modeling Considerations

Modeling is key to learning compact lexicons and high performing models

- Capture language complexity
- Satisfy system requirements
- Align with language units of meaning

Spatial Language Modeling

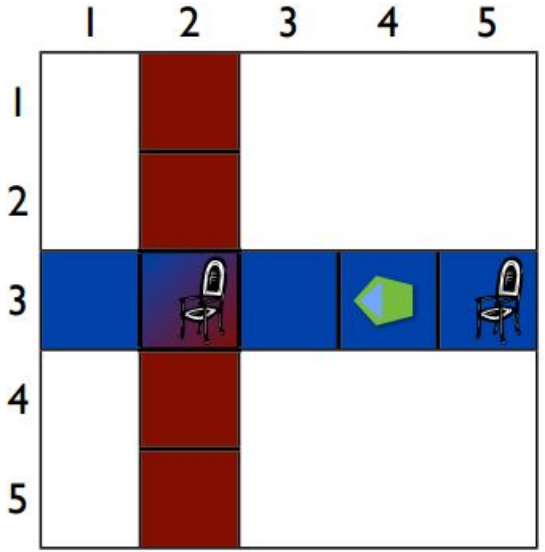


- Nouns
- Noun phrases
- Adjectives
- Prepositional phrases
- Spatial relations

Spatial Language Modeling

Four basic types in our model:

- entities e that are objects in the world
- events ev that specify actions in the world
- truth values t
- meta-entities m, such as numbers or directions



Spatial LM - Nouns(N)

- Nouns are functions that define entity type
  - mapped to **<e, t>**-type expressions that define a property.
  - For example, chair |- **N : λx.chair(x)**, which defines the set of objects for which the **<e, t>**-typed constant chair returns true

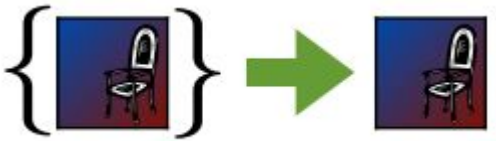




Spatial LM - Noun Phrases(NP)

- Nouns combined with determiners which pick out *specific objects in the world*.
  - consider both **definite** (the) and **indefinites** (a, an)

- Definite article is paired with new **<<e, t>, e>**-typed logical expression ι
  - given a set selects single object in the world (*referent* in linguistics)
  - can be viewed as new quantifier which **i**nstantiates its variable

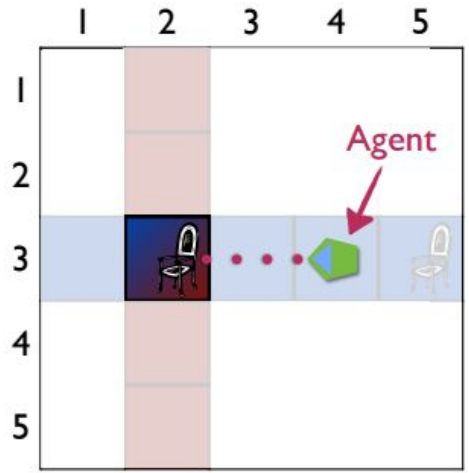


the chair

$\iota x.chair(x)$

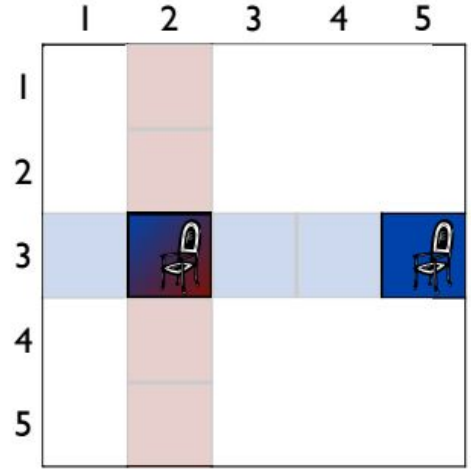
Spatial LM - NP Disambiguate from set

- Challenging when there's perceptual ambiguity
- Use simple heuristic approach:
  - rank referents based on combination of distance from agent and whether they are in front of it



the chair

$\iota x.chair(x)$



Spatial LM - NP with Indefinite Article (“*a*”)

- For indefinite article, introduce new quantifier **A** of type **<<e, t>, e>**
  - Depart from the Frege-Montague tradition of using existential quantifiers
  - Otherwise would have to have several complex lexical entries
    - Counterintuitive, “a” has very light semantic weight..

$$\frac{\text{a}}{\frac{}{PP\backslash(PP/NP)/N}\ \lambda f.\lambda g.\lambda y.\exists x.g(x,y) \wedge f(x)}$$

$$\frac{\text{a}}{\frac{}{S\backslash NP\backslash(S\backslash NP/NP)/N}\ \lambda f.\lambda g.\lambda y.\exists x.g(x,y) \wedge f(x)}$$

$$\frac{\text{a}}{\frac{}{S\backslash(S\backslash NP)/N}\ \lambda f.\lambda g.\lambda y.\exists x.g(x,y) \wedge f(x)}$$

$$\frac{\text{a}}{\frac{}{NP/N}\ \lambda f.\mathcal{A}x.f(x)}$$

Spatial LM - Prepositional phrase (PP) and Adjectives (ADJ)

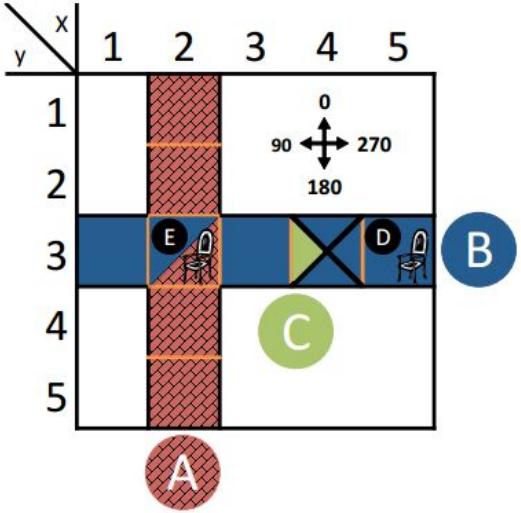
- Noun phrases with modifiers, such as adjectives and prepositional phrases are <e, t>-type expressions that implement *set intersection with logical conjunctions*

- "*blue hall*" is paired with λx.hall(x) ∧ blue(x)
  - We'll compare later with "hall is blue"

- "*chair in the corner*" is paired with
  λx.chair(x) ∧ intersect (ι (λy.corner(y)), x)

Spatial Relations

- perform complex reasoning over position sets and the relations between them
  - Binary relation ***in_front_of(x,y)*** tests if x is in front of y from point of view of the agent.
  - Binary relation ***intersect*** tests of its two arguments exist in the same position



in front of you

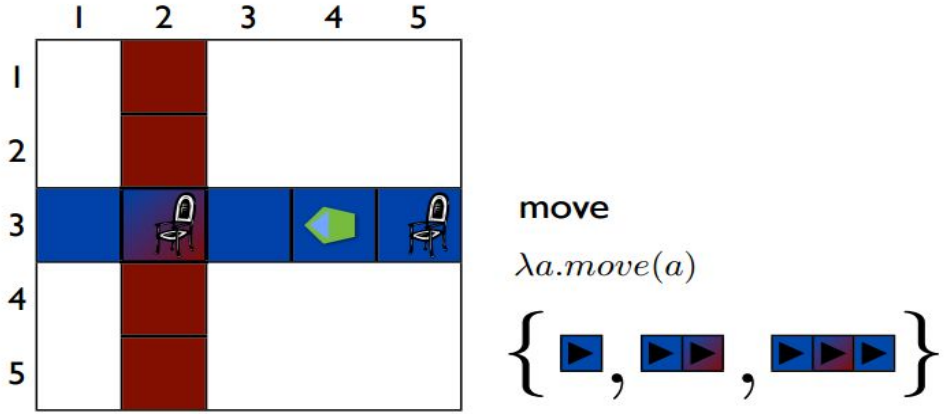
$\lambda x.\text{in\_front\_of}(\text{YOU}, x)$

Modeling Instructions

- Verb predicates have highly varied arity
  - Move forward |- λx λy move(x,y) ∧ dir(y,forward)
  - Move forward to the chair |- λx λy λz move(x,y,z) ∧ dir(y,forward) ∧ to(ιz.chair(z))
  - Move forward twice to the chair .....
  - Move forward twice to the chair on the right of the lamp ....
- Need a way to compactly represent each predicate regardless of modifiers
- Adopt Neo-Davidsonian event semantics (Davidson, 1967; Parsons, 1990)

Neo-Davidsonian Semantics

- Verb predicate “move” is actually a description of an event.
  - Represent this event in the semantics with **ev**-type primitive objects
- “move” is λ**a**.move(**a**) → all events **a** which involve moving actions

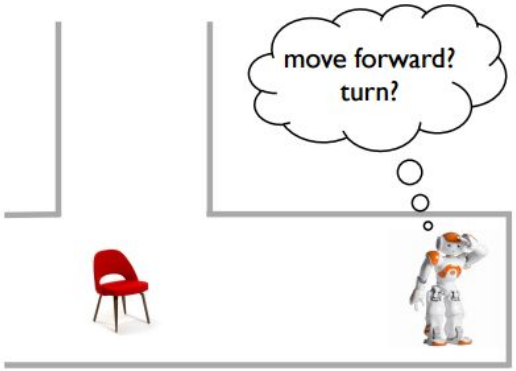


Modeling Instructions as event sets

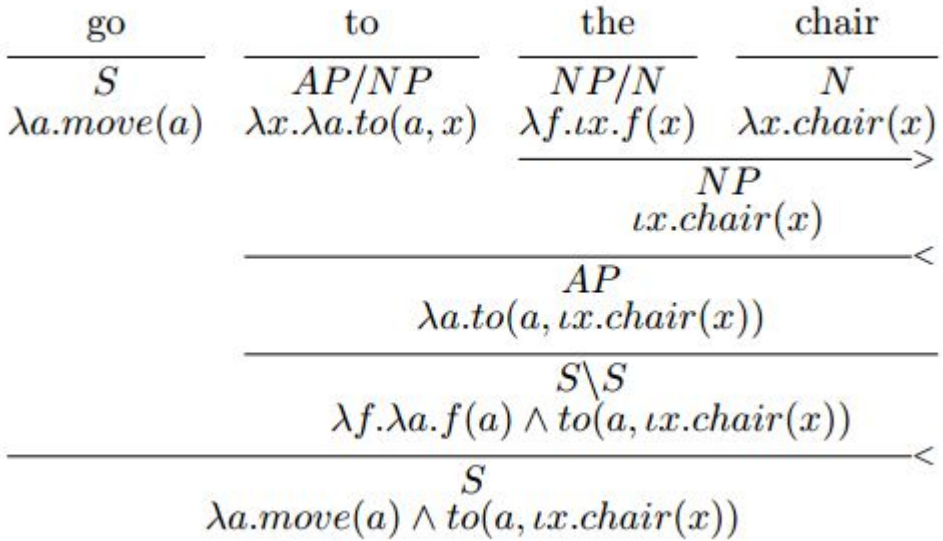
- Move forward |- λa move(a) ∧ dir(a,forward)
- Move forward to the chair |- λa move(a) ∧ dir(a,forward) ∧ to(a, ιz.chair(z))
- Move forward twice to the chair …..
- Move forward twice to the chair on the right of the lamp

Modeling instructions - CCG Parse

- Let’s parse the instruction “go to the chair” !



Modeling instructions - CCG Parse



Spatial and Instructional LM - Recap so far

Name objects

Noun phrases	Specific entities
Nouns	Sets of entities
Prepositional phrases Adjectives	Constrain sets

Instructions to execute

Verbs	Davidsonian events
Imperatives	Sets of events

Parsing instructional language with CCG

- Weighted CCG parser to compose logical expressions from instructional language
  - Weighted CCG is a variant of CCG which also learns features
- Use Factored CCG Lexicons
  - abstract over systematic variations in language
- Add rules which perform *type-raising*
  - compact the lexicon
  - account for missing content words

CCG Operations - Type Raising/Shifting

- Include “lower” type in lexicon and add type raising rules to reconstruct the “higher” type if necessary.

$$PP : g \rightarrow N\backslash N : \lambda f.\lambda x.f(x) \wedge g(x)$$

$$ADJ : g \rightarrow N/N : \lambda f.\lambda x.f(x) \wedge g(x)$$

$$AP : g \rightarrow S\backslash S : \lambda f.\lambda a.f(a) \wedge g(a)$$

$$AP : g \rightarrow S/S : \lambda f.\lambda a.f(a) \wedge g(a)$$

- Particularly useful when learning from sparse data

CCG Operations - Adjectives as Type Raising

- For “the hall is blue” use lower type of adjective (function intersection)

  blue |- ADJ: λx.blue(x)

- For “the blue hall” use type-raising (function composition → intersection)

  blue |- N/N : λf.λx.f(x) ∧ blue(x)

CCG Operations - Coordination

$$\text{and} \vdash C : conj$$
$$\text{or} \vdash C : disj$$

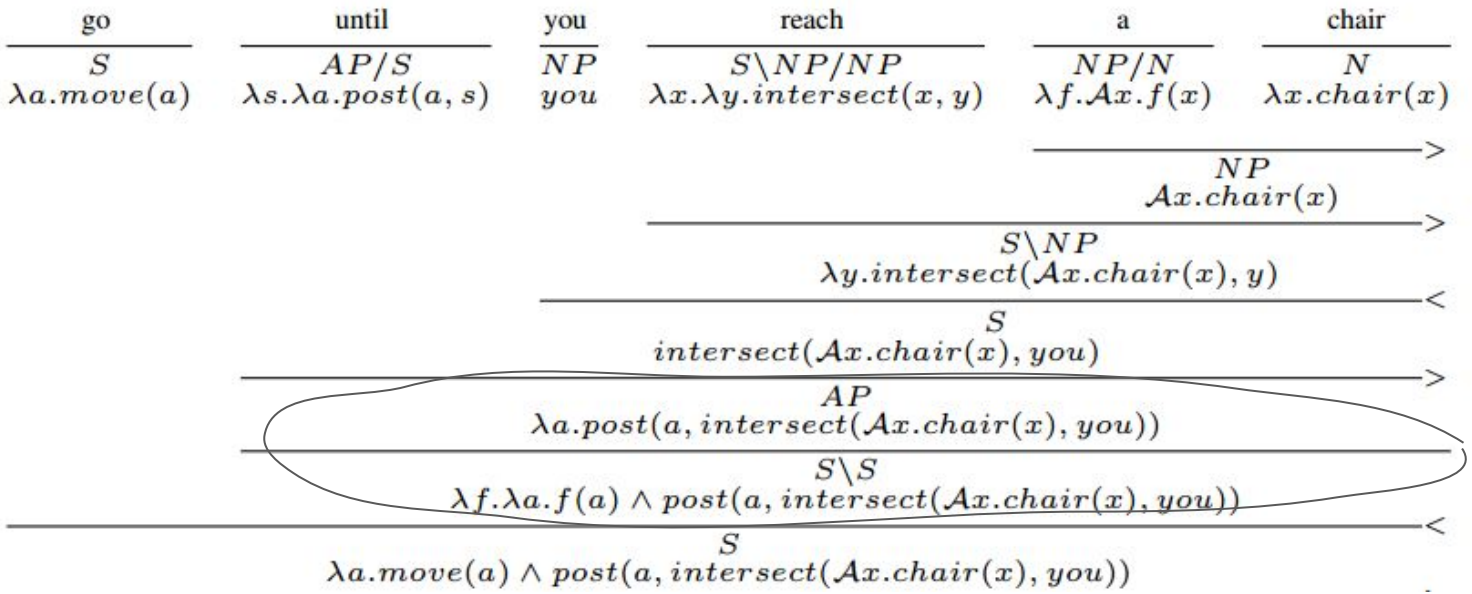
- Specific rules perform coordination
  - Coordinating operators are marked with special lexical entries

Non-standard coordination of adjectives

square	blue	or	round	yellow	pillow
ADJ	ADJ	C	ADJ	ADJ	N
$\lambda x.square(x)$	$\lambda x.blue(x)$	$disj$	$\lambda x.round(x)$	$\lambda x.yellow(x)$	$\lambda x.pillow(x)$
N/N	N/N		N/N	N/N	
$\lambda f.\lambda x.f(x) \wedge square(x)$	$\lambda f.\lambda x.f(x) \wedge blue(x)$		$\lambda f.\lambda x.f(x) \wedge round(x)$	$\lambda f.\lambda x.f(x) \wedge yellow(x)$	
>B			>B		
N/N			N/N		
$\lambda f.\lambda x.f(x) \wedge square(x) \wedge blue(x)$			$\lambda f.\lambda x.f(x) \wedge round(x) \wedge yellow(x)$		
<Φ>					
N/N					
$\lambda f.\lambda x.f(x) \wedge ((square(x) \wedge blue(x)) \vee (round(x) \wedge yellow(x)))$					
>					
N					
$\lambda x.pillow(x) \wedge ((square(x) \wedge blue(x)) \vee (round(x) \wedge yellow(x)))$					

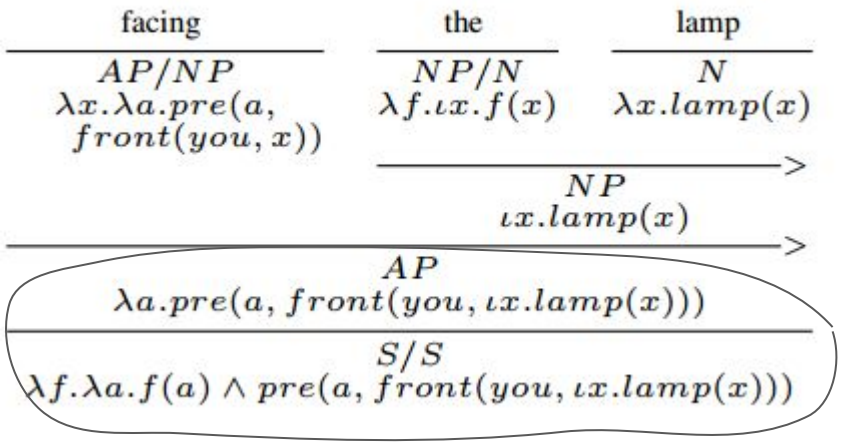
Type Raising - Adverbial phrases (AP)

- Adverbial phrase is a group of words operating adverbially
  - ➢ their syntactic function is the same as an adverb's.



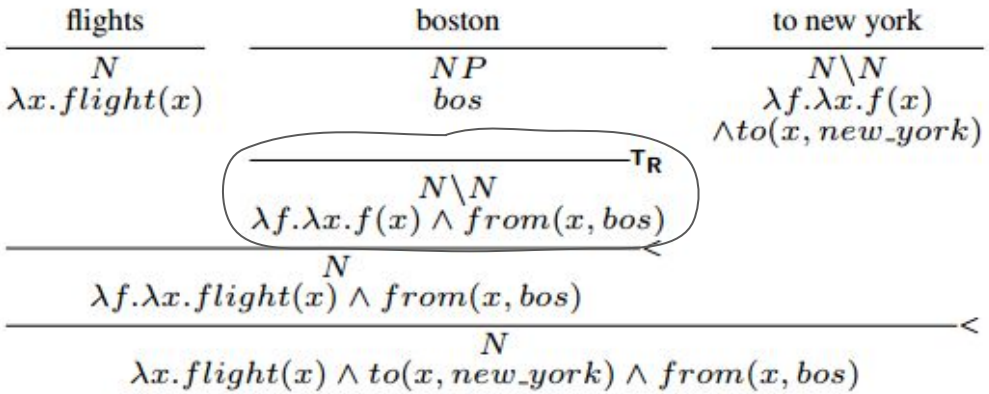
Type Raising - AP Topicalization

- Topicalization is a mechanism that establishes an expression as the sentence's or clause's topic
  - Turn right, when you are facing the lamp
  - Facing the lamp, turn right *(Topicalization of the adverbial phrase)*



Trace predicates (Zettlemoyer & Collins, 2007)

- Type raising can also help in the case of missing predicates
  - Infer missing predicate from context



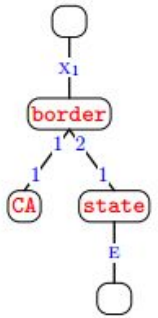
- NP : c => N\N : λf.λx.f(x) ∧ p(x,c)

Adding Combinators to CCG

- Additional combinatory rules lead to significantly more parses for any sentence x given a lexicon Λ.
- Many of these parses will be suspect from a linguistic perspective
  - broadening the set of CCG combinators might be considered a dangerous move
- However, learning algorithm can learn weights for new rules, effectively allowing the model to learn to use them only in appropriate contexts

Lambda-DCS

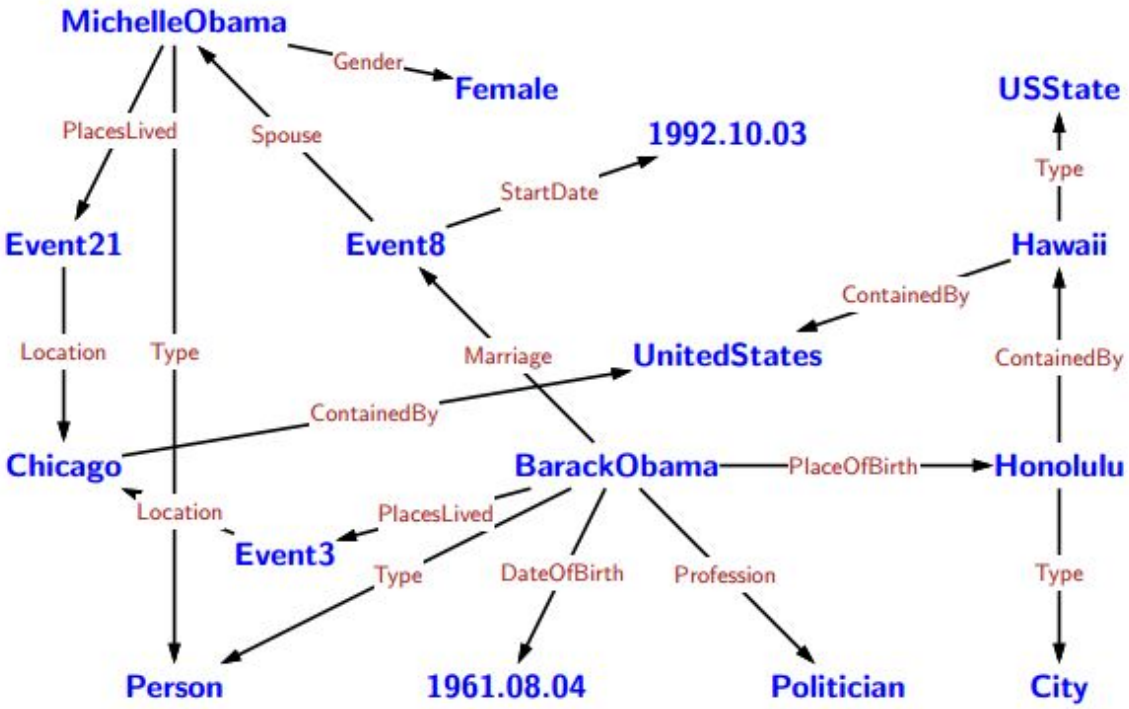
California borders which states?



Lambda DCS

- DCS = Dependency-based Compositional Semantics
- Attempts to remove explicit use of variables
  - “people who have lived in Seattle”
  - Logical form (lambda calculus): $\lambda x. \exists e.\text{PlacesLived}(x, e) \wedge \text{Location}(e, \text{Seattle})$
  - Logical form (lambda DCS): PlacesLived.Location.Seattle
- Designed in context of executing queries against knowledge-base
  - Specifically against Freebase, a large DB graph

Knowledge Graph



Lambda DCS - Base Cases

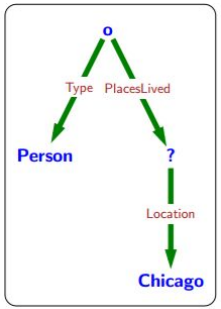
- The simplest lambda DCS form is a *single entity*
  - Seattle denotes the singleton set containing Seattle
  - equivalent in lambda calculus is: λx.[x = Seattle]
  - In general, [[e]] = λx.[x = e] , where is e is an unary logical form, [[·]] is conversion function

- The next form is a *property*
  - PlaceOfBirth denotes the set of pairs of people and where they were born
  - equivalent in lambda calculus is: λx.λy.PlaceOfBirth(x, y)
  - In general, [[p]] = λx.λy.p(x, y), where p is a binary logical form

Lambda DCS - Join Operation

- Join is the most central operation in lambda DCS
  - “people born in Seattle” is represented as PlaceOfBirth.Seattle
  - In lambda calculus : λx.PlaceOfBirth(x, Seattle)
  - In general, [[b.u]] = λx. ∃ y.[[b]](x, y) ∧ [[u]](y), where u and b,u are unary and b binary

- Implicit existential quantification over argument y shared by b and u
  - From DB perspective, b.u is the result set of joining second column of b with first column of u
  - Can be thought of as a graph template



Lambda DCS - More Operations

- **Intersection (conjunction)**: The set of scientists born in Seattle
  - Profession.Scientist ⊓ PlaceOfBirth.Seattle
  - λx.Profession(x, Scientist) ∧ PlaceOfBirth(x, Seattle).
  - In general, [[u1 ⊓ u2]] = λx.[[u1]](x) ∧ [[u2]](x)

- **Union (disjunction)**: The set containing "Oregon, Washington and Canadian provinces"
  - Oregon ⊔ Washington ⊔ Type.CanadianProvince
  - λx.[x = Oregon] ∨ [x = Washington] ∨ Type(x, CanadianProvince).
  - In general, [[u1 ⊔ u2]] = λx.[[u1]](x) ∨ [[u2]](x)

Lambda DCS - More Operations

- **Negation (not):** The set "states not bordering California"
  - Type.USState ⊓ ¬Border.California
  - λx.Type(x, USState) ∧ ¬Border(x, California)
  - In general, for a unary u, [[¬u]]K = λx.¬[[u]](x).

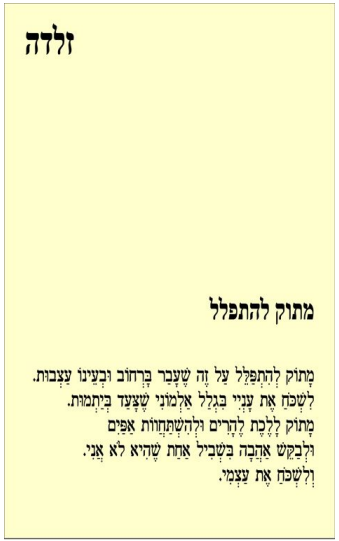
- **Higher order functions**: operate on sets of entities rather than on individual entities
  - Aggregation: count(Type.USState), argmax(Type.USState, Area)
  - Superlatives (taking the argmin/argmax)

So far logical forms were tree-structured, what about non-tree structures?

Anaphora - Definition

- Relationship between a “referentially dependent” expression (anaphor) and a “referentially independent” expression that serves as its antecedent and from which the anaphoric expression gets its reference

- “Those who knew *her* adored *Zelda*”



Anaphora in Lambda-DCS

"those who had a child who influenced them"

- μx.Children.Influenced.x
  - μx constraints: first argument of *Children* is bound to second argument of *Influenced*.
  - In linguistics, bound variable anaphora
- λx. ∃ y.Children(x, y).Influenced(y, x)

"those whose children influenced the Dalai Lama"

- HasChild.HasInfluence.TheDalaiLama
  - No need for mu abstraction here
- λx. ∃ y.HasChild(x, y) ∧ HasInfluence(y,TheDalaiLama)

Lambda DCS - Discussion

- Logical forms center around sets (unaries and binaries)
  - As opposed to truth values
- Eliminates use of variables for most cases
  - Tree-structured nature
  - Only use abstraction when denoting a function
- A good fit for working on graph database
  - Logical forms as graph patterns
- Syntactic sugar for lambda calculus?
  - Or variant which is closer to natural language

APPENDIX

Weighted Linear CCGs

- Given a weighted linear model:
  - CCG lexicon Λ
  - Feature function $f : X \times Y \rightarrow \mathbb{R}^m$
  - Weights $w \in \mathbb{R}^m$
- The best parse is:

$$y^* = \arg\max_y w \cdot f(x, y)$$

- We consider all possible parses y for sentence x given the lexicon Λ

Factored CCG Lexicon

house $\vdash ADJ : \lambda x.of(x, \iota y.house(y))$

house $\vdash N : \lambda x.house(x)$

garden $\vdash ADJ : \lambda x.of(x, \iota y.garden(y))$

Lexemes

$(\text{garden}, \{garden\})$

$(\text{house}, \{house\})$

Templates

$\lambda(\omega, \{v_i\}_1^n).$

$[\omega \vdash ADJ : \lambda x.of(x, \iota y.v_1(y))]$

$\lambda(\omega, \{v_i\}_1^n).$

$[\omega \vdash N : \lambda x.v_1(x)]$

Model

To map instructions to actions, we jointly reason about linguistic meaning and action execution. We use a weighted CCG grammar to rank possible meanings z for each instruction x. Each logical form z is mapped to a sequence of actions ~a with a deterministic executor. The final grounded CCG model, jointly constructs and scores z and ~a, allowing for robust situated reasoning during semantic interpretation

Learning

We assume access to a training set containing N examples {(xi , si , Vi) : i = 1 . . . N}, each containing a natural language sentence xi , a start state si , and a validation function Vi . The validation function Vi : A → {0, 1} maps an action sequence ~a ∈ A to 1 if it's correct according to available supervision, or 0 otherwise. This training data contains no direct evidence about the logical form z for each xi , or the grounded CCG analysis used to construct z. We model all these choices as latent variables. We experiment with two validation functions. The first, V D(~a), has access to an observable demonstration of the execution ~ai , a given ~a is valid iff ~a = ~ai . The second, V S i (~a), only encodes the final state s 0 i of the execution of x, therefore ~a is valid iff its final state is s 0 i .