

# Lecture 5: Semantic Parsing, CCGs, and Structured Classification

Kyle Richardson

kyle@ims.uni-stuttgart.de

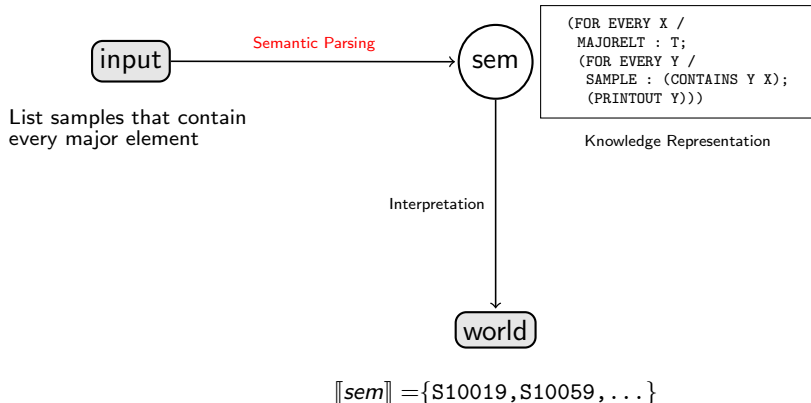
May 12, 2016

# Lecture Plan

- ▶ **paper:** Zettlemoyer and Collins (2012)
- ▶ **general topics:** (P)CCGs, compositional semantic models, log-linear models, (stochastic) gradient descent.

# The Big Picture (reminder)

- Standard processing pipeline



Lunar QA system (Woods (1973))

# Semantic Parsing: Basic Ingredients

- ▶ **Compositional Semantic Model:** Generates compositional meaning representations (e.g., logical forms, functional representations, ...)
- ▶ **Translation Model:** Maps text input to representations (tools already discussed: string rewriting, CFGs, SCFGs).
- ▶ **Rule Extraction:** Finds the candidate translation rules (e.g., SILT, word-alignments, ...)
- ▶ **Probabilistic Model:** Used for learning and finding the best translations (PCFGs, PSCFGs, EM).

# Semantic Parsing: Basic Ingredients

- ▶ **Compositional Semantic Model:** Generates compositional meaning representations (e.g., logical forms, functional representations, ...)
- ▶ **Translation Model:** Maps text input to representations (tools already discussed: string rewriting, CFGs, SCFGs).
- ▶ **Rule Extraction:** Finds the candidate translation rules (e.g., SILT, word-alignments, ...)
- ▶ **Probabilistic Model:** Used for learning and finding the best translations (PCFGs, PSCFGs, EM).
- ▶ **This lecture:** Introduce a new model based on (Combinatory) Categorical Grammar and log-linear models.

# Classical Categorical Grammar (CG)

- ▶ **Lexicalism:** lexical entries encode nearly all information about how words are combined, no separate syntactic component.
- ▶ An example (syntactic) lexicon  $\Lambda$

John	$:=$	NP	<i>(basic category)</i>
Mary	$:=$	NP	<i>(basic category)</i>
sleeps	$:=$	$S \backslash NP$	<i>(derived category)</i>
loves	$:=$	$(S \backslash NP) / NP$	-
quietly	$:=$	$(S \backslash NP) / (S \backslash NP)$	-

# Classical Categorical Grammar (CG)

- An example (syntactic) lexicon  $\Lambda$

John	$:=$	NP	<i>(basic category)</i>
Mary	$:=$	NP	<i>(basic category)</i>
sleeps	$:=$	$S \backslash NP$	<i>(derived category)</i>
loves	$:=$	$(S \backslash NP) / NP$	-
quietly	$:=$	$(S \backslash NP) / (S \backslash NP)$	-

```
>>> john = 'NP'; mary = 'NP'
```

```
>>> sleeps = lambda x : 'S' if x == 'NP' else None
```

# Classical Categorical Grammar (CG)

- An example (syntactic) lexicon  $\Lambda$

John	: =	NP	(basic category)
Mary	: =	NP	(basic category)
sleeps	: =	$S \backslash NP$	(derived category)
loves	: =	$(S \backslash NP) / NP$	-
quietly	: =	$(S \backslash NP) / (S \backslash NP)$	-

```
>>> john = 'NP'; mary = 'NP'
```

```
>>> sleeps = lambda x : 'S' if x == 'NP' else None
```

```
>>> sleeps(john)
```

```
=> 'S'
```



# Classical Categorical Grammar (CG)

► An example (syntactic) lexicon  $\Lambda$

John	$:=$	NP	<i>(basic category)</i>
Mary	$:=$	NP	<i>(basic category)</i>
sleeps	$:=$	$S \backslash NP$	<i>(derived category)</i>
loves	$:=$	$(S \backslash NP) / NP$	-
quietly	$:=$	$(S \backslash NP) / (S \backslash NP)$	-

```
>>> john = 'NP'; mary = 'NP'
```

```
>>> loves = lambda x : (lambda y : 'S' if y == 'NP' else None) \
    if x == NP else None
```

```
>>> loves(mary)
```

# Classical Categorical Grammar (CG)

- An example (syntactic) lexicon  $\Lambda$

John	:=	NP	(basic category)
Mary	:=	NP	(basic category)
sleeps	:=	$S \backslash NP$	(derived category)
loves	:=	$(S \backslash NP) / NP$	-
quietly	:=	$(S \backslash NP) / (S \backslash NP)$	-

```
>>> john = 'NP'; mary = 'NP'
```

```
>>> loves = lambda x : (lambda y : 'S' if y == 'NP' else None) \  
    if x == NP else None
```

```
>>> loves(mary)
```

```
=> <function __main__ . <lambda>>
```

# Classical Categorical Grammar (CG)

- An example (syntactic) lexicon  $\Lambda$

John	:=	NP	(basic category)
Mary	:=	NP	(basic category)
sleeps	:=	$S \backslash NP$	(derived category)
loves	:=	$(S \backslash NP) / NP$	-
quietly	:=	$(S \backslash NP) / (S \backslash NP)$	-

```
>>> john = 'NP'; mary = 'NP'
```

```
>>> loves = lambda x : (lambda y : 'S' if y == 'NP' else None) \  
    if x == NP else None
```

```
>>> loves(mary)(john)
```

```
=> 'S'
```

# CG Derivations

- Often shown in a tabular proof form, as a series of *cancellation* steps.

$$\begin{array}{ccccc} & & \text{loves} & & \text{Mary} \\ & & \hline & & (S \backslash NP) / NP & & NP \\ \text{John} & & & & \\ \hline & & & & \\ NP & & & & S \backslash NP \\ \hline & & & & (<) \\ & & S & & \end{array}$$

# CG Derivations

- ▶ Often shown in a tabular proof form, as a series of *cancellation* steps.

$$\begin{array}{ccc} & \text{loves} & \text{Mary} \\ & \hline & (S \backslash NP) / NP & NP \\ \text{John} & & \\ \hline & & \hline & & (>) \\ NP & & S \backslash NP \\ \hline & & (<) \\ & S & \end{array}$$

- ▶ **function application**

- ▶  $>: A/B \ B \longrightarrow A$
- ▶  $<: B \ A \backslash B \longrightarrow A$

# CG Derivations

- Often shown in a tabular proof form, as a series of *cancellation* steps.

	loves	Mary	
	_____	_____	
John	$(S \backslash NP) / NP$	NP	
_____	_____		$(>)$
NP	$S \backslash NP$		
_____	_____		$(<)$
	S		

```
>>> apply_right = lambda fun,arg : fun(arg)
```

```
>>> apply_left = lambda arg,fun : fun(arg)
```

```
>>> apply_right(loves,mary)
```

```
=> <function __main__.<lambda>>
```

# CG Derivations

- Often shown in a tabular proof form, as a series of *cancellation* steps.

	loves	Mary	
	_____	_____	
John	$(S \backslash NP) / NP$	NP	
_____	_____		$(>)$
NP	$S \backslash NP$		
_____	_____		$(<)$
	S		

```
>>> apply_right = lambda fun, arg: fun(arg)
```

```
>>> apply_left = lambda arg, fun: fun(arg)
```

```
>>> apply_left(john, apply_right(loves, mary))
```

```
=> 'S'
```

# CG and Semantics

- Lexical rules can be extended to have a compositional semantics.

John	$:=$	NP	:	john'
Mary	$:=$	NP	:	mary'
sleeps	$:=$	$S \backslash NP$	:	$\lambda x.sleep(x)$
loves	$:=$	$(S \backslash NP) / NP$	:	$\lambda y, \lambda x.loves(x, y)$
quietly	$:=$	$(S \backslash NP) / (S \backslash NP)$	:	$\lambda f. \lambda x. f(x) \wedge quiet(f, x)$



# CG Derivations

- ▶ Often shown in a tabular proof form, as a series of *cancellation* steps.

$$\begin{array}{c} \text{John} \quad \text{loves} \quad \text{Mary} \\ \hline \text{John} \quad (S \backslash NP) / NP : \lambda y, \lambda x. \text{love}(x, y) \quad NP : \text{mary}' \\ \hline NP : \text{john}' \quad S \backslash NP : \lambda x. \text{love}(x, \text{mary}') \\ \hline S : \text{love}(\text{john}', \text{mary}') \end{array} \begin{array}{l} \\ \\ (>) \\ (<) \end{array}$$

- ▶ **function application with semantics**
  - ▶  $>: A/B : f \quad B : g \longrightarrow A : f(g)$
  - ▶  $<: B : g \quad A \backslash B : f \longrightarrow A : f(g)$

# CG Derivations

- Often shown in a tabular proof form, as a series of *cancellation* steps.

$$\begin{array}{c}
 \text{John} \qquad \text{loves} \qquad \text{Mary} \\
 \hline
 \text{John} \quad (S \backslash NP) / NP : \lambda y, \lambda x. \text{love}(x, y) \quad NP : \text{mary}' \\
 \hline
 NP : \text{john}' \quad S \backslash NP : \lambda x. \text{love}(x, \text{mary}') \quad (>) \\
 \hline
 S : \text{love}(\text{john}', \text{mary}') \quad (<)
 \end{array}$$

- **Derivation:**  $(L, T)$ , where  $L$  is a logical form (top), and  $T$  is the derivation steps (parse tree).

# Combinatory Categorical Grammar (CCG)

- ▶ A particular theory of categorial grammar, which uses additional function application types (see paper for pointers or Steedman (2000)).

e.g., **composition**:  $A/B : f \ B/C : g \rightarrow A/C : \lambda x.f(g(x))$

- ▶ **Benefits:** Is linguistically motivated, much more powerful than context-free grammars (mildly context-sensitive), polynomial parsing.

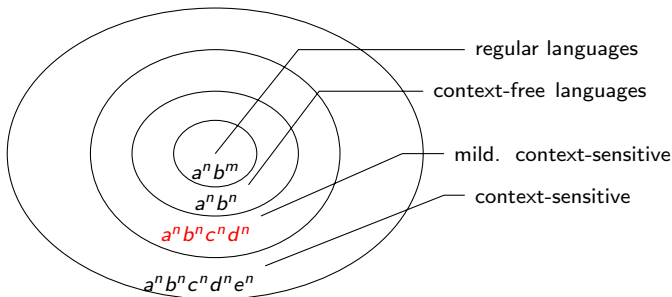
# Combinatory Categorical Grammar (CCG)

- ▶ A particular theory of categorial grammar, which uses additional function application types (see paper for pointers or Steedman (2000)).

e.g., **composition**:  $A/B : f \ B/C : g \rightarrow A/C : \lambda x.f(g(x))$

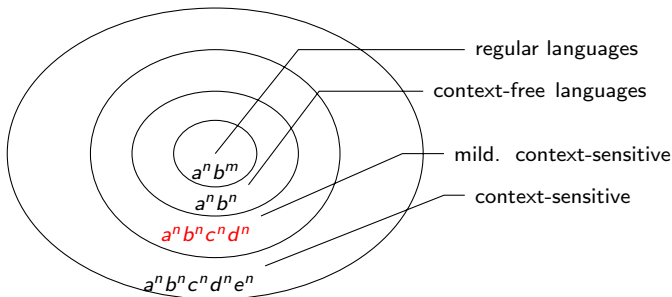
- ▶ **Benefits:** Is linguistically motivated, much more powerful than context-free grammars (mildly context-sensitive), polynomial parsing.
- ▶ **Parsing:** Extended version of CKY algorithm (Steedman (2000))

## A note about mild context-sensitivity



- ▶ CCGs and other mildly context-sensitive formalism (TAGs, LIGs, ...) allows derivations/graphs more general than trees.

# A note about mild context-sensitivity



- ▶ CCGs and other mildly context-sensitive formalism (TAGs, LIGs, ...) allows derivations/graphs more general than trees.
- ▶ **Theoretical question:** what types of grammar formalisms are best suited for semantic parsing?

# Earlier Lecture (Liang and Potts 2015)

- ▶ We have already seen something like categorial grammar.
  - ▶ Rules are divided between syntactic and semantic rules.

Syntax	Semantic representation	Denotation
$N \rightarrow one$	1	1
$N \rightarrow two$	2	2
$\vdots$	$\vdots$	$\vdots$
$R \rightarrow plus$	+	the $R$ such that $R(x, y) = x + y$
$R \rightarrow minus$	-	the $R$ such that $R(x, y) = x - y$
$R \rightarrow times$	$\times$	the $R$ such that $R(x, y) = x * y$
$S \rightarrow minus$	$\neg$	the $f$ such that $f(x) = -x$
$N \rightarrow S N$	$\ulcorner S \urcorner \ulcorner N \urcorner$	$[[\ulcorner S \urcorner]]([\ulcorner N \urcorner]])$
$N \rightarrow N_L R N_R$	$(\ulcorner R \urcorner \ulcorner N_L \urcorner \ulcorner N_R \urcorner)$	$[[\ulcorner R \urcorner]]([\ulcorner N_L \urcorner]][\ulcorner N_R \urcorner])$

# Compositional Semantics (past lecture)

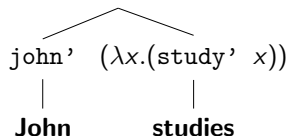
**Principle of Compositionality:** The meaning of a complex expression is a function of the meaning of its parts and the rules that combine them.

**Example:** *John studies.*

$\text{john}' \rightarrow \text{"John"}$

$(\lambda x.(\text{study}' x)) \rightarrow \text{"studies"}$

$(\lambda x.(\text{study}' x))(\text{john}) \rightarrow (\text{study}' \text{john}') \rightarrow \{True, False\}$





# Compositional Semantics (past lecture)

**Principle of Compositionality:** The meaning of a complex expression is a function of the meaning of its parts and the rules that combine them.

**Example:** *John studies.*

$$\begin{aligned}\text{john}' &\longrightarrow \text{"John"} \\ (\lambda x.(\text{study}' x)) &\longrightarrow \text{"studies"}\end{aligned}$$

```
>>> students_studying = set(["john", "mary"])
>>> study = lambda x : x in students_studying
>>> fun_application = lambda fun, val : fun(val)
>>> fun_application(study, "bill")
>>> False
```

# Geoquery Logical forms

- ▶ Zettlemoyer and Collins (2012) use a conventional logical language.
  - ▶ **constants:** **entities**, numbers, **functions**
  - ▶ **logical connectives:** **conjunction** ( $\wedge$ ), disjunction ( $\vee$ ), negation ( $\neg$ ), implication ( $\rightarrow$ )
  - ▶ **quantifiers:** universal ( $\forall$ ) and existential ( $\exists$ ).
  - ▶ **lambda expressions:** **anonymous functions** ( $\lambda x.f(x)$ )
  - ▶ **other quantifiers/functions:** arg max, definite descriptions ( $\iota$ ),...

**Example:** *What states border Texas?*

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

# Geoquery Logical forms

- ▶ Zettlemoyer and Collins (2012) use a conventional logical language.
  - ▶ **constants:** entities, numbers, **functions**
  - ▶ **logical connectives:** conjunction ( $\wedge$ ), disjunction ( $\vee$ ), negation ( $\neg$ ), implication ( $\rightarrow$ )
  - ▶ **quantifiers:** universal ( $\forall$ ) and existential ( $\exists$ ).
  - ▶ **lambda expressions:** **anonymous functions** ( $\lambda x.f(x)$ )
  - ▶ **other quantifiers/functions:** **arg max**, definite descriptions ( $\iota$ ),...

**Example:** *What is the largest state?*

$\text{arg max}(\lambda x.\text{state}(x), \lambda x.\text{size}(x))$

# A mini functional interpreter: Constants (Haskell)<sup>1</sup>

- **Example:** *What states border Texas?*

$$\lambda x. \text{state}(x) \wedge \text{borders}(x, \text{texas})$$

Texas	:=	NP	:	texas
border	:=	$(S \backslash NP) / NP$	:	$\lambda y \lambda x. \text{borders}(x, y)$
states	:=	$S \backslash N$	:	$\lambda x. \text{state}(x)$
which	:=	$(S / (S \backslash NP)) / N$	:	$\lambda f, \lambda g, \lambda x. f(x) \wedge g(x)$

```
Prelude > let state a = (elem a ["nh","ma","vt"])
```

```
Prelude > state "nh"
```

```
=> True
```

---

<sup>1</sup> you can try out these examples using <https://tryhaskell.org/>

# A mini functional interpreter: Constants (Haskell)<sup>1</sup>

- **Example:** *What states border Texas?*

$\lambda x. \text{state}(x) \wedge \text{borders}(x, \text{texas})$

Texas	:=	NP	:	texas
border	:=	$(S \backslash NP) / NP$	:	$\lambda y \lambda x. \text{borders}(x, y)$
states	:=	$S \backslash N$	:	$\lambda x. \text{state}(x)$
which	:=	$(S / (S \backslash NP)) / N$	:	$\lambda f, \lambda g, \lambda x. f(x) \wedge g(x)$

```
Prelude > let state a = (elem a ["nh","ma","vt"])
```

```
Prelude > state "nh"
```

```
=> True
```

```
Prelude > :type state
```

```
=> state :: [Char] -> Bool
```

---

<sup>1</sup> you can try out these examples using <https://tryhaskell.org/>

# A mini functional interpreter: Constants (Haskell)<sup>1</sup>

- **Example:** *What states border Texas?*

$\lambda x. \text{state}(x) \wedge \text{borders}(x, \text{texas})$

Texas	:=	NP	:	texas
border	:=	$(S \backslash NP) / NP$	:	$\lambda y \lambda x. \text{borders}(x, y)$
states	:=	$S \backslash N$	:	$\lambda x. \text{state}(x)$
which	:=	$(S / (S \backslash NP)) / N$	:	$\lambda f, \lambda g, \lambda x. f(x) \wedge g(x)$

*Prelude* > let **state** a = (elem a ["nh","ma","vt"])

*Prelude* > state "nh"

=> True

*Prelude* > :type state

=> state :: [Char] -> Bool

**semantic type:**  $e \longrightarrow t$

---

<sup>1</sup> you can try out these examples using <https://tryhaskell.org/>

# A mini functional interpreter: Constants (Haskell)

- **Example:** *What states border Texas?*

$$\lambda x. \text{state}(x) \wedge \text{borders}(x, \text{texas})$$

Texas	:=	NP	:	texas
border	:=	$(S \backslash NP) / NP$	:	$\lambda y \lambda x. \text{borders}(x, y)$
states	:=	$S \backslash N$	:	$\lambda x. \text{state}(x)$
which	:=	$(S / (S \backslash NP)) / N$	:	$\lambda f, \lambda g, \lambda x. f(x) \wedge g(x)$

```
Prelude > let borders :: ([Char], [Char]) -> Bool;
```

```
Prelude | borders a = (elem a [ "oklahoma", "texas" ])
```

# A mini functional interpreter: Constants (Haskell)

- **Example:** *What states border Texas?*

$$\lambda x. \text{state}(x) \wedge \text{borders}(x, \text{texas})$$

Texas	:=	NP	:	texas
border	:=	$(S \backslash NP) / NP$	:	$\lambda y \lambda x. \text{borders}(x, y)$
states	:=	$S \backslash N$	:	$\lambda x. \text{state}(x)$
which	:=	$(S / (S \backslash NP)) / N$	:	$\lambda f, \lambda g, \lambda x. f(x) \wedge g(x)$

```
Prelude > let borders :: ([Char], [Char]) -> Bool;
```

```
Prelude | borders a = (elem a [("oklahoma", "texas")])
```

```
Prelude > borders ("nh", "texas")
```

```
=> False
```



# CG and Binary Rules

- ▶ CGs typically assume binary rules/functions (or *curried functions*), so that function application applies to one argument at a time.
- ▶ **Example:** *What states border Texas?*

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

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

*Prelude* > :type borders

=> borders :: ([Char], [Char]) -> Bool

# CG and Binary Rules

- ▶ CGs typically assume binary rules/functions (or *curried functions*), so that function application applies to one argument at a time.
- ▶ **Example:** *What states border Texas?*

$$\lambda x. \text{state}(x) \wedge \text{borders}(x, \text{texas})$$

Texas	:=	NP	:	texas
border	:=	$(S \backslash NP) / NP$	:	$\lambda y \lambda x. \text{borders}(x, y)$
states	:=	$S \backslash N$	:	$\lambda x. \text{state}(x)$
which	:=	$(S / (S \backslash NP)) / N$	:	$\lambda f, \lambda g, \lambda x. f(x) \wedge g(x)$

```
Prelude > :type borders
```

```
=> borders :: ([Char], [Char]) -> Bool
```

```
Prelude > let borders_c = curry borders
```

# CG and Binary Rules

- ▶ CGs typically assume binary rules/functions (or *curried functions*), so that function application applies to one argument at a time.
- ▶ **Example:** *What states border Texas?*

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

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

```
Prelude > :type borders
```

```
=> borders :: ([Char], [Char]) -> Bool
```

```
Prelude > let borders_c = curry borders
```

```
Prelude > :type borders_c
```

```
=> borders_c :: [Char] -> [Char] -> Bool
```

# CG and Binary Rules

- ▶ CGs typically assume binary rules/functions (or *curried functions*), so that function application applies to one argument at a time.
- ▶ **Example:** *What states border Texas?*

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

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

```
Prelude > :type borders
```

```
=> borders :: ([Char], [Char]) -> Bool
```

```
Prelude > let borders_c = curry borders
```

```
Prelude > :type borders_c
```

```
=> borders_c :: [Char] -> [Char] -> Bool
```

**semantic type:**  $e \longrightarrow e \longrightarrow t$

# CG and Binary Rules

- ▶ CGs typically assume binary rules/functions (or *curried functions*), so that function application applies to one argument at a time.
- ▶ **Example:** *What states border Texas?*

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

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

```
Prelude > let borders_c = curry borders
```

```
Prelude > borders_c "oklahoma" "texas"
```

```
=> True
```

# CG and Binary Rules

- ▶ CGs typically assume binary rules/functions (or *curried functions*), so that function application applies to one argument at a time.
- ▶ **Example:** *What states border Texas?*

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

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

```
Prelude > let borders_c = curry borders
```

```
Prelude > borders_c "oklahoma" "texas"
```

```
=> True
```

```
Prelude > borders_c 10 "texas"
```

# CG and Binary Rules

- ▶ CGs typically assume binary rules/functions (or *curried functions*), so that function application applies to one argument at a time.
- ▶ **Example:** *What states border Texas?*

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

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

```
Prelude > let borders_c = curry borders
```

```
Prelude > borders_c "oklahoma" "texas"
```

```
=> True
```

```
Prelude > borders_c 10 "texas"
```

```
=> type error
```

# CG and Partial Function Application

$$\frac{\frac{\text{Oklahoma} \quad \frac{\text{borders} \quad \text{Texas}}{\text{NP : } \text{texas}'}}{(\text{S} \backslash \text{NP}) / \text{NP} : \lambda y, \lambda x. \text{borders}(x, y)}}{(\text{NP : } \text{oklahoma}' \quad \text{S} \backslash \text{NP} : \lambda x. \text{borders}(x, \text{texas}'))} \begin{matrix} (>) \\ (<) \end{matrix}$$

$\text{S} : \text{borders}(\text{oklahoma}', \text{texas}')$

```
Prelude > let borders_texas = borders_c "texas"
```

```
Prelude > :type borders_texas
```



# CG and Partial Function Application

$$\begin{array}{c} \text{Oklahoma} \quad \frac{\text{borders} \quad \text{Texas}}{(S \backslash NP) / NP : \lambda y, \lambda x. borders(x, y) \quad NP : \text{texas}'} \\ \hline NP : \text{oklahoma}' \quad S \backslash NP : \lambda x. borders(x, \text{texas}') \\ \hline S : borders(\text{oklahoma}', \text{texas}') \end{array} \begin{array}{l} (>) \\ (<) \end{array}$$

```
Prelude > let borders_texas = borders_c "texas"
```

```
Prelude > :type borders_texas
```

```
=> borders_texas :: [Char] -> Bool
```

# CG and Partial Function Application

$$\begin{array}{c} \text{Oklahoma} \quad \frac{\text{borders} \quad \text{Texas}}{\text{(S\NP)/NP : } \lambda y, \lambda x. \text{borders}(x, y) \quad \text{NP : } \text{texas}'} \\ \hline \text{NP : } \text{oklahoma}' \quad \text{S\NP : } \lambda x. \text{borders}(x, \text{texas}') \\ \hline \text{S : } \text{borders}(\text{oklahoma}', \text{texas}') \end{array} \begin{array}{l} (>) \\ (<) \end{array}$$

```
Prelude > let borders_texas = borders_c "texas"
```

```
Prelude > borders_texas "oklahoma"
```

```
=> True
```

# CG and Partial Function Application

	borders	Texas	
Oklahoma	$(S \backslash NP) / NP : \lambda y, \lambda x. borders(x, y)$	NP : texas'	
NP : oklahoma'	$S \backslash NP : \lambda x. borders(x, texas')$		(>)
$S : borders(oklahoma', texas')$			(<)

*Prelude* > right\_apply f x = (f x)

*Prelude* > right\_apply borders\_c "texas"

# Overall Model: Zettlemoyer and Collins (2012)

- ▶ **Compositional Semantic Model:** assumes the geo-query representations and semantics we've discussed. ✓
- ▶ **Probabilistic Model:** Deciding between different analyses, handling spurious ambiguity.
- ▶ **Lexical (Rule) Extraction:** Finding set of CCG lexical entries in  $\Lambda$ .

# Probabilistic CCG Model

- ▶ **Assumption:** Let's say (for now) we have a crude CCG lexicon  $\Lambda$  that over-generates for any given input
- ▶ **Derivation:** a pair,  $(L, T)$ , where  $L$  is the final logical form and  $T$  is the derivation tree.

**Example:** Oklahoma borders Texas

	borders	Texas	
	<hr/>		
Oklahoma	$(S \backslash NP) / NP : \lambda y, \lambda x. borders(x, y)$	NP : texas'	
<hr/>			(>)
NP : oklahoma'	$S \backslash NP : \lambda x. borders(x, texas')$		
<hr/>			(<)
S : borders(oklahoma', texas') ✓			

# Probabilistic CCG Model

- ▶ **Assumption:** Let's say (for now) we have a crude CCG lexicon  $\Lambda$  that over-generates for any given input
- ▶ **Derivation:** a pair,  $(L, T)$ , where  $L$  is the final logical form and  $T$  is the derivation tree.

**Example:** Oklahoma borders Texas

	borders	Texas	
Oklahoma	NP : texas' (S\NP)/NP : $\lambda y, \lambda x.borders(x, y)$		( < )
NP : oklahoma'	S\NP : $\lambda x.borders(x, texas')$		
			( < )
S : borders(oklahoma',texas') ✓			

# Probabilistic CCG Model

- ▶ **Assumption:** Let's say (for now) we have a crude CCG lexicon  $\Lambda$  that over-generates for any given input
- ▶ **Derivation:** a pair,  $(L, T)$ , where  $L$  is the final logical form and  $T$  is the derivation tree.

**Example:** Oklahoma borders Texas

	borders	Texas
Oklahoma	$(S \backslash NP) / NP : \lambda y, \lambda x. borders(x, y)$	$NP : texas'$
		(>)
$NP : ohio'$	$S \backslash NP : \lambda x. borders(x, texas')$	
		(<)
$S : borders(ohio', texas') \times$		

# Probabilistic CCG Model

- Use a log-linear formulation of CCG (Clark and Curran (2003)):

$$p(L, T \mid S; \theta) = \frac{e^{f(L, T, S) \cdot \theta}}{\sum_{(L, T)} e^{f(L, T, S) \cdot \theta}}$$

- **Parsing Problem:**

$$\operatorname{argmax}_L P(L \mid S; \theta) = \sum_T p(L, T \mid S; \theta)$$



# Probabilistic CCG Model

- ▶ Use a log-linear formulation of CCG (Clark and Curran (2003)):

$$p(L, T \mid S; \theta) = \frac{e^{f(L, T, S) \cdot \theta}}{\sum_{(L, T)} e^{f(L, T, S) \cdot \theta}}$$

- ▶ **Parsing Problem:**

$$\operatorname{argmax}_L P(L \mid S; \theta) = \sum_T p(L, T \mid S; \theta)$$

- ▶ **Note:** T might be very large, use dynamic programming.

# Probabilistic CCG Model

- ▶ Use a log-linear formulation of CCG (Clark and Curran (2003)):

$$p(L, T \mid S; \theta) = \frac{e^{f(L, T, S) \cdot \theta}}{\sum_{(L, T)} e^{f(L, T, S) \cdot \theta}}$$

- ▶ **Parsing Problem:**

$$\operatorname{argmax}_L P(L \mid S; \theta) = \sum_T p(L, T \mid S; \theta)$$

- ▶ **Note:** T might be very large, use dynamic programming.
- ▶ **Learning Setting:** The correct derivations are not annotated (latent), no further supervision is provided (weak). Lexicon is learned from data.

# Log-linear Model: Basics

## ► Log-Linear Model: <sup>2</sup>

- A set  $\mathcal{X}$  of inputs (e.g. sentences)
- A set  $\mathcal{Y}$  of labels/structures.
- A feature function  $f : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}^d$  for any pair  $(x,y)$
- A weight vector  $\theta$

## ► Conditional Model: for $x \in \mathcal{X}, y \in \mathcal{Y}$

$$p(y \mid x; \theta) = \frac{e^{f(x,y) \cdot \theta}}{Z(x, \theta)}$$

---

<sup>2</sup>Examples and ideas from Michael Collin's and Charles Elkan's tutorials (see syllabus).

# Log-linear Model: Basics

- **Conditional Model:** for  $x \in \mathcal{X}, y \in \mathcal{Y}$

$$p(y \mid x; \theta) = \frac{e^{f(x,y) \cdot \theta}}{Z(x, \theta)}$$

- $e^x$ : or exponential function  $\exp(x)$  (keeps scores positive)
- **inner product:** (sum of features  $f_j$  times feature weights  $\theta_j$ )

$$f(x, y) \cdot \theta = \sum_{k=1}^d \theta_k f_k(x, y)$$

- **normalization term** or partition function:

$$Z(x, \theta) = \sum_{y' \in \mathcal{Y}} e^{f(x, y') \cdot \theta}$$

# Structured Classification and Features

- ▶ **Structured classification:** We assume that labels in  $\mathcal{Y}$  has a rich internal structure (e.g., parse trees, pos tag sequences).
- ▶ **Individual feature functions:**

$$f_i(x, y) \rightarrow \mathbb{R} \text{ for } i = 1, \dots, d$$

# Structured Classification and Features

- ▶ **Structured classification:** We assume that labels in  $\mathcal{Y}$  has a rich internal structure (e.g., parse trees, pos tag sequences).
- ▶ **Individual feature functions:**

$$f_i(x, y) \rightarrow \mathbb{R} \text{ for } i = 1, \dots, d$$

- ▶ In the general case for log-linear models, there is no restriction on the types of features you can define.

# Structured Classification and Features

- ▶ **Structured classification:** We assume that labels in  $\mathcal{Y}$  has a rich internal structure (e.g., parse trees, pos tag sequences).
- ▶ **Individual feature functions:**

$$f_i(x, y) \rightarrow \mathbb{R} \text{ for } i = 1, \dots, d$$

- ▶ In the general case for log-linear models, there is no restriction on the types of features you can define.
- ▶ **Feature Templates:** features are not usually specified individually, but in terms of more general classes:

# Features: Part of speech tagging

- **Goal:** Assign to a given input word  $w_j$  in a sentence a part-of-speech tag given a set of tags  $\mathcal{Y}=\{N, ADJ, PN, \dots\}$

**Example:** The<sub>D</sub> dog<sub>N</sub> sleeps



# Features: Part of speech tagging

- **Goal:** Assign to a given input word  $w_j$  in a sentences a part-of-speech tag given a set of tags  $\mathcal{Y}=\{N, \text{ADJ}, \text{PN}, \dots\}$

**Example:** The<sub>D</sub> dog<sub>N</sub> sleeps

$$f_{id(w_j \wedge y)}(x, y) = \begin{cases} 1 & \text{each word } w_j \in x \text{ has tag } y \\ 0 & \text{otherwise} \end{cases}$$

$$f_{id(w_{j-1} \wedge w_j \wedge y)}(x, y) = \begin{cases} 1 & \text{each word } w_j \text{ and } w_{j-1} \text{ have tag } y \\ 0 & \text{otherwise} \end{cases}$$

# Features: Part of speech tagging

- **Goal:** Assign to a given input word  $w_j$  in a sentences a part-of-speech tag given a set of tags  $\mathcal{Y}=\{N, ADJ, PN, \dots\}$

**Example:** The<sub>D</sub> dog<sub>N</sub> sleeps

$$f_{id(w_j \wedge y)}(x, y) = \begin{cases} 1 & \text{each word } w_j \in x \text{ has tag } y \\ 0 & \text{otherwise} \end{cases}$$

$$f_{id(w_{j-1} \wedge w_j \wedge y)}(x, y) = \begin{cases} 1 & \text{each word } w_j \text{ and } w_{j-1} \text{ have tag } y \\ 0 & \text{otherwise} \end{cases}$$

$$f_{id(mother\_feature)}(x, y) = \begin{cases} 100 & \text{my mother likes } y \text{ tags} \\ 0 & \text{otherwise} \end{cases}$$

# Features: Part of speech tagging

- **Goal:** Assign to a given input word  $w_j$  in a sentences a part-of-speech tag given a set of tags  $\mathcal{Y}=\{N, ADJ, PN, \dots\}$

**Example:** The<sub>D</sub> dog<sub>N</sub> sleeps

$$f_{id(w_j \wedge y)}(x, y) = \begin{cases} 1 & \text{each word } w_j \in x \text{ has tag } y \\ 0 & \text{otherwise} \end{cases}$$

$$f_{id(w_{j-1} \wedge w_j \wedge y)}(x, y) = \begin{cases} 1 & \text{each word } w_j \text{ and } w_{j-1} \text{ have tag } y \\ 0 & \text{otherwise} \end{cases}$$

$$f_{id(mother\_feature)}(x, y) = \begin{cases} 100 & \text{my mother likes } y \text{ tags} \\ 0 & \text{otherwise} \end{cases}$$

"At the end of the day... the important factor is the features used" Domingos (2012)

# Local CCG features: Zettlemoyer and Collins (2012)

- ▶ **Output labels:**  $\mathcal{Y}$  is the set of (structured) CCG derivations.
- ▶ **Local features:** Limit features to lexical rules in derivations

# Local CCG features: Zettlemoyer and Collins (2012)

- **Output labels:**  $\mathcal{Y}$  is the set of (structured) CCG derivations.
- **Local features:** Limit features to lexical rules in derivations

$$\begin{array}{c} \text{Oklahoma} \quad \text{borders} \quad \text{Texas} \\ \hline \text{Oklahoma} \quad (S \backslash NP) / NP : \lambda y, \lambda x. borders(x, y) \quad NP : \text{texas}' \\ \hline NP : \text{oklahoma}' \quad S \backslash NP : \lambda x. borders(x, \text{texas}') \quad (>) \\ \hline S : borders(\text{oklahoma}', \text{texas}') \quad (<) \end{array}$$

# Local CCG features: Zettlemoyer and Collins (2012)

- **Output labels:**  $\mathcal{Y}$  is the set of (structured) CCG derivations.
- **Local features:** Limit features to lexical rules in derivations

$$\frac{\frac{\text{Oklahoma} \quad \text{borders} \quad \text{Texas}}{\text{(S} \backslash \text{NP) / NP : } \lambda y, \lambda x. \text{borders}(x, y) \quad \text{NP : texas'}}}{\text{NP : oklahoma'} \quad \text{S} \backslash \text{NP : } \lambda x. \text{borders}(x, \text{texas'})} \quad (>)$$
$$\frac{}{\text{S : borders(oklahoma', texas')} \quad (<)$$

$$f_{id(\text{NP : texas'})}(x, y) = \text{count}(\text{NP : texas'}) = 1$$

# Local versus non-local features

- ▶ **Why only local?:** Can be efficiently and easy extracted using our normal parsing algorithms and dynamic programming.
- ▶ Chart data-structure (e.g., in CKY) is a flat structure of cell entries.

	borders	Texas	
	_____	_____	
Oklahoma	$(S \backslash NP) / NP : \lambda y, \lambda x. borders(x, y)$	NP : texas'	
_____	_____	_____	(>)
NP : oklahoma'	$S \backslash NP : \lambda x. borders(x, texas')$		
_____	_____		(<)
	S : borders(oklahoma', texas')		

$$f_j(x, y) = \langle \text{"Oklahoma"}, \text{"borders"}, \text{NP : texas'} \rangle$$

# Local versus non-local features

- ▶ **Why only local?:** Can be efficiently and easily extracted using our normal parsing algorithms and dynamic programming.
- ▶ Chart data-structure (e.g., in CKY) is a flat structure of cell entries.
- ▶ **Non-local features:** getting around these issues.
  - ▶ **k-best parsing:** train a model on k-best trees (more on this later).
  - ▶ **forest-reranking:** Huang (2008).



# Parameter Estimation in Log-Linear Models: Basics

- ▶ **Learning task:** choose values for feature weights that solve some objective.

**Training Data:**  $D = \{(x_i, y_i)\}_{i=1}^n$

- ▶ **Maximum Likelihood:** find a model  $\theta^*$  that maximizes the probability of training data (or logarithm of conditional likelihood (LCL)):

$$\begin{aligned}\theta^* &= \max_{\theta} \prod_{i=1}^n p(y_i \mid x_i; \theta) \\ &= \max_{\theta} \sum_{i=1}^n \log p(y_i \mid x_i; \theta)\end{aligned}$$

# Optimization and Gradient methods

- ▶ **Optimization:** method for solving your objective.
  - ▶ **Intuitively:** assigning numbers to feature weights:  $\theta_1, \dots, \theta_d \in \theta^d$

# Optimization and Gradient methods

- ▶ **Optimization:** method for solving your objective.
  - ▶ **Intuitively:** assigning numbers to feature weights:  $\theta_1, \dots, \theta_d \in \theta^d$
- ▶ **Gradient-based optimization:** Uses a tool from calculus, the *gradient*

# Optimization and Gradient methods

- ▶ **Optimization:** method for solving your objective.
  - ▶ **Intuitively:** assigning numbers to feature weights:  $\theta_1, \dots, \theta_d \in \theta^d$
- ▶ **Gradient-based optimization:** Uses a tool from calculus, the *gradient*
  - ▶ **Gradient:** A type of derivative, or measure of the rate that a function changes
  - ▶ Tells the direction to move in order to get closer to objective.

# Gradient Methods: Intuitive explanation <sup>3</sup>

*Imagine your hobby is blindfolded mountain climbing, i.e., someone blindfolds you and puts you on the side of a mountain. Your goal is to reach the peak of the mountain by feeling things out, e.g., moving in the direction that feels upward until you reach the peak.*

*If mountain is concave, you will eventually reach your goal.*

---

<sup>3</sup>Example taken from Hal Daume III's book: <http://ciml.info/>

# Gradient Methods: Intuitive explanation <sup>3</sup>

*Imagine your hobby is blindfolded mountain climbing, i.e., someone blindfolds you and puts you on the side of a mountain. Your goal is to reach the peak of the mountain by feeling things out, e.g., moving in the direction that feels upward until you reach the peak.*

*If mountain is concave, you will eventually reach your goal.*

- **Objective:** Reach the maximum point (the peak) of the mountain.

---

<sup>3</sup>Example taken from Hal Daume III's book: <http://ciml.info/>

## Gradient Methods: Intuitive explanation <sup>3</sup>

*Imagine your hobby is blindfolded mountain climbing, i.e., someone blindfolds you and puts you on the side of a mountain. Your goal is to reach the peak of the mountain by feeling things out, e.g., moving in the direction that feels upward until you reach the peak.*

*If mountain is concave, you will eventually reach your goal.*

- ▶ **Objective:** Reach the maximum point (the peak) of the mountain.
- ▶ **Gradient:** The direction to move at each point to get closer to the point (or your objective).

---

<sup>3</sup>Example taken from Hal Daume III's book: <http://ciml.info/>

# Gradient Methods: Intuitive explanation <sup>3</sup>

*Imagine your hobby is blindfolded mountain climbing, i.e., someone blindfolds you and puts you on the side of a mountain. Your goal is to reach the peak of the mountain by feeling things out, e.g., moving in the direction that feels upward until you reach the peak.*

*If mountain is concave, you will eventually reach your goal.*

- ▶ **Objective:** Reach the maximum point (the peak) of the mountain.
- ▶ **Gradient:** The direction to move at each point to get closer to the point (or your objective).
- ▶ **Step Size:** How much to move at each step (parameter).

---

<sup>3</sup>Example taken from Hal Daume III's book: <http://ciml.info/>



# Gradient Ascent

- ▶ **Gradient ascent algorithm:** (abstract form)

- 1: Initialize  $\theta^d$  to 0
- 2: **While** not converged **do**
- 3:   Calculate  $\delta_k = \frac{\partial \mathcal{O}}{\partial \theta_k}$  for  $k = 1, \dots, d$
- 4:   Set each  $\theta_k = \theta_k + \alpha * \delta_k$
- 9: **Return**  $\theta^d$

- ▶ **Goal:** To find some maximum of a function.

- ▶ **Gradient  $\delta_k$ :** Direction to move each feature  $\theta_k$  to get closer to your objective  $\mathcal{O}$  (e.g., reaching the peak).

  - ▶ line 3: in batch case uses full dataset to estimate.

- ▶  $\alpha$ : learning rate or size of step to take (hyper-parameter).

# Gradient Ascent: Computing the Gradients

- ▶ **Dataset:** Assume that we have our dataset  $D = \{(x_i, y_i)\}_{i=1}^n$ , feature vector  $\theta^d$

- ▶ **LCL Objective:**

$$\mathcal{O}(\theta) = \sum_{i=1}^n \log p(y_i | x_i; \theta)$$

- ▶ **Computing Gradient** (using some calculus, full derivation not shown)

$$\frac{\partial}{\partial \theta_j} \log p(y | x; \theta) = \sum_{i=1}^n f_j(x_i, y_n) - \sum_{i=1}^n \sum_{y' \in \mathcal{Y}} p(y' | x_i; \theta) f_j(x_i, y')$$

# Gradient Ascent: Computing the Gradients

- ▶ **Dataset:** Assume that we have our dataset  $D = \{(x_i, y_i)\}_{i=1}^n$ , feature vector  $\theta^d$

- ▶ **LCL Objective:**

$$\mathcal{O}(\theta) = \sum_{i=1}^n \log p(y_i | x_i; \theta)$$

- ▶ **Computing Gradient** (using some calculus, full derivation not shown)

$$\frac{\partial}{\partial \theta_j} \log p(y | x; \theta) = \sum_{i=1}^n f_j(x_i, y_n) - \sum_{i=1}^n \sum_{y' \in \mathcal{Y}} p(y' | x_i; \theta) f_j(x_i, y')$$

- ▶ The main formula for computing line 3 (last page):
  - ▶ **Empirical counts:** The first half/summation above
  - ▶ **Expected counts:** The second half.

# Gradient Ascent: Computing the Gradients

- **Computing Gradient** (using some calculus, full derivation not shown)

$$\frac{\partial}{\partial \theta_j} \log p(y \mid x; \theta) = \sum_{i=1}^n f_j(x_i, y_n) - \sum_{i=1}^n \sum_{y' \in \mathcal{Y}} p(y \mid x_i; \theta) f_j(x_i, y')$$

- 1: Initialize  $\theta^d$  to 0
- 2: **While** not converged **do**
- 3:   Calculate  $\delta_k = \frac{\partial \mathcal{O}}{\partial \theta_k}$  for  $k = 1, \dots, d$
- 4:   Set each  $\theta_k = \theta_k + \alpha * \delta_k$
- 9: **Return**  $\theta^d$

- **Note:** Making updates (line 4) first requires first iterating over our full training training (line 3), is instance of *batch* learning.

# Gradients: Zettlemoyer and Collins (2012)

- ▶ Recall that a CCG derivation is a pair  $(L, T)$
- ▶ **LCL objective** (same objective, but a slightly different computation)

$$\begin{aligned}\mathcal{O}(\theta) &= \sum_{i=1}^n \log p(L_i \mid x_i; \theta) \\ &= \sum_{i=1}^n \log \left( \sum_T p(L_i, T \mid x_i; \theta) \right)\end{aligned}$$

# Gradients: Zettlemoyer and Collins (2012)

- ▶ Recall that a CCG derivation is a pair  $(L, T)$
- ▶ **LCL objective** (same objective, but a slightly different computation)

$$\begin{aligned}\mathcal{O}(\theta) &= \sum_{i=1}^n \log p(L_i \mid x_i; \theta) \\ &= \sum_{i=1}^n \log \left( \sum_T p(L_i, T \mid x_i; \theta) \right)\end{aligned}$$

- ▶ Computing Gradient:

$$\begin{aligned}\frac{\partial}{\partial \theta_j} \log p(y \mid x; \theta) \\ = \sum_{i=1}^n \sum_T f_j(L_i, T, x_i) p(T \mid L_i, x_i; \theta) - \sum_{i=1}^n \sum_{L, T} f_j(L, T, x_i) p(L, T \mid x_i; \theta)\end{aligned}$$

# Gradients: Zettlemoyer and Collins (2012)

- ▶ Computing Gradient:

$$\frac{\partial}{\partial \theta_j} \log p(L \mid x; \theta) \\ = \sum_{i=1}^n \sum_T f_j(L_i, T, x_i) p(T \mid L_i, x_i; \theta) - \sum_{i=1}^n \sum_{L, T} f_j(L, T, x_i) p(L, T \mid x_i; \theta)$$

- ▶ **Note:** This involves find the probability of all trees/derivations and their features given an input.

# Gradients: Zettlemoyer and Collins (2012)

- ▶ Computing Gradient:

$$\frac{\partial}{\partial \theta_j} \log p(L \mid x; \theta) \\ = \sum_{i=1}^n \sum_T f_j(L_i, T, x_i) p(T \mid L_i, x_i; \theta) - \sum_{i=1}^n \sum_{L, T} f_j(L, T, x_i) p(L, T \mid x_i; \theta)$$

- ▶ **Note:** This involves find the probability of all trees/derivations and their features given an input.
- ▶ **Dynamic programming:** Use variant of inside-outside probabilities covered in Lecture 3 (no big deal).



# Batch vs. Stochastic Gradient Descent

## ► Batch Gradient

$$\frac{\partial}{\partial \theta_j} \log p(y | x; \theta) = \sum_{i=1}^n f_j(x_i, y_n) - \sum_{i=1}^n \sum_{y' \in \mathcal{Y}} p(y | x_i; \theta) f_j(x_i, y')$$

- 1: Initialize  $\theta^d$  to 0
- 2: **While** not converged **do**
- 3:   Calculate  $\delta_k = \frac{\partial \mathcal{O}}{\partial \theta_k}$  for  $k = 1, \dots, d$
- 4:   Set each  $\theta_k = \theta_k + \alpha * \delta_k$
- 9: **Return**  $\theta^d$

## ► Stochastic Gradient

$$\frac{\partial}{\partial \theta_j} \log p(y | x; \theta) = f_j(x_i, y_n) - \sum_{y' \in \mathcal{Y}} p(y | x_i; \theta) f_j(x_i, y')$$

# Batch vs. Stochastic Gradient Descent

## ► Batch Gradient

$$\frac{\partial}{\partial \theta_j} \log p(y | x; \theta) = \sum_{i=1}^n f_j(x_i, y_n) - \sum_{i=1}^n \sum_{y' \in \mathcal{Y}} p(y | x_i; \theta) f_j(x_i, y')$$

- 1: Initialize  $\theta^d$  to 0
- 2: **While** not converged **do**
- 3:   Calculate  $\delta_k = \frac{\partial \mathcal{O}}{\partial \theta_k}$  for  $k = 1, \dots, d$
- 4:   Set each  $\theta_k = \theta_k + \alpha * \delta_k$
- 9: **Return**  $\theta^d$

## ► Stochastic Gradient

$$\frac{\partial}{\partial \theta_j} \log p(y | x; \theta) = f_j(x_i, y_n) - \sum_{y' \in \mathcal{Y}} p(y | x_i; \theta) f_j(x_i, y')$$

- **Online learning:** Updates are made at each example.

# Stochastic Gradient Ascent: Full

- ▶ **Dataset:** Assume that we have our dataset  $D = \{(x_i, y_i)\}_{i=1}^n$ , feature vector  $\theta^d$ 
  - 1: Initialize  $\theta^d$  to 0
  - 2: **While** not converged **do**
  - 3:   **Repeat for**  $i = 1, \dots, n$
  - 4:        $\theta_k = \theta_k + \alpha \times (f_k(x_i, y_i) - \sum_{y' \in \mathcal{Y}} p(y' \mid x_i; \theta) f_k(x_i, y'))$
  - 9: **Return**  $\theta^d$

# Stochastic Gradient Ascent: Full

- **Dataset:** Assume that we have our dataset  $D = \{(x_i, y_i)\}_{i=1}^n$ , feature vector  $\theta^d$

1: Initialize  $\theta^d$  to 0

2: **While** not converged **do**

3:     **Repeat for**  $i = 1, \dots, n$

4:          $\theta_k = \theta_k + \alpha \times (f_k(x_i, y_i) - \sum_{y' \in \mathcal{Y}} p(y' \mid x_i; \theta) f_k(x_i, y'))$

9: **Return**  $\theta^d$

- **line 3:** Start iterating through dataset
- **line 4:** Update at each example for LCL objective

# Stochastic Gradient Ascent: Full

- ▶ **Dataset:** Assume that we have our dataset  $D = \{(x_i, y_i)\}_{i=1}^n$ , feature vector  $\theta^d$ 
  - 1: Initialize  $\theta^d$  to 0
  - 2: **While** not converged **do**
  - 3:   **Repeat for**  $i = 1, \dots, n$
  - 4:      $\theta_k = \theta_k + \alpha \times (f_k(x_i, y_i) - \sum_{y' \in \mathcal{Y}} p(y' \mid x_i; \theta) f_k(x_i, y))$
  - 9: **Return**  $\theta^d$
- ▶ **line 3:** Start iterating through dataset
- ▶ **line 4:** Update at each example for LCL objective
- ▶ The simplest form, *vanilla gradient ascent*.

# Overall Model: Zettlemoyer and Collins (2012)

- ▶ **Compositional Semantic Model:** assumes the geo-query representations and semantics we've discussed. ✓
- ▶ **Probabilistic Model:** Deciding between different analyses, handling spurious ambiguity. ✓
- ▶ **Lexical (Rule) Extraction:** Finding set of CCG lexical entries in  $\Lambda$ .

# GENLEX: Lexical rule extraction

- ▶ So far we have assumed the existence of a CCG lexical  $\Lambda$
- ▶ GENLEX: Take a sentence and logical form and generates lexical items.

$$\text{GENLEX}(S, L) = \{x := y \mid x \in W(S), y \in C(L)\}$$

- ▶ **W(S)**: set of substrings in input  $S$
- ▶ **C(L)**: CCG rule templates or triggers

# Lexical rule templates (Triggers)

Rules		Categories produced from logical form $\arg \max(\lambda x.state(x) \wedge borders(x, texas), \lambda x.size(x))$
Input Trigger	Output Category	
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 $f$	$S / NP : \lambda x.f(x)$	$S / NP : \lambda x.size(x)$

- ▶ Templates specify patterns in logical forms (input triggers) and their mapping to CCG lexical entries (output category).
- ▶ Are hand-engineered (down side), which has been subsequently improved on in Kwiatkowski et al. (2010)



# Lexical rule templates: Example

**Example:** *Oklahoma borders Texas.*

*borders(oklahoma', texas')*

# Lexical rule templates: Example

**Example:** *Oklahoma borders Texas.*

*borders(oklahoma', texas')*

- ▶  $W(\text{Oklahoma borders Texas}) =$   
 $\{ \text{"Oklahoma"}, \text{"Texas"}, \text{"Oklahoma borders"}, \dots \}$

# Lexical rule templates: Example

**Example:** *Oklahoma borders Texas.*

*borders(oklahoma', texas')*

- ▶  $W(\text{Oklahoma borders Texas}) = \{ \text{"Oklahoma"}, \text{"Texas"}, \text{"Oklahoma borders"}, \dots \}$
- ▶  $C(\text{borders(oklahoma', texas')}) = \{ \text{borders}(\dots) \rightarrow (S \setminus NP) / NP : \lambda y, \lambda x. \text{borders}(x, y); \text{texas}' \rightarrow NP : \text{texas}', \dots \}$
- ▶ GENLEX: takes the combination of these.

# Synthesis: Lexical Learning + Parameter Estimation

- For  $t = 1 \dots T$

## Step 1: (Lexical generation)

- For  $i = 1 \dots n$ :
  - Set  $\lambda = \Lambda_0 \cup \text{GENLEX}(S_i, L_i)$ .
  - Calculate  $\pi = \text{PARSE}(S_i, L_i, \lambda, \bar{\theta}^{t-1})$ .
  - Define  $\lambda_i$  to be the set of lexical entries in  $\pi$ .
- Set  $\Lambda_t = \Lambda_0 \cup \bigcup_{i=1}^n \lambda_i$

## Step 2: (Parameter Estimation)

- Set  $\bar{\theta}^t = \text{ESTIMATE}(\Lambda_t, E, \bar{\theta}^{t-1})$

**Output:** Lexicon  $\Lambda_T$  together with parameters  $\bar{\theta}^T$ .

- ▶ **Learning:** Complete learning algorithm involves joining lexical learning with log-linear parameter estimation (via stochastic gradient ascent)
- ▶ **Big Idea:** Learn compact lexicons via greedy iterative method that works with high probability rules/derivations.

# Synthesis: Lexical Learning + Parameter Estimation

- For  $t = 1 \dots T$

## Step 1: (Lexical generation)

- For  $i = 1 \dots n$ :
  - Set  $\lambda = \Lambda_0 \cup \text{GENLEX}(S_i, L_i)$ .
  - Calculate  $\pi = \text{PARSE}(S_i, L_i, \lambda, \bar{\theta}^{t-1})$ .
  - Define  $\lambda_i$  to be the set of lexical entries in  $\pi$ .
- Set  $\Lambda_t = \Lambda_0 \cup \bigcup_{i=1}^n \lambda_i$

## Step 2: (Parameter Estimation)

- Set  $\bar{\theta}^t = \text{ESTIMATE}(\Lambda_t, E, \bar{\theta}^{t-1})$

**Output:** Lexicon  $\Lambda_T$  together with parameters  $\bar{\theta}^T$ .

- ▶ **Step 1:** Search for small set of lexical entries to parse data, then parse and find most probable rules.
- ▶ **Step 2:** Re-estimate log-linear model based on these *compact* lexical entries.

# Results (brief)

	Geo880		Jobs640	
	P	R	P	R
Our Method	96.25	79.29	97.36	79.29
COCKTAIL	89.92	79.40	93.25	79.84

- ▶ Two benchmark datasets (still being used today).
- ▶ Highest results reported at the time of publishing.
- ▶ Quite impressive increases in precision (though not so impressive recall).

# Conclusions and Take-aways

- ▶ Introduced (C)CG, a new formalism for semantic parsing.
  - ▶ **Lexicalism:** lexical entries describe combination rules.
  - ▶ Nice formalism for jointly modeling syntax-semantics.

# Conclusions and Take-aways

- ▶ Introduced (C)CG, a new formalism for semantic parsing.
  - ▶ **Lexicalism:** lexical entries describe combination rules.
  - ▶ Nice formalism for jointly modeling syntax-semantics.
- ▶ Log-linear CCG model for parsing from Zettlemoyer and Collins (2012)
  - ▶ **Log-linear models:** In particular, conditional log-linear model.
  - ▶ **Gradient methods:** gradient-based optimization and (stochastic) gradient ascent



# Roadmap

- ▶ **Next session:** start of student presentations!
  - ▶ 30 minutes each, plus 10-15 for questions.
  - ▶ **Due data:** slides (or draft slides) must be submitted one week advance for approval.
  - ▶ **Questions:** I will submit specific question that I expect you to address in your talk (not exam questions, only meant to help).
- ▶ **Schedule update:** I will give another lecture on the final class session (another opportunity for writing a reading summary).

# References I

- Clark, S. and Curran, J. R. (2003). Log-linear models for wide-coverage ccg parsing. In *Proceedings of the 2003 conference on Empirical methods in natural language processing*, pages 97–104. Association for Computational Linguistics.
- Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87.
- Huang, L. (2008). Forest reranking: Discriminative parsing with non-local features. In *ACL*, pages 586–594.
- Kwiatkowski, T., Zettlemoyer, L., Goldwater, S., and Steedman, M. (2010). Inducing probabilistic ccg grammars from logical form with higher-order unification. In *Proceedings of the 2010 conference on empirical methods in natural language processing*, pages 1223–1233. Association for Computational Linguistics.  
<http://www.aclweb.org/anthology/D/D10/D10-1119.pdf>.
- Steedman, M. (2000). *The syntactic process*, volume 24. MIT Press.
- Woods, W. A. (1973). Progress in natural language understanding: an application to lunar geology. In *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition*, pages 441–450.
- Zettlemoyer, L. S. and Collins, M. (2012). Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *arXiv preprint arXiv:1207.1420*. <http://arxiv.org/abs/1207.1420>.