# Natural Language Processing

Lecture 15: Semantic Parsing - Formal Semantics and CCG

11/13/2018

COMS W4705
Daniel Bauer

# Semantic Parsing

- Goal : Convert raw input text into some meaning representation.
- vs. Semantic Role Labeling (SRL):
  SRL only considers individual predicate-argument structures.
- Today we will consider meaning representations for complete sentences.

# Semantic Parsing Questions

- What should the target representations be?
- Can we compute sentence meaning from lexical meaning?
- What is the role of syntax?
- What algorithms and resources (training data etc.) do we need for semantic parsing?

# Goals for Meaning Representations

- Should be **unambiguous** (resolve any ambiguity [syntactic ambiguity, word-sense etc.] present in the natural language utterance).

- **Canonical form**: Different sentences that have the same meaning should have the same canonical meaning representation.

- **Support inference:** Want to use meaning representation to draw logical conclusions.

- **Expressiveness:** Want to capture a wide range of semantic phenomena and subject matter.

# First-Order Logic (FOL)

- Traditional approach to meaning representation ("Logical Form")

- Formal properties are well understood. Efficient algorithms for inference.

- First-order logic is written in one particular formal language, can describe this language using a CFG.

- Components of a FOL formula: **Constants, Functions, Terms, Quantifiers, Variables, Logical Connectives**.

- Model-theoretic semantics: A formula is "true" in a particular set of models (truth conditions).

# Terms

- Constants refer to exactly one specific entities in the world.

*ColumbiaUniversity, Alice*

- Functions map a constant to a specific other constant.

*Age(Alice), LocationOf(ColumbiaUniversity), Age(MotherOf(Alice))*

- Variables are used to make generalizations over sets of objects in the world. (a term that does not contain a variable is called a "ground" term)

# Predicates

- Predicates are used to make statements over terms

  *Student(Alice)* — *Alice is a student.*
  *Attends(Alice, ColumbiaUniversity)* — *Alice goes to Columbia.*
  *Love(Mother(Alice), Alice)* — *Alice's mother loves her.*

- A predicate applied to a list of terms is an **atomic FOL-formula.**

# Logical Connectives

- Logical connectives ¬, ∧, ∨, →,are used to build formulas from atomic formulas.

*Student(Alice)* ∧ *Attends(Alice, ColumbiaUniversity)*
Alice is a student who attends Columbia

*¬Love(Mother(Alice), Alice)*
Alice's mother does not love her.

*Attends(Alice, ColumbiaUniversity)* → *Student(Alice)*
Alice attends Columbia, therefore she is a student.

*Attends(Alice, ColumbiaUniversity)* ∨ *Attends(Alice, CityCollege)*
Alice attends Columbia or CityCollege.

# Variables and Quantifiers

- Terms may contain variables. These are used in two ways: Existential quantification and universal quantification.

- Existentially quantified formulas assert that some entity exists for which the formula is true.

$$\exists x.\ Attends(x, ColumbiaUniversity)$$

*There is someone who attends Columbia.*

- Universally quantified formulas assert that the formula is true for all entities (in the world).

$$\forall x.\ Love(Mother(x), x)$$

*Everyone is loved by their mother.*

# Variables and Quantifiers

- Existential quantification often appears with conjuction.

$$\exists x\ Attends(x, ColumbiaUniversity) \land Likes(x, IceCream)$$

*Someone goes to Columbia who likes ice cream.*

- Universal quantification often appears with implication.

$$\forall x\ Attends(x, ColumbiaUniversity) \rightarrow Student(x)$$

*Everyone who attends Columbia is a student.*

# CFG for FOL formulae

Formula → AtomicFormula
| Formula Connective Formula
| Quantifier Variable . Formula
| ¬ Formula
| *(* Formula *)*

AtomicFormula → Predicate*(*Term, ...*)*

Term → Function*(*Term*)*
| Constant
| Variable

Term → Function(Term)
| Constant
| Variable

Connective → ∧ | ∨ | →

Quantifier → ∃ | ∀

Constant → *Alice* | *ColumbiaUniversity* |*...*

Variable → *x* | *y* | *z* |*...*

Predicate → *Attends* | *Loves* | *Student* |*...*

Function → *Mother* | *Age* | *...*

# Model Theoretic Semantics

- The meaning of a formula is its truth conditions.
- Truth conditions can be described as a set of "possible worlds" (models) that make the expression true.
  - Such a model must contain all entities referred to by the terms of the formula.
  - For example

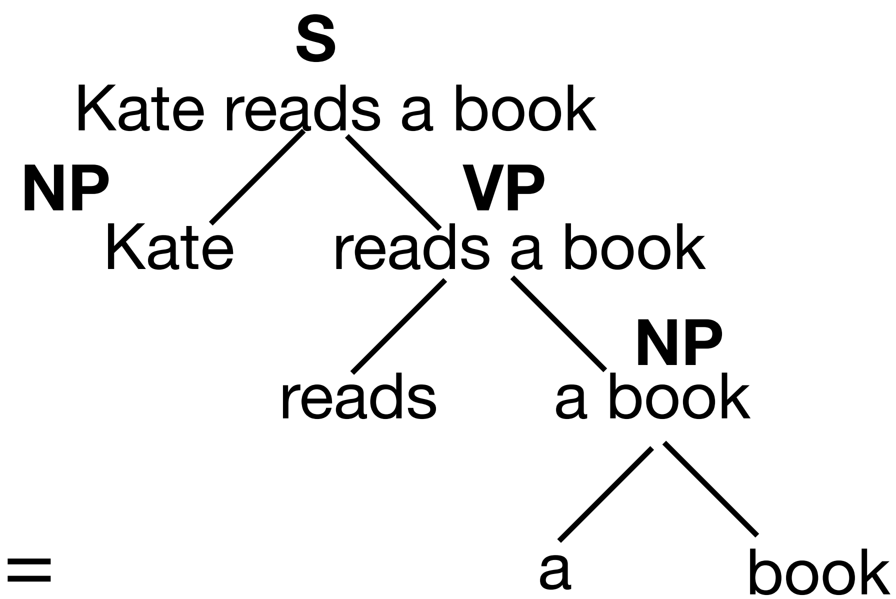
$$\forall x \; Attends(x, ColumbiaUniversity) \rightarrow Student(x)$$

is true in all worlds in which everyone who attends Columbia is a student.

# Principle of Compositionality

- The meaning of a complex expression should be completely determined by
  - the sub-parts of the expression
  - the rules used to combine these expressions.
- We should be able to compute sentence meaning from word meaning.

# Compositionality




$$[[\text{ Kate reads a book }]] =$$

$$C_1(\ [[\text{ Kate }]],\ [[\text{ reads a book }]]\ ) =$$

$$C_1(\ [[\text{ Kate }]],\ C_2(\ [[\text{ reads }]],\ [[\text{a book }]]\ )\ ) =$$

$$C_1(\ [[\text{ Kate }]],\ C_2(\ [[\text{ reads }]],\ C_3\ (\ [[\text{a}]],\ [[\text{book }]]\ )\ )\ )$$

# Semantic Analysis with First-Order Logic

**(Richard Montague, 1970s)**

- Basic approach: Use syntax to guide composition of first-order logical expressions.
- We need:
  - A representation for lexical entries (meaning associated with each word).
  - Rules to perform the composition.

# Lambda Expressions

- We need a way to compose FOL formulas from components.

- Lambda notation extends FOL to include expressions of the following form:

  $\lambda x.P(x)$

  where $x$ is a variable and P is some FOL formula containing $x$ as a *free variable* (not bound by a quantifier).

  (think of this a function with a single parameter x)

# Combining Lambda Expressions

- Lambda expressions can be applied to other expressions to make new expressions.

$$\frac{\lambda x.P(x)\ (A)}{P(A)}\ \text{beta-}\textit{reduction}$$

- The body of a lambda expression can be another lambda expression.

$$\frac{\dfrac{\lambda x.\lambda y.Likes(y,x) \quad (\text{Alice})}{\lambda y.Likes(y, Alice)} \qquad (\text{Bob})}{\text{Likes(Bob, Alice)}}$$

- Taking a predicate with multiple arguments and turning it into a sequence of single-argument predicates is called Currying.

# Higher-order Functions and Types

- Arguments to lambda expressions can be other lambda expressions (not just constants).

**S**
every student sleeps

**NP** every student — **VP** sleeps

every student

[[every student sleeps]] =

[[every student]] ([[sleeps]]) =

([[every]] ([[student]])) ([[sleeps]])

[[sleeps]] = $\lambda y.sleeps(y)$ **Type: <e**ntity**,t**ruth value**>**

[[student]] = $\lambda z.student(z)$ **Type: <e,t>**

[[every]] =$\lambda P \lambda Q. \forall x(P(x) \rightarrow Q(x))$ **Type: <<e, t>, <e, t>>, t>**

# Function Application Example

[[every student]] = [[every]] ([[student]])

$$\frac{\lambda P \lambda Q. \forall x\, P(x) \rightarrow Q(x) \quad (\lambda z.\mathrm{student}(z))}{\begin{array}{l}\lambda Q. \forall x\, (\lambda z.\mathrm{student}(z)\ (x)) \rightarrow Q(x) \\ \lambda Q. \forall x\, \mathrm{student}(x) \rightarrow Q(x)\end{array}}$$

[[sleeps]] = $\lambda y.\mathrm{sleeps}(y)$

[[student]] = $\lambda z.\mathrm{student}(z)$

[[every]] =$\lambda P \lambda Q. \forall x(P(x) \rightarrow Q(x))$

# Function Application Example

[[every student sleeps]] = [[every student]] ([[sleeps]])

$$\frac{\lambda Q.\forall x\ \text{student}(x) \rightarrow Q(x) \qquad (\lambda y.\text{sleep}(y))}{\forall x\ \text{student}(x) \rightarrow (\lambda y.\text{sleep}(y)\ (x))}$$

$\forall$**x student(x) $\rightarrow$ sleep(x)**

[[sleeps]] = $\lambda y.\text{sleeps}(y)$

[[student]] = $\lambda z.\text{student}(z)$

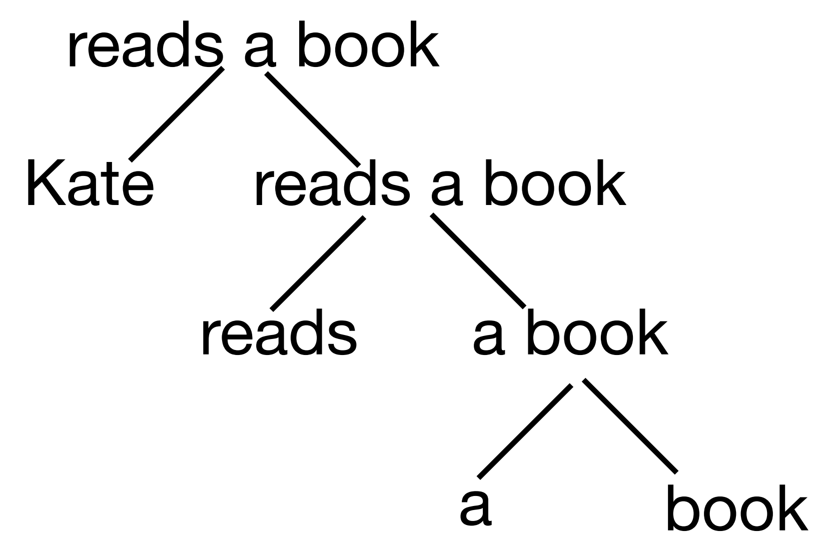
[[every]] = $\lambda P \lambda Q.\forall x(P(x) \rightarrow Q(x))$

# Another Example

[[Kate]] = *Kate*

[[reads]] = $\lambda x.\lambda y.\mathrm{Reads}(y,x)$

[[a book]] = $\lambda P.\lambda y.\ \exists x\ \mathrm{Book}(x) \wedge P(y,x)$




[[Kate reads a book]]

= ( [[a book]] ([[reads]]) ) ([[Kate]]) Apply "a book" to "reads"

$= \exists x\ Book(x) \wedge Reads(Kate,x)$

# Another Example

[[reads a book]] = [[a book]] ([[reads]])

$$\frac{\lambda P.\lambda y.\ \exists x\ \mathrm{Book}(x) \wedge P(y,x) \qquad (\lambda s.\lambda t.\mathrm{Reads}(s,t))}{\lambda y.\ \exists x\ \mathrm{Book}(x) \wedge \mathit{Reads}(y,x)}$$

[[Kate]] = *Kate*

[[reads]] = $\lambda x.\lambda y.\mathrm{Reads}(y,x)$

[[a book]] = $\lambda P.\lambda y.\ \exists x\ \mathrm{Book}(x) \wedge P(y,x)$

# Another Example

[[Kate reads a book]] = [[reads a book]] ([[Kate]])

$$\frac{\lambda y.\ \exists x\ \text{Book}(x) \wedge \mathit{Reads}(y,x) \quad (\text{Kate})}{\exists x\ \text{Book}(x) \wedge \mathit{Reads}(\text{Kate},x)}$$

[[Kate]] = *Kate*

[[reads]] = $\lambda x.\lambda y.\text{Reads}(y,x)$

[[a book]] = $\lambda P.\lambda y.\ \exists x\ \text{Book}(x) \wedge P(y,x)$

# Another Example

[[every student reads a book]] = [[every student]] ([[reads]])

$$\frac{\lambda P.\lambda y.\ \exists x\ \mathrm{Book}(x) \wedge P(y,x) \qquad (\lambda s.\lambda t.\mathrm{Reads}(s,t))}{\lambda y.\ \exists x\ \mathrm{Book}(x) \wedge \mathit{Reads}(y,x)}$$

[[Kate]] = *Kate*

[[reads]] = $\lambda x.\lambda y.\mathrm{Reads}(y,x)$

[[a book]] = $\lambda P.\lambda y.\ \exists x\ \mathrm{Book}(x) \wedge P(y,x)$

# Categorical Grammar

- An alternative approach to building phrase structure.
  - Phrases are associated with syntactic *categories* rather than non-terminal symbols.
  - Each lexicon entry is associated with a lexical category.
  - There is a small set of rules to guide combination of these categories in context.
- Inspired by lambda-calculus: Categories are higher-order functions applied to other categories.

**Ajdukiewicz (1935), Bar-Hillel (1953)**

# Categorical Grammar

- A category is either:
  - An atomic constituent symbol NP, S, ...
  - A single-argument function mapping a desired category to a category.
    (X / Y) - take category Y as an argument (to the right), return X
    (X \ Y) - take category Y as an argument (to the left), return Y.

Example Lexicon:

| | | |
|---|---|---|
| Mary | := | NP |
| musicals | := | NP |
| likes | := | ((S \ NP) / NP) |
| gives | := | (((S \ NP ) / NP) / NP) |

# Categorical Grammar: Rules and Derivations

- Forward function application: >

$$\frac{(X / Y) \qquad Y}{X} >$$

- Backward function application: <

$$\frac{Y \qquad (X \backslash Y)}{X} <$$

| | | |
|---|---|---|
| Mary | := | NP |
| musicals | := | NP |
| likes | := | ((S \ NP) / NP) |

$$\frac{\frac{\text{Mary}}{NP} \qquad \frac{\frac{\text{likes}}{((S \backslash NP) / NP)} \qquad \frac{\text{musicals}}{NP}}{(S \backslash NP)} >}{S} <$$

Observation: This is exactly like CFG so far!

# Categorical Grammar and Semantic Construction

Mary | likes | musicals
--- | --- | ---
NP : Mary | ((S \ NP) / NP): $\lambda x.\lambda y.Likes(y,x)$ | NP:Musicals

$$\dfrac{\text{likes: } ((S \backslash NP) / NP): \lambda x.\lambda y.Likes(y,x) \qquad \text{musicals: } NP:Musicals}{(S \backslash NP): \lambda y.Likes(y,Musicals)}>$$

$$\dfrac{\text{Mary: } NP : Mary \qquad (S \backslash NP): \lambda y.Likes(y,Musicals)}{S : Likes(Mary,Musicals)}<$$

| | | | |
|---|---|---|---|
| Mary | := | NP | : Mary |
| musicals | := | NP | : Musicals |
| likes | := | ((S \ NP) / NP) | : $\lambda x.\lambda y.Likes(y,x)$ |

# Semantic Construction and Categorial Grammars

| Every | student | sleeps |
|---|---|---|
| (NP / N) | N | (S \ NP) |
| NP | > | |
| S | | < |

Every := (NP / N)
student := N
sleeps := (S \ NP)

This corresponds to the following function applications:
sleeps (every (student))

# Semantic Construction and Categorial Grammars

| Every | student | sleeps |
|---|---|---|
| (NP / N) | N | (S \ NP) |

$$\dfrac{\dfrac{\text{Every} \quad \text{student}}{\text{NP}}\,> \qquad \text{sleeps}}{\text{S}}\,<$$

| | | | | |
|---|---|---|---|---|
| Every | := | (NP / N) | : | $\lambda P \lambda Q. \forall x(P(x) \rightarrow Q(x))$ |
| student | := | N | : | $\lambda z.\ student(z)$ |
| sleeps | := | (S \ NP) | : | $\lambda y.\ student(y)$ |

**! Problem, types don't match**

This corresponds to the following function applications:
sleeps (every (student))

Problem: to compute the logical form we need:
(every (student)) (sleeps)

# Semantic Construction and Categorial Grammars

| Every | student | sleeps |
|---|---|---|
| $((S / (S \backslash NP)) / N) : \lambda P \lambda Q. \forall x(P(x) \rightarrow Q(x))$ | $N : \lambda z.\ student(z)$ | $(S \backslash NP) : \lambda y.\ sleeps(y)$ |

$$\xrightarrow{\hspace{2em}>\hspace{2em}}\quad (S / (S \backslash NP)) : \lambda Q. \forall x(student(x) \rightarrow Q(x))$$

$$\xrightarrow{\hspace{2em}>\hspace{2em}}\quad S : \forall x(student(x) \rightarrow sleep(x))$$

| | | | |
|---|---|---|---|
| Every | := | $((S / (S \backslash NP)) / N)$ | : $\lambda P \lambda Q. \forall x(P(x) \rightarrow Q(x))$ |
| student | := | $N$ | : $\lambda z.\ student(z)$ |
| sleeps | := | $(S \backslash NP)$ | : $\lambda y.\ sleeps(y)$ |

We need to change the syntactic category for "every" (type raising)

# Combinatory Categorial Grammar (CCG)

- In addition to forward/backward application, we add a number of function *combinators.*

- Forward composition:

$$\frac{(X / Y) \quad (Y / Z)}{(X / Z)} >B$$

- Backward composition:

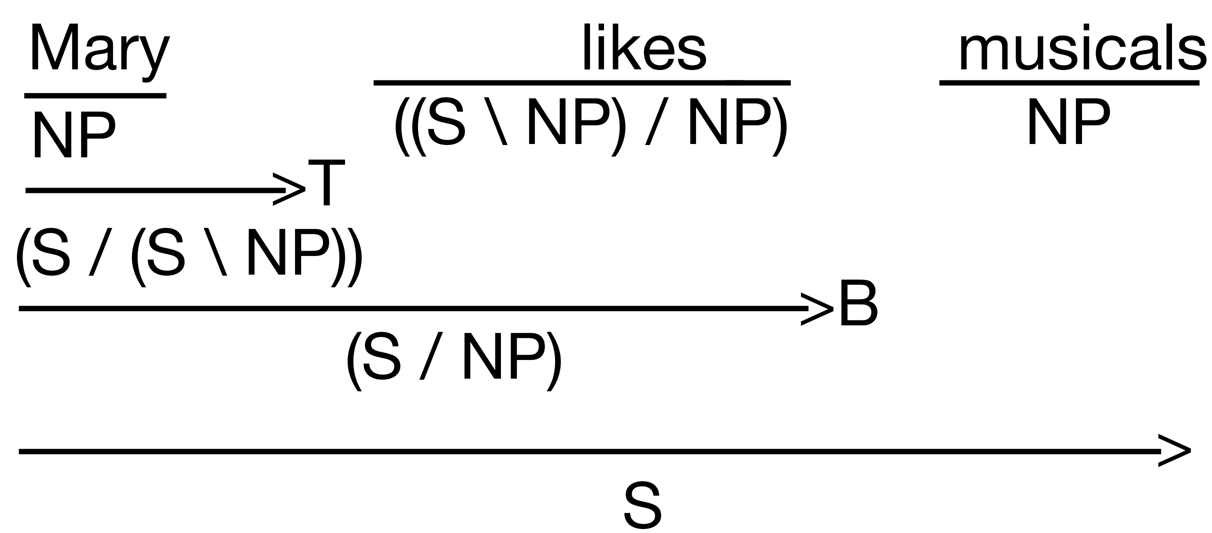
$$\frac{(Y \backslash Z) \quad (X \backslash Y)}{(X \backslash Z)} <B$$

- Type raising: $\dfrac{X}{(T / (T \backslash X))} >T$ or $\dfrac{X}{(T \backslash (T / X))} <T$

Steedman (1996)

# Type Raising Example

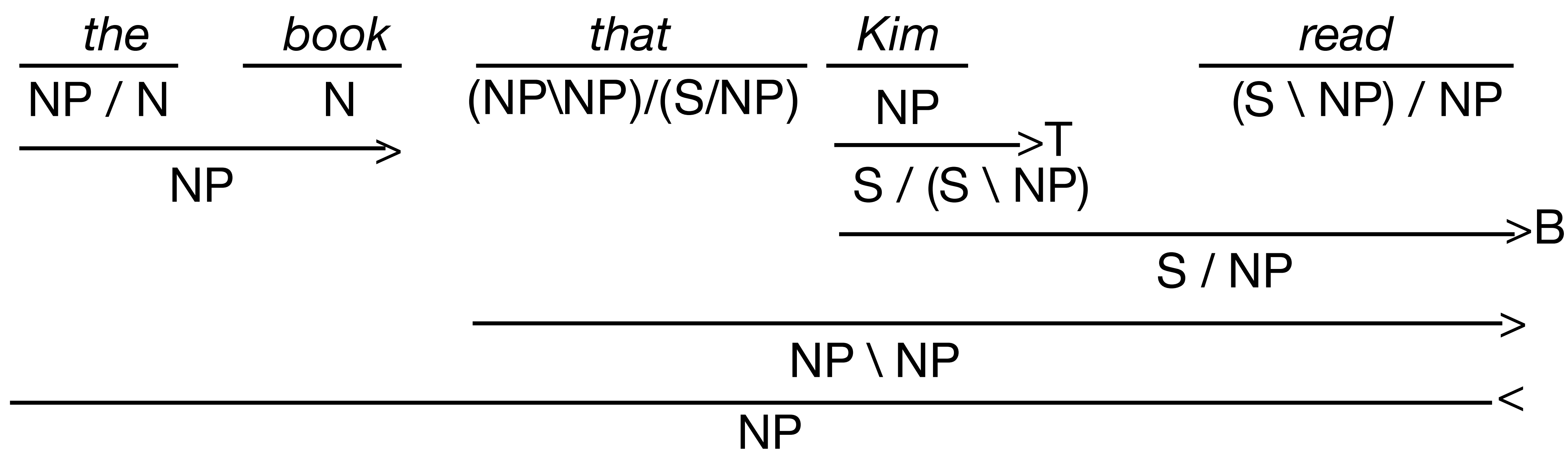
| | | |
|---|---|---|
| Mary | := | NP |
| musicals | := | NP |
| likes | := | ((S \ NP) / NP) |




Note:

- Can process input tokens left-to-right.
- The (S / NP) cartegory does not correspond to a traditional English constituent.

# Long-Distance Dependencies in CCG




Note:

- Grammar needs only one lexical entry for read!
- Type raising allows us to combine *Kim* with *read* before the object NP is attached. The missing NP is represented n the new category S / NP.

# CCG Observations

- CCG has become really popular. CCGBank (Hockenmaier & Steedman 2007) is an automatic conversion of the Penn Treebank to CCG.
- CCG generates the same class of string languages as TAG ("mildly context sensitive").
- Parsing is more expensive (can be done in $O(N^6)$).
  - Efficient greedy Algorithms exist (e.g. Lewis 2015)

# Compositional Revisited

- Is natural language really compositional?

| | |
|---|---|
| '*Sam is **taking** a **shower***' | light verbs |
| '*He **kicked** the **bucket***' | idioms |
| '*They **caught up** with them*' | particle verbs |