



Lecture 18: Logic III



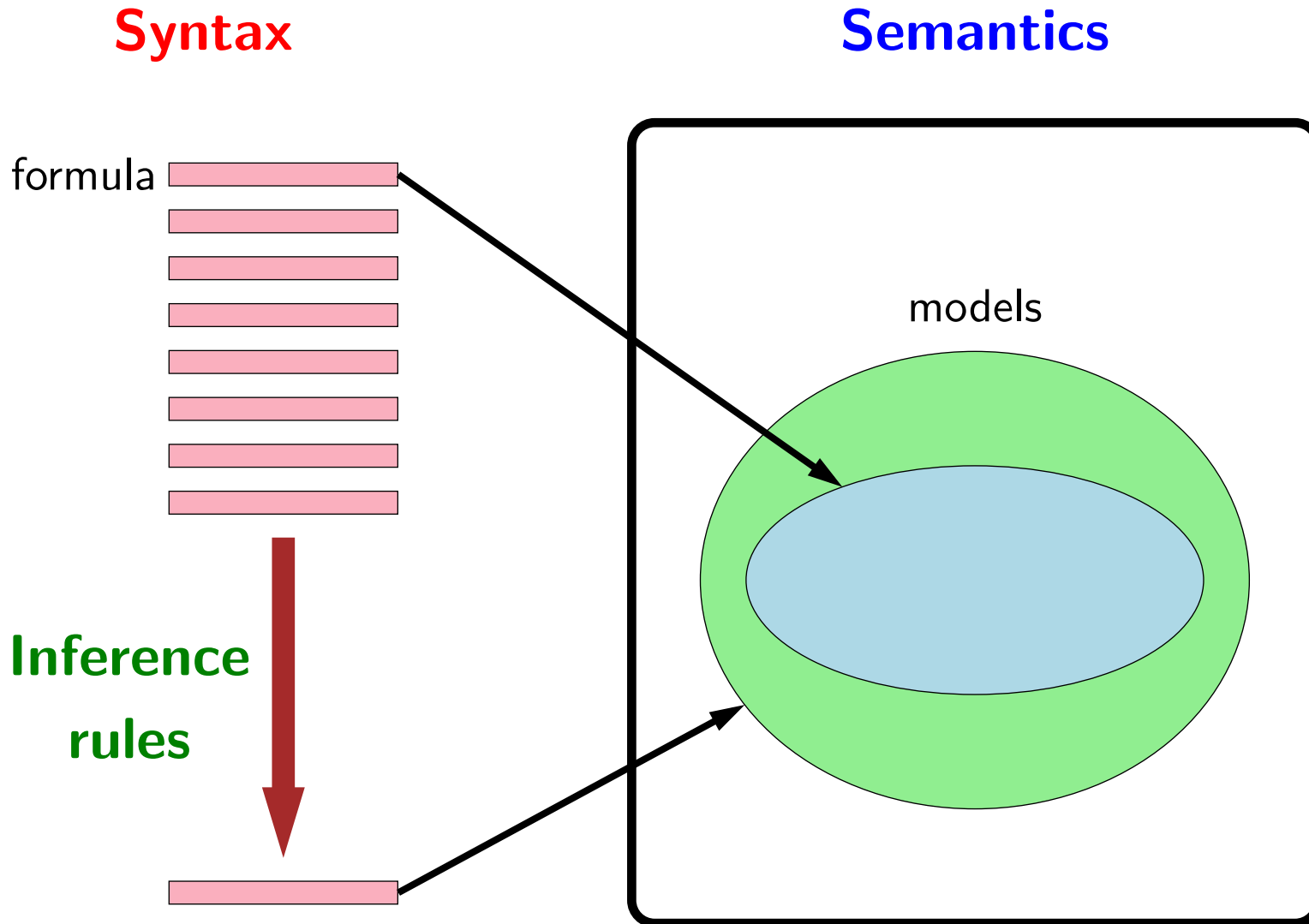


cs221.stanford.edu/q

Question

How would you write *Alice believes it will rain* in logic?

Review: schema





Roadmap

Other logics

Markov logic

Language to logic

Motivation

Goal of logic: represent knowledge and perform inferences

Propositional logic:

AliceIsStudent \rightarrow AliceKnowsArithmetic

BobIsStudent \rightarrow BobKnowsArithmetic

First-order logic:

$\forall x \text{ Student}(x) \rightarrow \text{Knows}(x, \text{arithmetic})$

- In the last two lectures, we have presented two types of logics: propositional logic and first-order logic.
- In propositional logic, we build formulas by putting connectives around propositional symbols. These allow us to make statements about the truth values of specific facts.
- In first-order logic, the key conceptual difference is that we are making statements about objects and their relations. Propositional symbols become atomic formulas which are predicates applied to terms (constants, variables, and functions). Together with quantifiers, this allows us to make compact statements that represent a huge (possibly infinite) set of objects.
- From a modeling point of view, first-order logic is more expressive than propositional logic, but it does come at a increase in computational cost.

Motivation

Why use anything beyond first-order logic?

Expressiveness:

- Temporal logic: express time
- Epistemic logic: express beliefs
- Lambda calculus: generalized quantifiers

Notational convenience, computational efficiency:

- Description logic

- There are two ways we can improve on first-order logic.
- The first makes modeling easier by increasing expressiveness, allowing us to "say more things", so we can talk about time, beliefs, not just objects and relations. We can also improve the modeler's life by making notation simpler.
- The second makes life easier for algorithms by decreasing expressiveness.

Temporal logic

Barack Obama is the US president.

President(BarackObama, US)

*George Washington **was** the US president.*

P President(GeorgeWashington, US)

*Some woman **will be** the US president.*

F $\exists x \text{ Female}(x) \wedge \text{President}(x, \text{US})$

- So far, we have talked about objects and relations as if they held universally over time. But things change. For example, the "president" relation depends on what year it is. Furthermore, we'd like to make statements about the past and the future without specifying the exact time.
- To do this, let's introduce an additional piece of syntax (e.g., **P** for past, **F** for future).

Temporal logic

Setup: all formulas interpreted at a current time.

$\mathcal{I}(f, w, t) = 1$ if f is true in w at time t

$\mathcal{I}(\mathbf{P}f, w, t) = 1$ if exists $s < t$ such that $\mathcal{I}(f, w, s) = 1$

The following operators change the current time and quantify over it:

P f : f held at some point in the past

F f : f will hold at some point in the future

H f : f held at every point in the past

G f : f will hold at every point in the future

Every student will at some point never be a student again.

$\forall x. \text{Student}(x) \rightarrow \mathbf{FG} \neg \text{Student}(x)$

- We now need to endow the new notation (e.g., \mathbf{P}) with semantics. Before, a formula f was interpreted with respect to a particular model w : $\mathcal{I}(f, w)$. But now, let's evaluate formulas at both a model w and a time t : $\mathcal{I}(f, t, w)$. Then the past operator simply looks for a past time and evaluates the formula f at each shifted time s (existential quantification over the past).
- Temporal logic consists four such temporal quantifiers, which correspond to existential or universal quantification over the past or future.
- We can use multiple temporal quantifiers to express rather complex statements with just a few symbols — highlighting the power of logic.

Modal logic for propositional attitudes

Alice believes one plus one is two.

Believes(alice, Equals(Sum(1, 1), 2))??

Alice believes Boston is a city.

Believes(alice, City(boston))??

Problem: Equals(Sum(1, 1), 2) is true, City(boston) is true, but the two are not interchangeable in this context.

- Another type of knowledge we'd like to encode are propositional attitudes (what people believe, know, want, etc.). For example, the relation Believes isn't simply taking objects as arguments, but rather entire propositions.
- However, writing down the logical forms naively doesn't work because if two propositions have the same truth value, then they would reduce to the same formula, but believing that one plus one is two surely isn't the same as believing that Boston is a city, despite the fact that both are true.

Epistemic logic

Alice believes Boston is a city.

$\mathbf{B}_{\text{alice}}$ City(boston)

Every place Alice has lived she believes is a city.

$\forall x \mathbf{P}\text{LivesIn}(\text{alice}, x) \rightarrow \mathbf{B}_{\text{alice}} \text{City}(x)$

Semantics:

- $\mathcal{F}_{\text{alice}}$ is set of internally-consistent formulas that Alice believes
- $\mathcal{I}(\mathbf{B}_{\text{alice}}f, w) = 1$ if $\mathcal{I}(f, w) = 1$ for all worlds w consistent with Alice's beliefs ($w \in \mathcal{M}(\mathcal{F}_{\text{alice}})$)

- We can fix this problem with epistemic logic, which allows us to represent beliefs and knowledge. The basic idea is that for each agent a (e.g., alice), we have a consisting of set of formulas \mathcal{F}_a (e.g., $\{\text{City}(\text{boston}), \text{City}(\text{california})\}$). Importantly, the formulas \mathcal{F}_a do not have to have any relationship with the real world, but they must be internally consistent ($\mathcal{M}(\mathcal{F}_a) \neq \emptyset$).
- We now introduce a new operator \mathbf{B}_a , which takes a formula f and checks if it is **entailed** by a 's beliefs \mathcal{F}_a ; that is, for every possible world that a might belief we're in, f holds.
- Note that a consequence of this formulation of belief is that agents have arbitrary inferential power and are fully rational: When an agent a believes \mathcal{F}_a , that agent must believe all the entailed consequences of a . See <http://plato.stanford.edu/entries/logic-epistemic/> for more information.
- Just like the past operator \mathbf{P} shifted interpretation of its argument f into a different time point, \mathbf{B}_a shifts interpretation to a different world.
- These operators can be embedded into first-order logic formulas with temporal operators to express rather intricate facts about the world.

Lambda calculus

Simple:

Alice has visited some museum.

$\exists x \text{Museum}(x) \wedge \text{Visited}(\text{alice}, x)$

More complex:

*Alice has visited **at least 10** museums.*

$\lambda x \text{Museum}(x) \wedge \text{Visited}(\text{alice}, x)$: boolean function representing **set** of museums Alice has visited

$\text{GreaterThan}(\text{Count}(\lambda x \text{Museum}(x) \wedge \text{Visited}(\text{alice}, x)), 10)$

- Existential and universal quantification only allow a formula to look at one value at a time. But sometimes, we need to look at the set of all values satisfying a formula, for example, if we wanted to count.
- Here, we will use lambda calculus to define **higher-order functions** that allow us to do precisely this.
- In lambda calculus, $\lambda x P(x)$ denotes the set of x for which $P(x)$ is true. This set can be passed in as an argument into the function Count, which produces a term.

Description logic

People with at least three sons who are married to doctors, and at most two daughters who are married to professors...are weird.

First-order logic + lambda calculus:

$$\begin{aligned} &\forall x \left(\text{Person}(x) \right. \\ &\quad \wedge \text{Count}(\lambda y \text{ Son}(x, y) \wedge \forall z \text{ Spouse}(y, z) \wedge \text{Doctor}(z)) \geq 3 \\ &\quad \left. \wedge \text{Count}(\lambda y \text{ Daughter}(x, y) \wedge \forall z \text{ Spouse}(y, z) \wedge \text{Professor}(z)) \leq 2 \right) \rightarrow \text{Weird}(x) \end{aligned}$$

Description logic:

$$\begin{aligned} &(\text{Person} \\ &\quad \sqcap (\geq 3 \text{ Son} . \forall \text{ Spouse} . \text{Doctor}) \\ &\quad \sqcap (\leq 2 \text{ Daughter} . \forall \text{ Spouse} . \text{Professor})) \sqsubseteq \text{Weird} \end{aligned}$$

Advantages: more compact (no variables), supports counting, but still decidable

- With lambda calculus, we can really capture any statement that we want, but it is often too powerful, which makes life difficult for algorithms. Description logic is a reaction to this, and carves out a logic which can be more intuitive to read, and also is actually decidable (recall that even first-order logic was semi-decidable).
- The lesson is that the complexity of a logic is not a one-dimensional quantity. The art of designing a logical language is coming up with operators that focus on representing formulas that actually arise in practice, while giving up on those that don't. By giving up on those things, we can hope to win on computation.
- Description logic is in a way more powerful than first-order logic because it can support "at least", but it is restricted in various ways that makes it decidable/tractable.



Summary of logics

- Propositional logic: $A \wedge B$
- First-order logic: $\forall x P(x) \rightarrow Q(x)$
- Temporal / epistemic logic: $\mathbf{F}(A \wedge B), \mathbf{B}_{\text{alice}}(A \wedge B)$
- Description logic: $P \sqsubseteq Q$
- Lambda calculus: $\lambda x P(x) \wedge Q(x)$

- We have quickly surveyed a host of logics with varying levels of expressive power and computational complexity. There is much more to say about each logic, but the high-level takeaway is an appreciation for the complexity of statements we can make formally.



Roadmap

Other logics

Markov logic

Language to logic

Limitations of hard logic

So far: a formula carves out subset of models.

$\text{Rain} \wedge \neg \text{Snow}$

	Snow	
	0	1
Rain, Traffic	0,0	
	0,1	
	1,0	
	1,1	

In reality, there is uncertainty.



Key idea: distribution

Model a distribution over models $\mathbb{P}(W = w; \theta)$.

- Now let's fall back down to first-order logic, and try to address another limitation of first-order logic, and actually more generally, a limitation of all classic hard logics, in which all statements are either true or false.
- So far, we've used formulas to carve out a set of models. But this doesn't allow us to represent the fact that some models might be more likely than others. For this, we will turn to probability.



Hiking

Everyone who likes hiking likes nature.

$$\forall x \text{ Likes}(x, \text{hiking}) \rightarrow \text{Likes}(x, \text{nature})$$

Bob doesn't like nature.

$$\neg \text{Likes}(\text{bob}, \text{nature})$$

Does Bob like hiking?

$$\text{Likes}(\text{bob}, \text{hiking})?$$

Does Alice like nature?

$$\text{Likes}(\text{alice}, \text{nature})?$$

- Before we dive into probability, let's set up a running example.
- Suppose we are given the first two first-order logic formulas, which we add to our knowledge base. How can we answer the next two questions? From last lecture, we could convert all the formulas to CNF form, and apply resolution to determine whether the questions are true, false, or indeterminate.
- This certainly works, but we will explore an alternative which is more conducive to a probabilistic generalization.



Hiking: propositionalization

First-order logic KB

$$\forall x \text{ Likes}(x, \text{hiking}) \rightarrow \text{Likes}(x, \text{nature})$$
$$\neg \text{Likes}(\text{bob}, \text{nature})$$

Assume unique names + domain closure.

Propositionalized KB

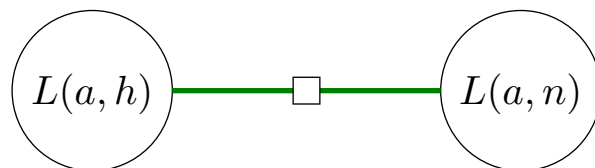
$$\text{Likes}(\text{alice}, \text{hiking}) \rightarrow \text{Likes}(\text{alice}, \text{nature})$$
$$\text{Likes}(\text{bob}, \text{hiking}) \rightarrow \text{Likes}(\text{bob}, \text{nature})$$
$$\neg \text{Likes}(\text{bob}, \text{nature})$$

- The first step is to propositionalize the first-order logic KB, which means to expand all quantified statements (this was covered last lecture).
- The end result is that we have a set of **ground formulas**, which are formulas without quantifiers and whose atomic formulas (e.g., Likes(alice, hiking)) do not contain variables.

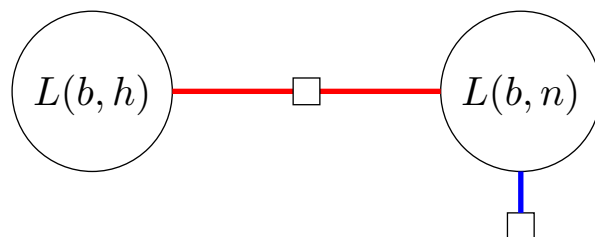


Hiking: factor graph

$[\text{Likes}(\text{alice}, \text{hiking}) \rightarrow \text{Likes}(\text{alice}, \text{nature})]$



$[\text{Likes}(\text{bob}, \text{hiking}) \rightarrow \text{Likes}(\text{bob}, \text{nature})]$



$[\neg \text{Likes}(\text{bob}, \text{nature})]$

[demo]

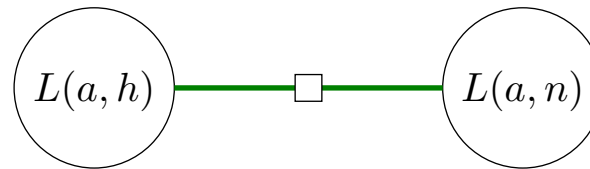
- Now recall that a set of formulas of this form can essentially be thought of as a set of propositional logic formulas (with atomic formulas rather than propositional symbols), and thus be written as a (boolean) constraint satisfaction problem, or more generally, a factor graph.
- Recall that the variables are atomic formulas, and the factors are the ground formulas. In the demo code, note that we are using a for loop to create all the variables and factors. This loop corresponds to the unrolling of the quantifier.
- We can use any inference algorithm (backtracking search, variable elimination, Gibbs sampling) to (approximately) solve the CSP, and determine that Bob doesn't like hiking (this is true in all three compatible models) and Alice could either like hiking or not.

Hiking: factor graph

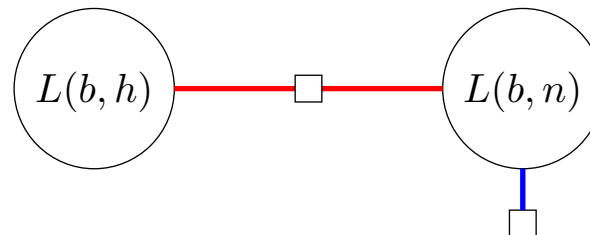
Problem: always exceptions!

Solution: soften factors from $\{0, 1\}$ to $\{1, 2\}$

$$[\text{Likes}(\text{alice}, \text{hiking}) \rightarrow \text{Likes}(\text{alice}, \text{nature})] + 1$$



$$[\text{Likes}(\text{bob}, \text{hiking}) \rightarrow \text{Likes}(\text{bob}, \text{nature})] + 1$$

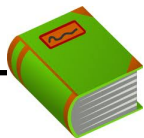


$$[\neg \text{Likes}(\text{bob}, \text{nature})] + 1$$

[demo: +1 in implies, +1 in not]

- However, what if we didn't believe 100% that everyone who likes hiking likes nature, since in the real world, there are always exceptions. Thus, we would like to soften the factors so that they don't return 0 and 1.
- One quick hack that gives the right intuition is to simply add 1 to all the factor values, so that all the values are 1 or 2 instead of 0 or 1.
- In the new factor graph, note that all 16 models have non-zero weight. If we normalize the weights to get a probability distribution over models, then we see that all models also have non-zero probability. We can compute the marginal probability of any atomic formula such as $\mathbb{P}(\text{Likes}(\text{bob}, \text{hiking}) = 1)$, which turns out to be 40%, not 0% as before.

Markov logic



Definition: Markov logic network

First-order logic formulas: f_1, \dots, f_m

Sets of ground formulas: S_1, \dots, S_m

Parameters: $\theta_1, \dots, \theta_m$

Recall: $\mathcal{I}(f, w) \in \{0, 1\}$ is the interpretation of f in w

A **Markov logic network** defines a probability distribution over models:

$$\mathbb{P}(W = w; \theta) \propto \exp \left\{ \sum_{j=1}^m \theta_j \sum_{f \in S_j} \mathcal{I}(f, w) \right\}$$

- Now let's formalize the intuition that we've developed so far. A Bayesian network was a factor graph that was defined via local conditional distributions; we will define a **Markov logic network** in terms of weighted first-order logic formulas f_1, \dots, f_m .
- Recall that each such formula f_j when propositionalized yields a set of ground formulas S_j . Also associate with each f_j a real-valued parameter $\theta_j \in \mathbb{R}$, which determines how true we think this formula is (the more positive it is, the more we believe it is true; negative corresponds to false).
- Given these elements, we define a distribution over models w (which are assignments of truth values to atomic formulas). The way Markov logic networks are generally defined is the exponential of a sum. Equivalently, we can think about this in terms of a factor graph, where each ground formula (factor) $f \in S_j$ has weight e^{θ_j} if f is satisfied in w ($\mathcal{I}(f, w) = 1$), and $e^0 = 1$ otherwise.

Markov logic: example



Example: Markov logic network

$$f_1 = \forall x \text{ Likes}(x, \text{hiking}) \rightarrow \text{Likes}(x, \text{nature})$$

$$f_2 = \neg \text{Likes}(\text{bob}, \text{nature})$$

$$S_1 = \{ \text{Likes}(\text{alice}, \text{hiking}) \rightarrow \text{Likes}(\text{alice}, \text{nature}), \\ \text{Likes}(\text{bob}, \text{hiking}) \rightarrow \text{Likes}(\text{bob}, \text{nature}) \}$$

$$S_2 = \{ \neg \text{Likes}(\text{bob}, \text{nature}) \}$$

$$\text{Hard logic: } \theta_1, \theta_2 \rightarrow \infty$$

$$\text{Soft logic: } \theta_1 = \theta_2 = \log(2) \text{ (for example)}$$

- To encode our hiking example in terms of a Markov logic network, we can define the first-order logic formulas f_1, f_2 and ground formulas S_1, S_2 as shown.
- To recover hard logic, we let the parameters tend to infinity, which means we most strongly believe the formulas are true. This setting of parameters will actually do a bit more: assuming $\theta_1 = \theta_2$, it will put a uniform distribution over all models which have the most number of ground formulas satisfied. This is because each additional satisfied formula contributes a weight of $e^{\theta_1} = e^{\theta_2}$, which tends to infinity.
- To recover our initial attempt to soften, we can set the parameter to $\log(2)$.

Markov logic: learning

Model:

$$\mathbb{P}(W = w; \theta) \propto \exp \left\{ \sum_{j=1}^m \theta_j \sum_{f \in S_j} \mathcal{I}(f, w) \right\}$$

Maximum likelihood (data: w_0)

$$\max_{\theta} \log \mathbb{P}(W = w_0; \theta)$$

Algorithm (gradient descent):

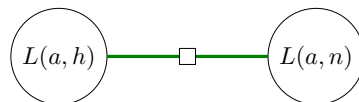
$$\theta \leftarrow \theta + \eta \nabla_{\theta} \log \mathbb{P}(W = w; \theta)$$

- So far, we have assumed the parameters $\theta_1, \dots, \theta_m$ are fixed, and we can do probabilistic inference given these parameters (using Gibbs sampling, particle filtering, etc.).
- Now let's turn to learning, which is the inverse problem: if we see a particular model w_0 , can we learn the parameters?
- To do learning, we turn to our faithful maximum likelihood principle, which served us so well for Bayesian networks. The goal is to find the θ that maximizes the probability of the data w_0 .
- Given this objective function, the most canonical method to optimize it is to use gradient descent. Computing the gradient involves computing the probability of each ground formula being true, which requires probabilistic inference.

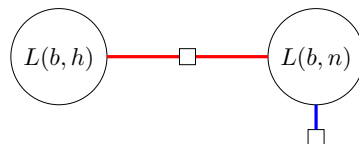


Markov logic

$$[\text{Likes}(\text{alice}, \text{hiking}) \rightarrow \text{Likes}(\text{alice}, \text{nature})] + 1$$



$$[\text{Likes}(\text{bob}, \text{hiking}) \rightarrow \text{Likes}(\text{bob}, \text{nature})] + 1$$



$$[\neg \text{Likes}(\text{bob}, \text{nature})] + 1$$

- Defines distribution over possible worlds (models)
- Parameter sharing between all grounded instances of a formula
- Probabilistic inference: Gibbs sampling
- Learning: maximum likelihood, gradient descent

- In summary, Markov logic takes first-order logic (under unique names and domain closure) and casts it as a factor graph, from which we can apply standard factor graph inference and learning algorithms that aren't specialized to logic.
- One additional point which we didn't discuss is the fact that the factors are not arbitrary, but rather based on first-order logic formulas. We can exploit this structure, by grouping variables (which correspond to ground formulas) that behave similarly. This is analogous to performing first-order logic inference without propositionalization. Operating implicitly in this way is called **lifted inference**.



Roadmap

Other logics

Markov logic

Language to logic

From language to logic

Alice likes hiking.  Likes(alice, hiking)

Alice likes geometry.  Likes(alice, geometry)

Bob likes hiking.  Likes(bob, hiking)

Bob likes geometry.  Likes(bob, geometry)

Lots of regularities — can we convert language to logic automatically?

- So far, we've used natural language as a pedagogical means of helping you understand the semantics of logical formulas. But after a while, you might note that there is actually quite a bit of similarity between sentences in natural language and logical formulas. This suggests that it might be possible to build a system that can automate this conversion.
- Such a system would be incredibly useful, since it would allow us to talk to a computer using natural language, conveying the rich information contained with the implied logical formulas. Another motivation is that much of human knowledge is actually written down in text. We'd like a computer to internalize this knowledge and support question answering from it. Think next-generation search.

From language to logic



Key idea: principle of compositionality

The semantics of a sentence is combination of meanings of its parts.

Sentence:

Alice likes hiking. \longrightarrow Likes(alice, hiking)

Words:

Alice \longrightarrow alice

hiking \longrightarrow hiking

likes \longrightarrow $\lambda y \lambda x \text{ Likes}(x, y)$

- To make the mapping from sentences to logical formulas work, we leverage the principle of compositionality, which basically says that this mapping is defined recursively based on subparts.
- We can say that the word *Alice* maps to the logical term *alice*; *hiking* to *hiking*. But if *likes* simply *Likes*, then we don't have enough information about how to combine the parts. Therefore, we let *likes* map onto a function that takes two arguments y and x (in that order), and returns $Likes(x, y)$.

Lambda calculus: example 1

Function:

$$\lambda x \text{ Student}(x) \wedge \text{Likes}(x, \text{hiking})$$

Argument:

alice

Function application:

$$(\lambda x \text{ Student}(x) \wedge \text{Likes}(x, \text{hiking}))(\text{alice})$$
$$\text{Student}(\text{alice}) \wedge \text{Likes}(\text{alice}, \text{hiking})$$

- Before getting into how we build up the full logical form, let's give a few quick lambda calculus examples.
- First, suppose we have a function and argument. Function application substitutes the argument (e.g., alice) in for the variable of the function (x).

Lambda calculus: example 2

Function:

$$\lambda y \lambda x \text{ Likes}(x, y)$$

Argument:

hiking

Function application:

$$(\lambda y \lambda x \text{ Likes}(x, y))(\text{hiking}) =$$
$$\lambda x \text{ Likes}(x, \text{hiking})$$

Lambda calculus: example 2

Function:

$$\lambda f \lambda x \neg f(x)$$

Argument:

$$\lambda y \text{ Likes}(y, \text{hiking})$$

Function application:

$$(\lambda f \lambda x \neg f(x))(\lambda y \text{ Likes}(y, \text{hiking})) =$$

$$\lambda x \neg(\lambda y \text{ Likes}(y, \text{hiking}))(x) =$$

$$\lambda x \neg \text{Likes}(x, \text{hiking})$$

- As a more complex example, arguments can themselves be functions. Working through the substitutions, we get that this function performs negation, turning "people who like hiking" to "people who don't like hiking".
- One of the powerful aspects of lambda calculus is exactly that functions can be passed into other functions. But to be absolutely rigorous, we would use simply-typed lambda calculus, which associates each function with a type (e.g., $\text{object} \rightarrow \text{bool}$).


Grammar

Grammar

Lexicon:

Alice  alice

hiking  hiking

likes  $\lambda y \lambda x \text{ Likes}(x, y)$

Forward application:

f *x*  $f(x)$

Backward application:

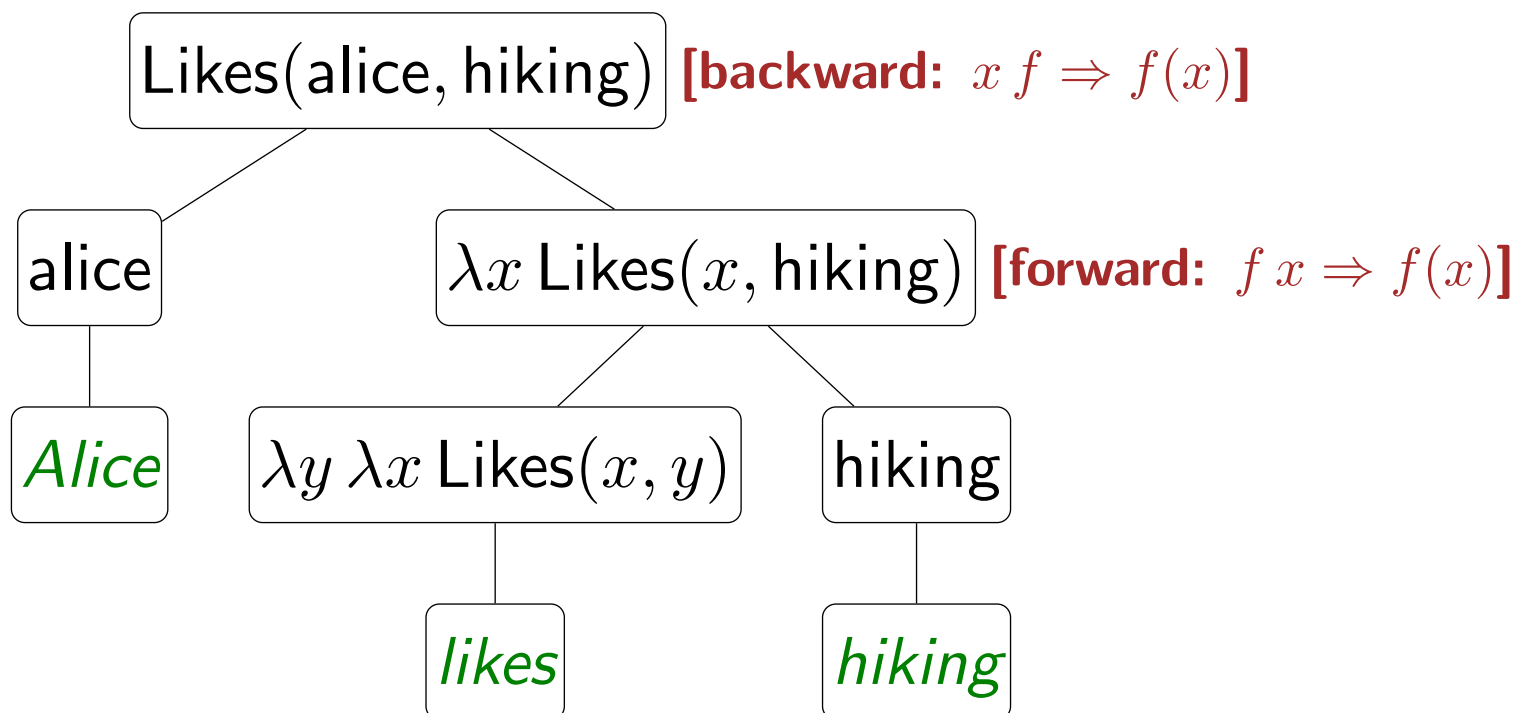
x *f*  $f(x)$

- A grammar is a set of rules. The lexicon contains a set of lexical rules, which map natural language to formulas. We also have two rules that perform forward and backward application (depending on whether the function precedes or succeeds the argument).
- Advanced: if you take a semantic class in linguistics or CS224U, you will get a richer linguistic treatment of this material. We have deliberately tried to suppress that in the interest of keeping things simple. Each rule can not only produce a logical formula, but also a syntactic category (e.g., sentence or noun phrase), which can be used to restrict when rules are applied.
- Advanced: also note that a context-free grammar (CFG) is also a different beast than the grammars here because the emphasis is on determining which sentences are licensed under the CFG (yes or no), whereas here, our expressed emphasis is to actually produce the logical formula for the meaning of that sentence.

Basic derivation

Leaves: input words

Internal nodes: produced by applying rule to children



- This is perhaps the most important slide. Given a sequence of input words (as the leaves of the tree), we build a logical form at each intermediate node by applying a rule to the formulas of its children. The formula at the root is the formula that's output and returned.
- Next, we will walk through some important linguistic phenomena.

Negation

Alice does not like hiking.

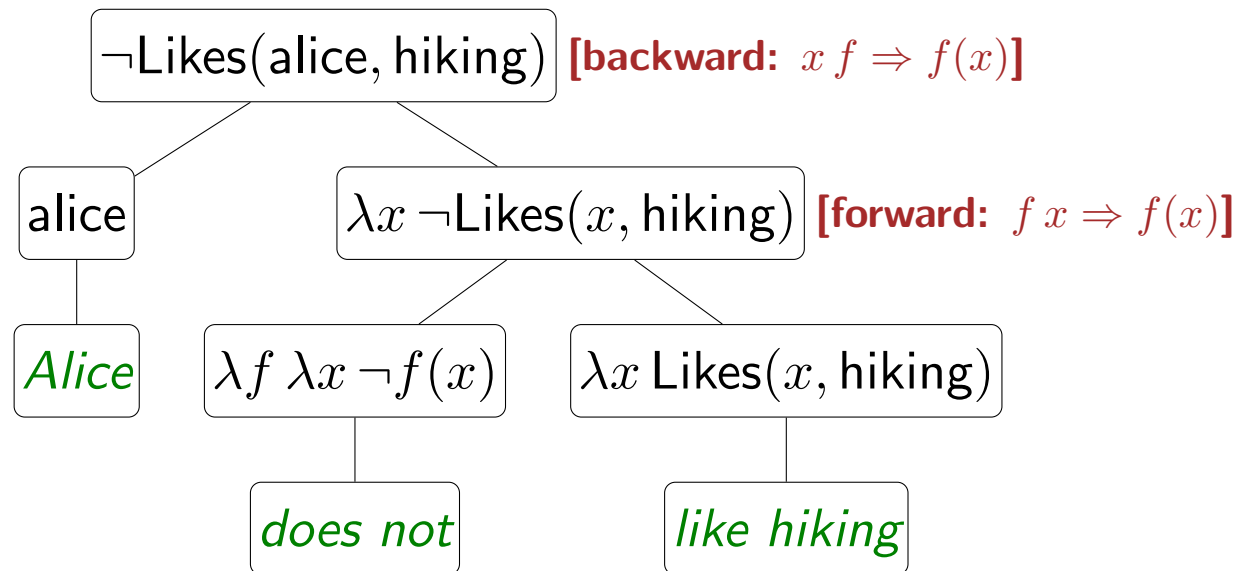
$\neg \text{Likes}(\text{alice}, \text{hiking})$

Grammar

...

Negation:

does not $\longrightarrow \lambda f \lambda x \neg f(x)$



- Negation is an important construct that inverts the meaning of a sentence. As we saw earlier with lambda calculus example 2, we can turn $\lambda x \text{ Likes}(x, \text{hiking})$ into $\lambda x \neg \text{Likes}(x, \text{hiking})$.

Coordination 1

Alice likes hiking **and** *Bob likes swimming*.

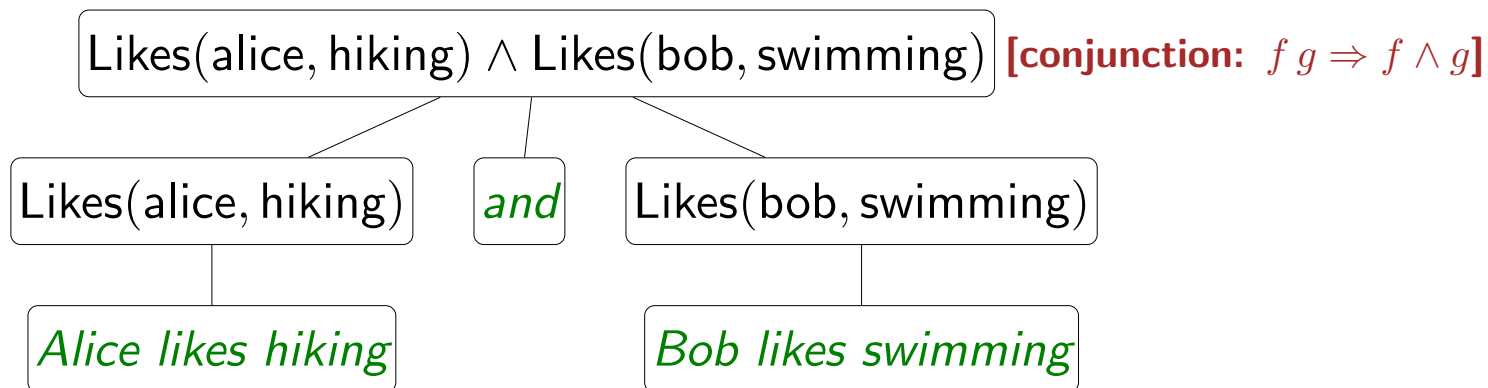
Likes(alice, hiking) \wedge Likes(bob, swimming)

Grammar

...

Coordination:

f and g \longrightarrow $f \wedge g$



- Coordination refers to the use of connective words such as *and*, *or*, etc. In the simplest coordination case, we are simply taking the conjunction of two formulas.

Coordination 2

Alice likes hiking and hates swimming.

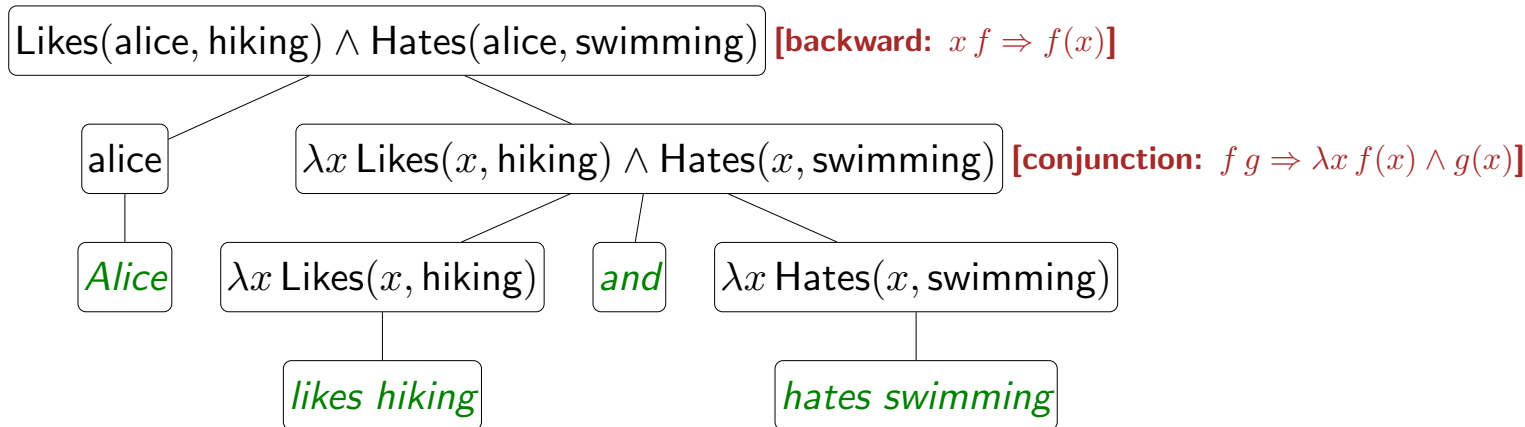
Likes(alice, hiking) \wedge Hates(alice, swimming)

Grammar

...

Coordination:

f and g \longrightarrow $\lambda x f(x) \wedge g(x)$



- However, the same word *and* can also be used to conjoin two incomplete sentences (*likes hiking* and *hates swimming*). Notice that *Alice* only shows up once in the sentence but twice in the formula.
- Here, we define a different coordination rule which takes the conjunction not of two formulas, but of two functions. Intuitively, this allows us to combine Likes and Hates "underneath" the λx .

Quantification

Every student likes hiking.

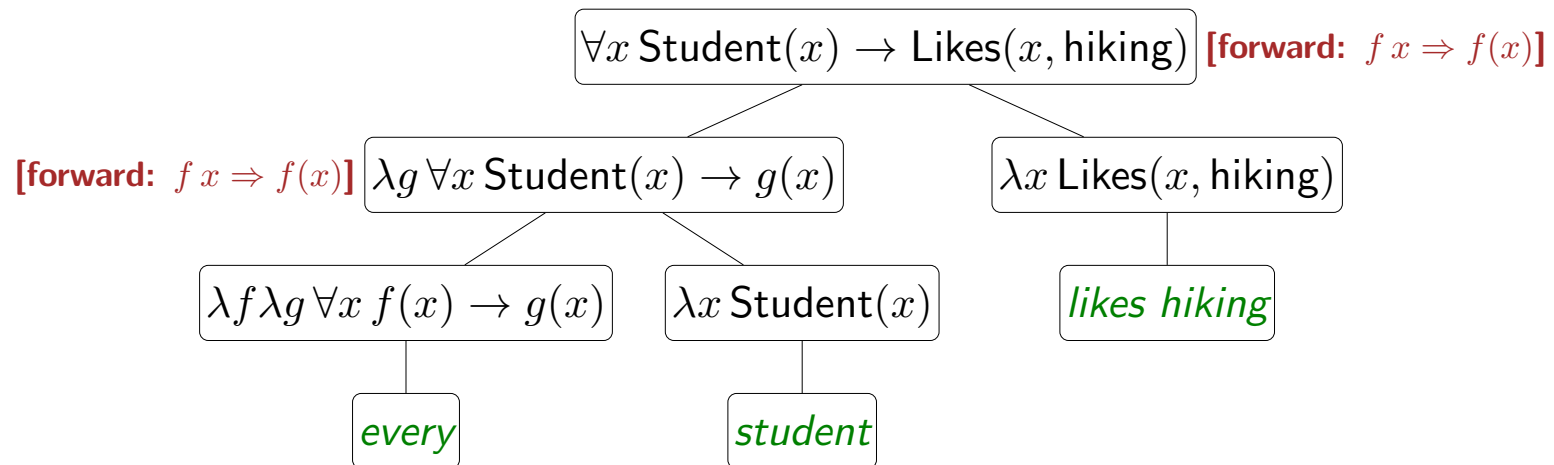
$\forall x \text{ Student}(x) \rightarrow \text{Likes}(x, \text{hiking})$

Grammar

...

Universal quantification:

every $\longrightarrow \lambda f \lambda g \forall x f(x) \rightarrow g(x)$



- The last phenomenon that we will discuss is quantification, which is a very rich topic, and is a major point of first-order logic too.
- In the simplest case where there is just one quantifier, we can add a rule that maps *every* to something that takes two functions, f and g , and returns the appropriate formula.
- In the context of the sentence, f gets bound to $\lambda x \text{ Student}(x)$ and g gets bound to $\lambda x \text{ Likes}(x, \text{hiking})$.

Ambiguity

Lexical ambiguity:

Alice went to the bank.  Travel(alice, RiverBank)

Alice went to the bank.  Travel(alice, MoneyBank)

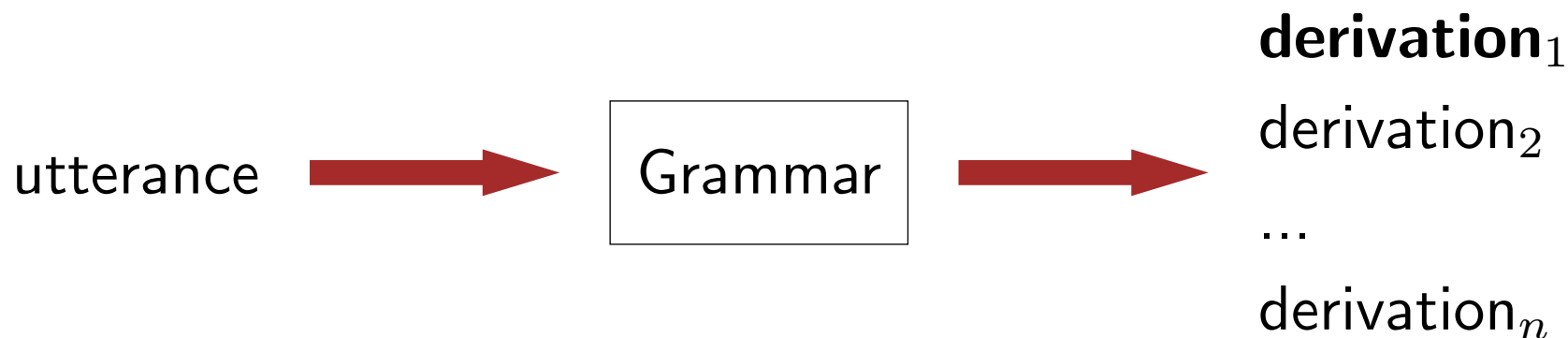
Scope ambiguity:

Everyone likes someone.  $\forall x \exists y \text{ Likes}(x, y)$

Everyone likes someone.  $\exists y \forall x \text{ Likes}(x, y)$

- Everything has worked out smoothly because we have deliberately chosen the right set of rules to apply so that we get the correct logical formulas. But in some case, the same piece of text can map to multiple valid logical formula that have different meanings!
- This discrepancy might be due to lexical ambiguity (individual words mean many different meanings) or scope ambiguity where the actual order of quantification is not specified by the text (natural language is weird like that).

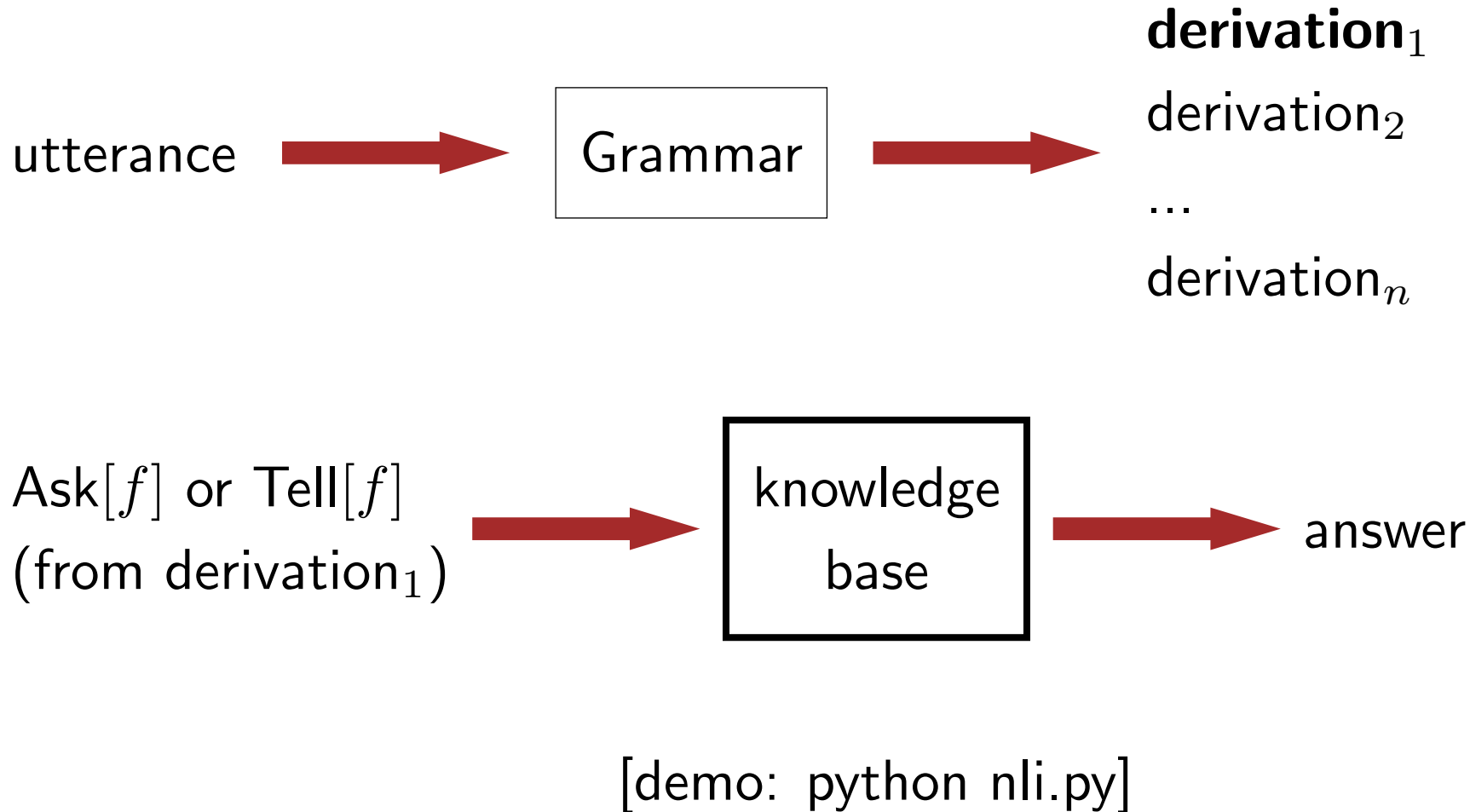
Algorithms



- **Inference (parsing)**: construct derivations recursively (dynamic programming)
- **Learning**: define ranking loss function, optimize with stochastic gradient descent

- We've seen many different linguistic phenomena, which all can be handled by adding rules. In practice, we construct a parsing algorithm that can output a set of derivations gotten by parsing the utterance with respect to the grammar.
- These algorithms are based on the recursive structure of language and look a lot like dynamic programming algorithms. Specifically, for each subsequence of the sentence, we build a set of sub-derivations.
- In learning semantic parsers, we are given a set of utterances paired with derivations. To learn the parameters, we can define a standard ranking function, which can be optimized with stochastic gradient descent.

Putting it together



- The result of the derivation is a logical formula equipped with either an ask or tell request. This can be again used to tell the knowledge base facts or ask questions. This KB could even be probabilistic via Markov logic.



Summary

- Logic is used to **represent** knowledge and perform **inferences** on it
- **Considerations**: expressiveness, notational convenience, inferential complexity
- **First-order logic**: objects/relations, quantify over variables
- **Other logics**: more expressivity or more efficient
- **Markov logic**: marry logic (abstract reasoning by working on formulas) and probability (maintaining uncertainty)
- **Semantic parsing**: map natural language to logic

- In conclusion, hopefully you have gotten a taste of how powerful logic can be as a means of expressing pretty complex facts using a very small means.
- There is no one true logic, and which one is the best to choose depends on the needs of the application coupled with computational budget.
- Also, there is no contradiction between probability and logic. Logic provides with notions of objects/relations — answering the question of what we should even be modeling. Probability allows us to place distributions over these notions defined by logic.
- The story is far from complete: classical logics still have much more expressiveness than Markov logic, so bringing richer types of statements into the realm of probability is an active area of research.
- Finally, semantic parsing provides an initial step towards the goal of being able to process arbitrary text into a knowledge base and automatically make inferences about it. While we talked mostly about the structural aspects of semantic parsers (e.g., the grammar), there has been a surge of interest lately in making these semantic parsers more robust and scalable. There is still much work to be done though.