Chapter 12

Logical semantics

The previous few chapters have focused on building systems that reconstruct the **syntax** of natural language — its structural organization — through tagging and parsing. But some of the most exciting and promising potential applications of language technology involve going beyond syntax to **semantics** — the underlying meaning of the text:

- Answering questions, such as *where is the nearest coffeeshop?* or *what is the middle name of the mother of the 44th President of the United States?*.
- Building a robot that can follow natural language instructions to execute tasks.
- Translating a sentence from one language into another, while preserving the underlying meaning.
- Fact-checking an article by searching the web for contradictory evidence.
- Logic-checking an argument by identifying contradictions, ambiguity, and unsupported assertions.

Semantic analysis involves converting natural language into a **meaning representation**. To be useful, a meaning representation must meet several criteria:

- **c1**: it should be unambiguous: unlike natural language, there should be exactly one meaning per statement;
- **c2**: it should provide a way to link language to external knowledge, observations, and actions;
- **c3**: it should support computational **inference**, so that meanings can be combined to derive additional knowledge;
- **c4**: it should be expressive enough to cover the full range of things that people talk about in natural language.

Much more than this can be said about the question of how best to represent knowledge for computation (e.g., Sowa, 2000), but this chapter will focus on these four criteria.

12.1 Meaning and denotation

The first criterion for a meaning representation is that statements in the representation should be unambiguous — they should have only one possible interpretation. Natural language does not have this property: as we saw in chapter 10, sentences like *cats scratch people with claws* have multiple interpretations.

But what does it mean for a statement to be unambiguous? Programming languages provide a useful example: the output of a program is completely specified by the rules of the language and the properties of the environment in which the program is run. For example, the python code `5 + 3` will have the output 8, as will the codes `(4*4)-(3*3)+1` and `((8))`. This output is known as the **denotation** of the program, and can be written as,

$$[\![\texttt{5+3}]\!] = [\![\texttt{(4*4)-(3*3)+1}]\!] = [\![\texttt{((8))}]\!] = 8. \qquad [12.1]$$

The denotations of these arithmetic expressions are determined by the meaning of the **constants** (e.g., `5`, `3`) and the **relations** (e.g., `+`, `*`, `(`, `)`). Now let's consider another snippet of python code, `double(4)`. The denotation of this code could be, $[\![\texttt{double(4)}]\!] = \texttt{8}$, or it could be $[\![\texttt{double(4)}]\!] = \texttt{44}$ — it depends on the meaning of `double`. This meaning is defined in a **world model** $\mathcal{M}$ as an infinite set of pairs. We write the denotation with respect to model $\mathcal{M}$ as $[\![\cdot]\!]_{\mathcal{M}}$, e.g., $[\![\texttt{double}]\!]_{\mathcal{M}} = \{(\texttt{0},\texttt{0}), (\texttt{1},\texttt{2}), (\texttt{2},\texttt{4}), \ldots\}$. The world model would also define the (infinite) list of constants, e.g., $\{\texttt{0,1,2,...}\}$. As long as the denotation of string ϕ in model $\mathcal{M}$ can be computed unambiguously, the language can be said to be unambiguous.

This approach to meaning is known as **model-theoretic semantics**, and it addresses not only criterion $c1$ (no ambiguity), but also $c2$ (connecting language to external knowledge, observations, and actions). For example, we can connect a representation of the meaning of a statement like *the capital of Georgia* with a world model that includes knowledge base of geographical facts, obtaining the denotation `Atlanta`. We might populate a world model by detecting and analyzing the objects in an image, and then use this world model to evaluate **propositions** like *a man is riding a moose*. Another desirable property of model-theoretic semantics is that when the facts change, the denotations change too: the meaning representation of *President of the USA* would have a different denotation in the model $\mathcal{M}_{2014}$ as it would in $\mathcal{M}_{2022}$.

12.2 Logical representations of meaning

Criterion $c3$ requires that the meaning representation support inference — for example, automatically deducing new facts from known premises. While many representations have been proposed that meet these criteria, the most mature is the language of first-order logic.[1]

12.2.1 Propositional logic

The bare bones of logical meaning representation are Boolean operations on propositions:

Propositional symbols. Greek symbols like ϕ and ψ will be used to represent **propositions**, which are statements that are either true or false. For example, ϕ may correspond to the proposition, *bagels are delicious*.

Boolean operators. We can build up more complex propositional formulas from Boolean operators. These include:

- Negation $\neg\phi$, which is true if ϕ is false.
- Conjunction, $\phi \wedge \psi$, which is true if both ϕ and ψ are true.
- Disjunction, $\phi \vee \psi$, which is true if at least one of ϕ and ψ is true
- Implication, $\phi \Rightarrow \psi$, which is true unless ϕ is true and ψ is false. Implication has identical truth conditions to $\neg\phi \vee \psi$.
- Equivalence, $\phi \Leftrightarrow \psi$, which is true if ϕ and ψ are both true or both false. Equivalence has identical truth conditions to $(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$.

It is not strictly necessary to have all five Boolean operators: readers familiar with Boolean logic will know that it is possible to construct all other operators from either the NAND (not-and) or NOR (not-or) operators. Nonetheless, it is clearest to use all five operators. From the truth conditions for these operators, it is possible to define a number of "laws" for these Boolean operators, such as,

- *Commutativity*: $\phi \wedge \psi = \psi \wedge \phi, \quad \phi \vee \psi = \psi \vee \phi$
- *Associativity*: $\phi \wedge (\psi \wedge \chi) = (\phi \wedge \psi) \wedge \chi, \quad \phi \vee (\psi \vee \chi) = (\phi \vee \psi) \vee \chi$
- *Complementation*: $\phi \wedge \neg\phi = \bot, \quad \phi \vee \neg\phi = \top$, where $\top$ indicates a true proposition and $\bot$ indicates a false proposition.

[1]Alternatives include the "variable-free" representation used in semantic parsing of geographical queries (Zelle and Mooney, 1996) and robotic control (Ge and Mooney, 2005), and dependency-based compositional semantics (Liang et al., 2013).

These laws can be combined to derive further equivalences, which can support logical inferences. For example, suppose ϕ = *The music is loud* and ψ = *Max can't sleep*. Then if we are given,

$\phi \Rightarrow \psi$	*If the music is loud, Max can't sleep.*
ϕ	*The music is loud.*

we can derive ψ (*Max can't sleep*) by application of **modus ponens**, which is one of a set of **inference rules** that can be derived from more basic laws and used to manipulate propositional formulas. **Automated theorem provers** are capable of applying inference rules to a set of premises to derive desired propositions (Loveland, 2016).

12.2.2 First-order logic

Propositional logic is so named because it treats propositions as its base units. However, the criterion $c4$ states that our meaning representation should be sufficiently expressive. Now consider the sentence pair,

(12.1) If anyone is making noise, then Max can't sleep.
Abigail is making noise.

People are capable of making inferences from this sentence pair, but such inferences require formal tools that are beyond propositional logic. To understand the relationship between the statement *anyone is making noise* and the statement *Abigail is making noise*, our meaning representation requires the additional machinery of **first-order logic** (FOL).

In FOL, logical propositions can be constructed from relationships between entities. Specifically, FOL extends propositional logic with the following classes of terms:

Constants. These are elements that name individual entities in the model, such as MAX and ABIGAIL. The denotation of each constant in a model $\mathcal{M}$ is an element in the model, e.g., $[\![\text{MAX}]\!] = \texttt{m}$ and $[\![\text{ABIGAIL}]\!] = \texttt{a}$.

Relations. Relations can be thought of as sets of entities, or sets of tuples. For example, the relation CAN-SLEEP is defined as the set of entities who can sleep, and has the denotation $[\![\text{CAN-SLEEP}]\!] = \{\texttt{a}, \texttt{m}, \ldots\}$. To test the truth value of the proposition CAN-SLEEP(MAX), we ask whether $[\![\text{MAX}]\!] \in [\![\text{CAN-SLEEP}]\!]$. Logical relations that are defined over sets of entities are sometimes called *properties*.

Relations may also be ordered tuples of entities. For example BROTHER(MAX,ABIGAIL) expresses the proposition that MAX is the brother of ABIGAIL. The denotation of such relations is a set of tuples, $[\![\text{BROTHER}]\!] = \{\texttt{(m,a)}, \texttt{(x,y)}, \ldots\}$. To test the truth value of the proposition BROTHER(MAX,ABIGAIL), we ask whether the tuple $([\![\text{MAX}]\!], [\![\text{ABIGAIL}]\!])$ is in the denotation $[\![\text{BROTHER}]\!]$.

Using constants and relations, it is possible to express statements like *Max can't sleep* and *Max is Abigail's brother*:

$$\neg\text{CAN-SLEEP}(\text{MAX})$$
$$\text{BROTHER}(\text{MAX},\text{ABIGAIL}).$$

These statements can also be combined using Boolean operators, such as,

$$(\text{BROTHER}(\text{MAX},\text{ABIGAIL}) \vee \text{BROTHER}(\text{MAX},\text{STEVE})) \Rightarrow \neg\text{CAN-SLEEP}(\text{MAX}).$$

This fragment of first-order logic permits only statements about specific entities. To support inferences about statements like *If* anyone *is making noise, then Max can't sleep*, two more elements must be added to the meaning representation:

Variables. Variables are mechanisms for referring to entities that are not locally specified. We can then write CAN-SLEEP(x) or BROTHER(x, ABIGAIL). In these cases, x is a **free variable**, meaning that we have not committed to any particular assignment.

Quantifiers. Variables are bound by quantifiers. There are two quantifiers in first-order logic.[2]

- The **existential quantifier** $\exists$, which indicates that there must be at least one entity to which the variable can bind. For example, the statement $\exists x$MAKES-NOISE(X) indicates that there is at least one entity for which MAKES-NOISE is true.
- The **universal quantifier** $\forall$, which indicates that the variable must be able to bind to any entity in the model. For example, the statement,

$$\text{MAKES-NOISE}(\text{ABIGAIL}) \Rightarrow (\forall x \neg\text{CAN-SLEEP}(x)) \qquad [12.3]$$

asserts that if Abigail makes noise, no one can sleep.

The expressions $\exists x$ and $\forall x$ make x into a **bound variable**. A formula that contains no free variables is a **sentence**.

Functions. Functions map from entities to entities, e.g., $[\![\text{CAPITAL-OF}(\text{GEORGIA})]\!] = [\![\text{ATLANTA}]\!]$. With functions, it is convenient to add an equality operator, supporting statements like,

$$\forall x \exists y \text{MOTHER-OF}(x) = \text{DAUGHTER-OF}(y). \qquad [12.4]$$

[2]In first-order logic, it is possible to quantify only over entities. In **second-order logic**, it is possible to quantify over properties. This makes it possible to represent statements like *Butch has every property that a good boxer has* (example from Blackburn and Bos, 2005),

$$\forall P \forall x ((\text{GOOD-BOXER}(x) \Rightarrow P(x)) \Rightarrow P(\text{BUTCH})). \qquad [12.2]$$

Note that $\textsc{mother-of}$ is a functional analogue of the relation $\textsc{mother}$, so that $\textsc{mother-of}(x) = y$ if $\textsc{mother}(x, y)$. Any logical formula that uses functions can be rewritten using only relations and quantification. For example,

$$\textsc{makes-noise}(\textsc{mother-of}(\textsc{abigail})) \qquad [12.5]$$

can be rewritten as $\exists x \textsc{makes-noise}(x) \wedge \textsc{mother}(x, \textsc{abigail})$.

An important property of quantifiers is that the order can matter. Unfortunately, natural language is rarely clear about this! The issue is demonstrated by examples like *everyone speaks a language*, which has the following interpretations:

$$\forall x \exists y \ \textsc{speaks}(x, y) \qquad [12.6]$$

$$\exists y \forall x \ \textsc{speaks}(x, y). \qquad [12.7]$$

In the first case, y may refer to several different languages, while in the second case, there is a single y that is spoken by everyone.

Truth-conditional semantics

One way to look at the meaning of an FOL sentence ϕ is as a set of **truth conditions**, or models under which ϕ is satisfied. But how to determine whether a sentence is true or false in a given model? We will approach this inductively, starting with a predicate applied to a tuple of constants. The truth of such a sentence depends on whether the tuple of denotations of the constants is in the denotation of the predicate. For example, $\textsc{capital}(\textsc{georgia},\textsc{atlanta})$ is true in model $\mathcal{M}$ iff,

$$(\llbracket \textsc{georgia} \rrbracket_{\mathcal{M}}, \llbracket \textsc{atlanta} \rrbracket_{\mathcal{M}}) \in \llbracket \textsc{capital} \rrbracket_{\mathcal{M}}. \qquad [12.8]$$

The Boolean operators $\wedge, \vee, \ldots$ provide ways to construct more complicated sentences, and the truth of such statements can be assessed based on the truth tables associated with these operators. The statement $\exists x \phi$ is true if there is some assignment of the variable x to an entity in the model such that ϕ is true; the statement $\forall x \phi$ is true if ϕ is true under all possible assignments of x. More formally, we would say that ϕ is **satisfied** under $\mathcal{M}$, written as $\mathcal{M} \models \phi$.

Truth conditional semantics allows us to define several other properties of sentences and pairs of sentences. Suppose that in every $\mathcal{M}$ under which ϕ is satisfied, another formula ψ is also satisfied; then ϕ **entails** ψ, which is also written as $\phi \models \psi$. For example,

$$\textsc{capital}(\textsc{georgia},\textsc{atlanta}) \models \exists x \textsc{capital}(\textsc{georgia}, x). \qquad [12.9]$$

A statement that is satisfied under any model, such as $\phi \vee \neg\phi$, is **valid**, written $\models (\phi \vee \neg\phi)$. A statement that is not satisfied under any model, such as $\phi \wedge \neg\phi$, is **unsatisfiable**, or **inconsistent**. A **model checker** is a program that determines whether a sentence ϕ is satisfied in $\mathcal{M}$. A **model builder** is a program that constructs a model in which ϕ is satisfied. The problems of checking for consistency and validity in first-order logic are **undecidable**, meaning that there is no algorithm that can automatically determine whether an FOL formula is valid or inconsistent.

Inference in first-order logic

Our original goal was to support inferences that combine general statements *If anyone is making noise, then Max can't sleep* with specific statements like *Abigail is making noise*. We can now represent such statements in first-order logic, but how are we to perform the inference that *Max can't sleep*? One approach is to use "generalized" versions of propositional inference rules like modus ponens, which can be applied to FOL formulas. By repeatedly applying such inference rules to a knowledge base of facts, it is possible to produce proofs of desired propositions. To find the right sequence of inferences to derive a desired theorem, classical artificial intelligence search algorithms like backward chaining can be applied. Such algorithms are implemented in interpreters for the `prolog` logic programming language (Pereira and Shieber, 2002).

12.3 Semantic parsing and the lambda calculus

The previous section laid out a lot of formal machinery; the remainder of this chapter links these formalisms back to natural language. Given an English sentence like *Alex likes Brit*, how can we obtain the desired first-order logical representation, LIKES(ALEX,BRIT)? This is the task of **semantic parsing**. Just as a syntactic parser is a function from a natural language sentence to a syntactic structure such as a phrase structure tree, a semantic parser is a function from natural language to logical formulas.

As in syntactic analysis, semantic parsing is difficult because the space of inputs and outputs is very large, and their interaction is complex. Our best hope is that, like syntactic parsing, semantic parsing can somehow be decomposed into simpler sub-problems. This idea, usually attributed to the German philosopher Gottlob Frege, is called the **principle of compositionality**: the meaning of a complex expression is a function of the meanings of that expression's constituent parts. We will define these "constituent parts" as syntactic constituents: noun phrases and verb phrases. These constituents are combined using function application: if the syntactic parse contains the production $x \rightarrow y\ z$, then the semantics of x, written x.sem, will be computed as a function of the semantics of the

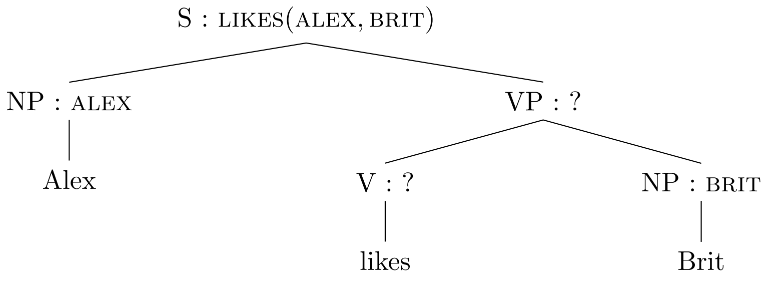


Figure 12.1: The principle of compositionality requires that we identify meanings for the constituents *likes* and *likes Brit* that will make it possible to compute the meaning for the entire sentence.

constituents, y.sem and z.sem.[3] [4]

12.3.1 The lambda calculus

Let's see how this works for a simple sentence like *Alex likes Brit*, whose syntactic structure is shown in Figure 12.1. Our goal is the formula, LIKES(ALEX,BRIT), and it is clear that the meaning of the constituents *Alex* and *Brit* should be ALEX and BRIT. That leaves two more constituents: the verb *likes*, and the verb phrase *likes Brit*. The meanings of these units must be defined in a way that makes it possible to recover the desired meaning for the entire sentence by function application. If the meanings of *Alex* and *Brit* are constants, then the meanings of *likes* and *likes Brit* must be functional expressions, which can be applied to their siblings to produce the desired analyses.

Modeling these partial analyses requires extending the first-order logic meaning representation. We do this by adding **lambda expressions**, which are descriptions of anonymous functions,[5] e.g.,

$$\lambda x.\text{LIKES}(x, \text{BRIT}). \tag{12.10}$$

This functional expression is the meaning of the verb phrase *likes Brit*; it takes a single argument, and returns the result of substituting that argument for x in the expression

[3]§ 9.3.2 briefly discusses Combinatory Categorial Grammar (CCG) as an alternative to a phrase-structure analysis of syntax. CCG is argued to be particularly well-suited to semantic parsing (Hockenmaier and Steedman, 2007), and is used in much of the contemporary work on machine learning for semantic parsing, summarized in § 12.4.

[4]The approach of algorithmically building up meaning representations from a series of operations on the syntactic structure of a sentence is generally attributed to the philosopher Richard Montague, who published a series of influential papers on the topic in the early 1970s (e.g., Montague, 1973).

[5]Formally, all first-order logic formulas are lambda expressions; in addition, if ϕ is a lambda expression, then $\lambda x.\phi$ is also a lambda expression. Readers who are familiar with functional programming will recognize lambda expressions from their use in programming languages such as Lisp and Python.

$\text{LIKES}(x, \text{BRIT})$. We write this substitution as,

$$(\lambda x.\text{LIKES}(x, \text{BRIT}))@\text{ALEX} = \text{LIKES}(\text{ALEX},\text{BRIT}), \qquad [12.11]$$

with the symbol "@" indicating function application. Function application in the lambda calculus is sometimes called β**-reduction** or β-conversion. The expression $\phi@\psi$ indicates a function application to be performed by β-reduction, and $\phi(\psi)$ indicates a function or predicate in the final logical form.

Equation 12.11 shows how to obtain the desired semantics for the sentence *Alex likes Brit*: by applying the lambda expression $\lambda x.\text{LIKES}(x, \text{BRIT})$ to the logical constant ALEX. This rule of composition can be specified in a **syntactic-semantic grammar**, in which syntactic productions are paired with semantic operations. For the syntactic production S $\to$ NP VP, we have the semantic rule VP.sem@NP.sem.

The meaning of the transitive verb phrase *likes Brit* can also be obtained by function application on its syntactic constituents. For the syntactic production VP $\to$ V NP, we apply the semantic rule,

$$\begin{aligned}\text{VP.sem} =& (\text{V.sem})@\text{NP.sem} && [12.12]\\ =& (\lambda y.\lambda x.\text{LIKES}(x, y))@(\text{BRIT}) && [12.13]\\ =& \lambda x.\text{LIKES}(x, \text{BRIT}). && [12.14]\end{aligned}$$

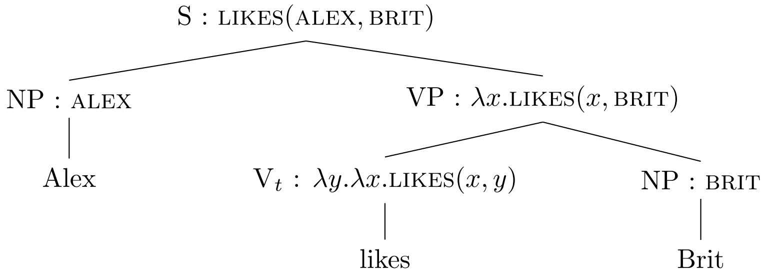
Thus, the meaning of the transitive verb *likes* is a lambda expression whose output is *another* lambda expression: it takes y as an argument to fill in one of the slots in the LIKES relation, and returns a lambda expression that is ready to take an argument to fill in the other slot.[6]

Table 12.1 shows a minimal syntactic-semantic grammar fragment, G_1. The complete **derivation** of *Alex likes Brit* in G_1 is shown in Figure 12.2. In addition to the transitive verb *likes*, the grammar also includes the intransitive verb *sleeps*; it should be clear how to derive the meaning of sentences like *Alex sleeps*. For verbs that can be either transitive or intransitive, such as *eats*, we would have two terminal productions, one for each sense (terminal productions are also called the **lexical entries**). Indeed, most of the grammar is in the **lexicon** (the terminal productions), since these productions select the basic units of the semantic interpretation.

12.3.2 Quantification

Things get more complicated when we move from sentences about named entities to sentences that involve more general noun phrases. Let's consider the example, *A dog sleeps*,

[6]This can be written in a few different ways. The notation $\lambda y, x.\text{LIKES}(x, y)$ is a somewhat informal way to indicate a lambda expression that takes two arguments; this would be acceptable in functional programming. Logicians (e.g., Carpenter, 1997) often prefer the more formal notation $\lambda y.\lambda x.\text{LIKES}(x)(y)$, indicating that each lambda expression takes exactly one argument.



$S : \textsc{likes}(\textsc{alex}, \textsc{brit})$
$NP : \textsc{alex}$
Alex
$VP : \lambda x.\textsc{likes}(x, \textsc{brit})$
$V_t : \lambda y.\lambda x.\textsc{likes}(x, y)$
likes
$NP : \textsc{brit}$
Brit

Figure 12.2: Derivation of the semantic representation for *Alex likes Brit* in the grammar G_1.

S	$\rightarrow$ NP VP	VP.sem@NP.sem
VP	$\rightarrow$ V_t NP	V_t.sem@NP.sem
VP	$\rightarrow$ V_i	V_i.sem
V_t	$\rightarrow$ *likes*	$\lambda y.\lambda x.\textsc{likes}(x, y)$
V_i	$\rightarrow$ *sleeps*	$\lambda x.\textsc{sleeps}(x)$
NP	$\rightarrow$ *Alex*	ALEX
NP	$\rightarrow$ *Brit*	BRIT

Table 12.1: G_1, a minimal syntactic-semantic context-free grammar

which has the meaning $\exists x \textsc{dog}(x) \wedge \textsc{sleeps}(x)$. Clearly, the DOG relation will be introduced by the word *dog*, and the SLEEP relation will be introduced by the word *sleeps*. The existential quantifier $\exists$ must be introduced by the lexical entry for the determiner *a*.[7] However, this seems problematic for the compositional approach taken in the grammar G_1: if the semantics of the noun phrase *a dog* is an existentially quantified expression, how can it be the argument to the semantics of the verb *sleeps*, which expects an entity? And where does the logical conjunction come from?

There are a few different approaches to handling these issues.[8] We will begin by reversing the semantic relationship between subject NPs and VPs, so that the production S $\rightarrow$ NP VP has the semantics NP.sem@VP.sem: the meaning of the sentence is now the semantics of the noun phrase applied to the verb phrase. The implications of this change are best illustrated by exploring the derivation of the example, shown in Figure 12.3. Let's

[7]Conversely, the sentence *Every dog sleeps* would involve a universal quantifier, $\forall x \textsc{dog}(x) \Rightarrow \textsc{sleeps}(x)$. The definite article *the* requires more consideration, since *the dog* must refer to some dog which is uniquely identifiable, perhaps from contextual information external to the sentence. Carpenter (1997, pp. 96-100) summarizes recent approaches to handling definite descriptions.

[8]Carpenter (1997) offers an alternative treatment based on combinatory categorial grammar.

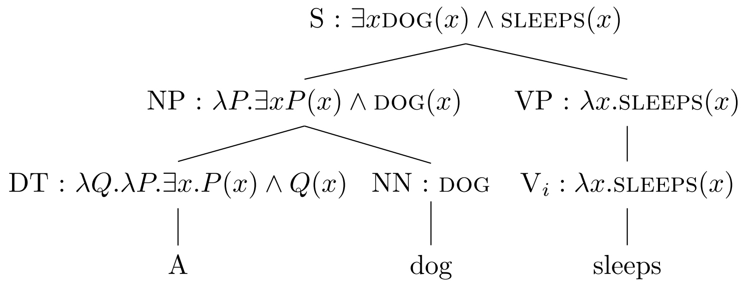

S : $\exists x \textsc{dog}(x) \wedge \textsc{sleeps}(x)$
NP : $\lambda P.\exists x P(x) \wedge \textsc{dog}(x)$
VP : $\lambda x.\textsc{sleeps}(x)$
DT : $\lambda Q.\lambda P.\exists x.P(x) \wedge Q(x)$
NN : $\textsc{dog}$
V_i : $\lambda x.\textsc{sleeps}(x)$
A
dog
sleeps

Figure 12.3: Derivation of the semantic representation for *A dog sleeps*, in grammar G_2

start with the indefinite article *a*, to which we assign the rather intimidating semantics,

$$\lambda P.\lambda Q.\exists x P(x) \wedge Q(x). \tag{12.15}$$

This is a lambda expression that takes two **relations** as arguments, P and Q. The relation P is scoped to the outer lambda expression, so it will be provided by the immediately adjacent noun, which in this case is $\textsc{dog}$. Thus, the noun phrase *a dog* has the semantics,

$$\begin{aligned} \text{NP.sem} =& \text{DET.sem}@\text{NN.sem} & [12.16] \\ =& (\lambda P.\lambda Q.\exists x P(x) \wedge Q(x))@(\textsc{dog}) & [12.17] \\ =& \lambda Q.\exists x \textsc{dog}(x) \wedge Q(x). & [12.18] \end{aligned}$$

This is a lambda expression that is expecting another relation, Q, which will be provided by the verb phrase, $\textsc{sleeps}$. This gives the desired analysis, $\exists x \textsc{dog}(x) \wedge \textsc{sleeps}(x)$.[9]

If noun phrases like *a dog* are interpreted as lambda expressions, then proper nouns like *Alex* must be treated in the same way. This is achieved by **type-raising** from constants to lambda expressions, $x \Rightarrow \lambda P.P(x)$. After type-raising, the semantics of *Alex* is $\lambda P.P(\textsc{alex})$ — a lambda expression that expects a relation to tell us something about $\textsc{alex}$.[10] Again, make sure you see how the analysis in Figure 12.3 can be applied to the sentence *Alex sleeps*.

[9]When applying β-reduction to arguments that are themselves lambda expressions, be sure to use unique variable names to avoid confusion. For example, it is important to distinguish the x in the semantics for *a* from the x in the semantics for *likes*. Variable names are abstractions, and can always be changed — this is known as α-**conversion**. For example, $\lambda x.P(x)$ can be converted to $\lambda y.P(y)$, etc.

[10]Compositional semantic analysis is often supported by **type systems**, which make it possible to check whether a given function application is valid. The base types are entities e and truth values t. A property, such as $\textsc{dog}$, is a function from entities to truth values, so its type is written $\langle e, t\rangle$. A transitive verb has type $\langle e, \langle e, t\rangle\rangle$: after receiving the first entity (the direct object), it returns a function from entities to truth values, which will be applied to the subject of the sentence. The type-raising operation $x \Rightarrow \lambda P.P(x)$ corresponds to a change in type from e to $\langle\langle e, t\rangle, t\rangle$: it expects a function from entities to truth values, and returns a truth value.

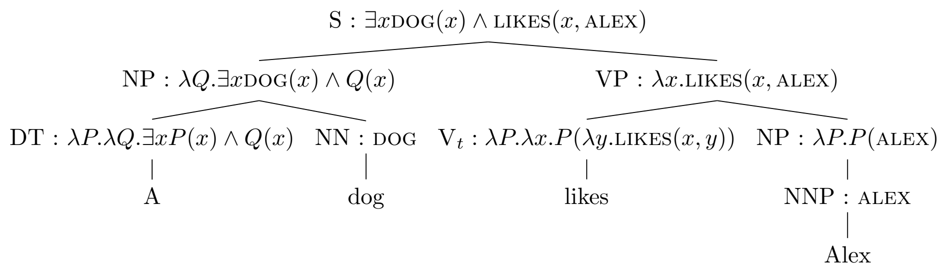

S : $\exists x \text{DOG}(x) \wedge \text{LIKES}(x, \text{ALEX})$
NP : $\lambda Q.\exists x \text{DOG}(x) \wedge Q(x)$
VP : $\lambda x.\text{LIKES}(x, \text{ALEX})$
DT : $\lambda P.\lambda Q.\exists x P(x) \wedge Q(x)$
NN : DOG
V_t : $\lambda P.\lambda x.P(\lambda y.\text{LIKES}(x, y))$
NP : $\lambda P.P(\text{ALEX})$
A
dog
likes
NNP : ALEX
Alex

Figure 12.4: Derivation of the semantic representation for *A dog likes Alex*.

Direct objects are handled by applying the same type-raising operation to transitive verbs: the meaning of verbs such as *likes* is raised to,

$$\lambda P.\lambda x.P(\lambda y.\text{LIKES}(x, y)) \qquad [12.19]$$

As a result, we can keep the verb phrase production VP.sem = V.sem@NP.sem, knowing that the direct object will provide the function P in Equation 12.19. To see how this works, let's analyze the verb phrase *likes a dog*. After uniquely relabeling each lambda variable,

$$\begin{aligned}
\text{VP.sem} =& \text{V.sem@NP.sem} \\
=& (\lambda P.\lambda x.P(\lambda y.\text{LIKES}(x, y)))@(\lambda Q.\exists z \text{DOG}(z) \wedge Q(z)) \\
=& \lambda x.(\lambda Q.\exists z \text{DOG}(z) \wedge Q(z))@(\lambda y.\text{LIKES}(x, y)) \\
=& \lambda x.\exists z \text{DOG}(z) \wedge (\lambda y.\text{LIKES}(x, y))@z \\
=& \lambda x.\exists z \text{DOG}(z) \wedge \text{LIKES}(x, z).
\end{aligned}$$

These changes are summarized in the revised grammar G_2, shown in Table 12.2. Figure 12.4 shows a derivation that involves a transitive verb, an indefinite noun phrase, and a proper noun.

12.4 Learning semantic parsers

As with syntactic parsing, any syntactic-semantic grammar with sufficient coverage risks producing many possible analyses for any given sentence. Machine learning is the dominant approach to selecting a single analysis. We will focus on algorithms that learn to score logical forms by attaching weights to features of their derivations (Zettlemoyer and Collins, 2005). Alternative approaches include transition-based parsing (Zelle and Mooney, 1996; Misra and Artzi, 2016) and methods inspired by machine translation (Wong and Mooney, 2006). Methods also differ in the form of supervision used for learning, which can range from complete derivations to much more limited training signals. We will begin with the case of complete supervision, and then consider how learning is still possible even when seemingly key information is missing.

S	$\rightarrow$ NP VP	NP.sem@VP.sem
VP	$\rightarrow$ V$_t$ NP	V$_t$.sem@NP.sem
VP	$\rightarrow$ V$_i$	V$_i$.sem
NP	$\rightarrow$ DET NN	DET.sem@NN.sem
NP	$\rightarrow$ NNP	$\lambda P.P($NNP.sem$)$
DET	$\rightarrow$ *a*	$\lambda P.\lambda Q.\exists x P(x) \wedge Q(x)$
DET	$\rightarrow$ *every*	$\lambda P.\lambda Q.\forall x(P(x) \Rightarrow Q(x))$
V$_t$	$\rightarrow$ *likes*	$\lambda P.\lambda x.P(\lambda y.\text{LIKES}(x,y))$
V$_i$	$\rightarrow$ *sleeps*	$\lambda x.\text{SLEEPS}(x)$
NN	$\rightarrow$ *dog*	DOG
NNP	$\rightarrow$ *Alex*	ALEX
NNP	$\rightarrow$ *Brit*	BRIT

Table 12.2: G_2, a syntactic-semantic context-free grammar fragment, which supports quantified noun phrases

Datasets Early work on semantic parsing focused on natural language expressions of geographical database queries, such as *What states border Texas*. The GeoQuery dataset of Zelle and Mooney (1996) was originally coded in prolog, but has subsequently been expanded and converted into the SQL database query language by Popescu et al. (2003) and into first-order logic with lambda calculus by Zettlemoyer and Collins (2005), providing logical forms like $\lambda x.\text{STATE}(x) \wedge \text{BORDERS}(x, \text{TEXAS})$. Another early dataset consists of instructions for RoboCup robot soccer teams (Kate et al., 2005). More recent work has focused on broader domains, such as the Freebase database (Bollacker et al., 2008), for which queries have been annotated by Krishnamurthy and Mitchell (2012) and Cai and Yates (2013). Other recent datasets include child-directed speech (Kwiatkowski et al., 2012) and elementary school science exams (Krishnamurthy, 2016).

12.4.1 Learning from derivations

Let $\boldsymbol{w}^{(i)}$ indicate a sequence of text, and let $\boldsymbol{y}^{(i)}$ indicate the desired logical form. For example:

$$\boldsymbol{w}^{(i)} = \text{Alex eats shoots and leaves}$$
$$\boldsymbol{y}^{(i)} = \text{EATS(ALEX,SHOOTS)} \wedge \text{EATS(ALEX,LEAVES)}$$

In the standard supervised learning paradigm that was introduced in § 2.3, we first define a feature function, $\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y})$, and then learn weights on these features, so that $\boldsymbol{y}^{(i)} = \text{argmax}_{\boldsymbol{y}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y})$. The weight vector $\boldsymbol{\theta}$ is learned by comparing the features of the true label $\boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$ against either the features of the predicted label $\boldsymbol{f}(\boldsymbol{w}^{(i)}, \hat{\boldsymbol{y}})$ (perceptron,

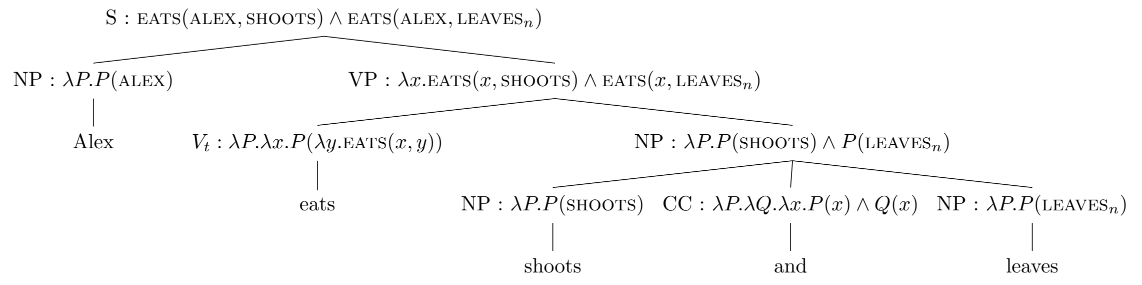

S : EATS(ALEX, SHOOTS) ∧ EATS(ALEX, LEAVES$_n$)
NP : $\lambda P.P$(ALEX)
VP : $\lambda x.$EATS$(x,$ SHOOTS$)$ ∧ EATS$(x,$ LEAVES$_n)$
Alex
V_t : $\lambda P.\lambda x.P(\lambda y.$EATS$(x, y))$
NP : $\lambda P.P$(SHOOTS) ∧ P(LEAVES$_n$)
eats
NP : $\lambda P.P$(SHOOTS)
CC : $\lambda P.\lambda Q.\lambda x.P(x) \wedge Q(x)$
NP : $\lambda P.P$(LEAVES$_n$)
shoots
and
leaves

Figure 12.5: Derivation for gold semantic analysis of *Alex eats shoots and leaves*

support vector machine) or the expected feature vector $E_{\boldsymbol{y}|\boldsymbol{w}}[\boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y})]$ (logistic regression).

While this basic framework seems similar to discriminative syntactic parsing, there is a crucial difference. In (context-free) syntactic parsing, the annotation $\boldsymbol{y}^{(i)}$ contains all of the syntactic productions; indeed, the task of identifying the correct set of productions is identical to the task of identifying the syntactic structure. In semantic parsing, this is not the case: the logical form EATS(ALEX,SHOOTS) ∧ EATS(ALEX,LEAVES) does not reveal the syntactic-semantic productions that were used to obtain it. Indeed, there may be **spurious ambiguity**, so that a single logical form can be reached by multiple derivations. (We previously encountered spurious ambiguity in transition-based dependency parsing, § 11.3.2.)

These ideas can be formalized by introducing an additional variable $\boldsymbol{z}$, representing the derivation of the logical form $\boldsymbol{y}$ from the text $\boldsymbol{w}$. Assume that the feature function decomposes across the productions in the derivation, $\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}) = \sum_{t=1}^{T} \boldsymbol{f}(\boldsymbol{w}, z_t, \boldsymbol{y})$, where z_t indicates a single syntactic-semantic production. For example, we might have a feature for the production S → NP VP : NP.sem@VP.sem, as well as for terminal productions like NNP → *Alex* : ALEX. Under this decomposition, it is possible to compute scores for each semantically-annotated subtree in the analysis of $\boldsymbol{w}$, so that bottom-up parsing algorithms like CKY (§ 10.1) can be applied to find the best-scoring semantic analysis.

Figure 12.5 shows a derivation of the correct semantic analysis of the sentence *Alex eats shoots and leaves*, in a simplified grammar in which the plural noun phrases *shoots* and *leaves* are interpreted as logical constants SHOOTS and LEAVES$_n$. Figure 12.6 shows a derivation of an incorrect analysis. Assuming one feature per production, the perceptron update is shown in Table 12.3. From this update, the parser would learn to prefer the noun interpretation of *leaves* over the verb interpretation. It would also learn to prefer noun phrase coordination over verb phrase coordination.

While the update is explained in terms of the perceptron, it would be easy to replace the perceptron with a conditional random field. In this case, the online updates would be

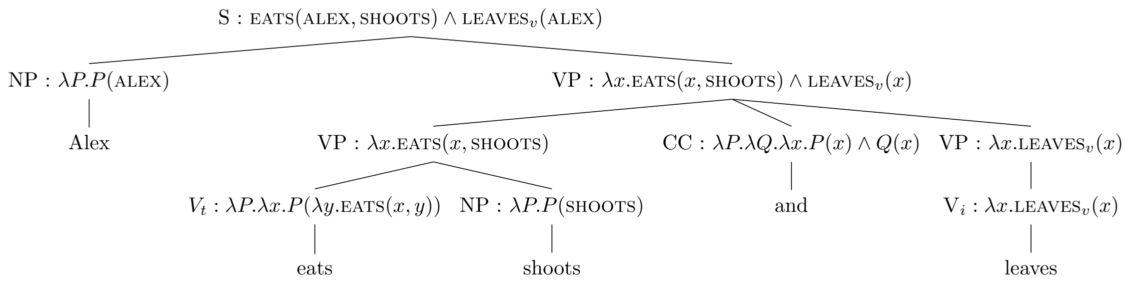

S : EATS(ALEX, SHOOTS) ∧ LEAVES$_v$(ALEX)
NP : $\lambda P.P$(ALEX)
Alex
VP : λx.EATS(x, SHOOTS) ∧ LEAVES$_v$(x)
VP : λx.EATS(x, SHOOTS)
V_t : $\lambda P.\lambda x.P(\lambda y$.EATS($x,y$))
eats
NP : $\lambda P.P$(SHOOTS)
shoots
CC : $\lambda P.\lambda Q.\lambda x.P(x) \wedge Q(x)$
and
VP : λx.LEAVES$_v$(x)
V$_i$: λx.LEAVES$_v$(x)
leaves

Figure 12.6: Derivation for incorrect semantic analysis of *Alex eats shoots and leaves*

$\text{NP}_1 \rightarrow \text{NP}_2\ \text{CC}\ \text{NP}_3$	$(\text{CC.sem}@(\text{NP}_2.\text{sem}))@(\text{NP}_3.\text{sem})$	+1
$\text{VP}_1 \rightarrow \text{VP}_2\ \text{CC}\ \text{VP}_3$	$(\text{CC.sem}@(\text{VP}_2.\text{sem}))@(\text{VP}_3.\text{sem})$	-1
NP → *leaves*	LEAVES_n	+1
$\text{VP} \rightarrow \text{V}_i$	$\text{V}_i.\text{sem}$	-1
$\text{V}_i \rightarrow$ *leaves*	$\lambda x.\text{LEAVES}_v$	-1

Table 12.3: Perceptron update for analysis in Figure 12.5 (gold) and Figure 12.6 (predicted)

based on feature expectations, which can be computed using the inside-outside algorithm (§ 10.6).

12.4.2 Learning from logical forms

Complete derivations are expensive to annotate, and are rarely available.[11] One solution is to focus on learning from logical forms directly, while treating the derivations as **latent variables** (Zettlemoyer and Collins, 2005). In a conditional probabilistic model over logical forms $\boldsymbol{y}$ and derivations $\boldsymbol{z}$, we have,

$$\text{p}(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{w}) = \frac{\exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}))}{\sum_{\boldsymbol{y}', \boldsymbol{z}'} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}', \boldsymbol{y}'))}, \qquad [12.20]$$

which is the standard log-linear model, applied to the logical form $\boldsymbol{y}$ and the derivation $\boldsymbol{z}$.

Since the derivation $\boldsymbol{z}$ unambiguously determines the logical form $\boldsymbol{y}$, it may seem silly to model the joint probability over $\boldsymbol{y}$ and $\boldsymbol{z}$. However, since $\boldsymbol{z}$ is unknown, it can be marginalized out,

$$\text{p}(\boldsymbol{y} \mid \boldsymbol{w}) = \sum_{\boldsymbol{z}} \text{p}(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{w}). \qquad [12.21]$$

[11] An exception is the work of Ge and Mooney (2005), who annotate the meaning of each syntactic constituents for several hundred sentences.

The semantic parser can then select the logical form with the maximum log marginal probability,

$$\log \sum_{\boldsymbol{z}} \mathrm{p}(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{w}) = \log \sum_{\boldsymbol{z}} \frac{\exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}))}{\sum \boldsymbol{y}', \boldsymbol{z}' \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}', \boldsymbol{y}'))} \qquad [12.22]$$

$$\propto \log \sum_{\boldsymbol{z}} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}', \boldsymbol{y}')) \qquad [12.23]$$

$$\geq \max_{\boldsymbol{z}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}). \qquad [12.24]$$

It is impossible to push the $\log$ term inside the sum over $\boldsymbol{z}$, so our usual linear scoring function does not apply. We can recover this scoring function only in approximation, by taking the max (rather than the sum) over derivations $\boldsymbol{z}$, which provides a lower bound.

Learning can be performed by maximizing the log marginal likelihood,

$$\ell(\boldsymbol{\theta}) = \sum_{i=1}^{N} \log \mathrm{p}(\boldsymbol{y}^{(i)} \mid \boldsymbol{w}^{(i)}; \boldsymbol{\theta}) \qquad [12.25]$$

$$= \sum_{i=1}^{N} \log \sum_{\boldsymbol{z}} \mathrm{p}(\boldsymbol{y}^{(i)}, \boldsymbol{z}^{(i)} \mid \boldsymbol{w}^{(i)}; \boldsymbol{\theta}). \qquad [12.26]$$

This log-likelihood is not **convex** in $\boldsymbol{\theta}$, unlike the log-likelihood of a fully-observed conditional random field. This means that learning can give different results depending on the initialization.

The derivative of Equation 12.26 is,

$$\frac{\partial \ell_i}{\partial \boldsymbol{\theta}} = \sum_{\boldsymbol{z}} \mathrm{p}(\boldsymbol{z} \mid \boldsymbol{y}, \boldsymbol{w}; \boldsymbol{\theta}) \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}) - \sum_{\boldsymbol{y}', \boldsymbol{z}'} \mathrm{p}(\boldsymbol{y}', \boldsymbol{z}' \mid \boldsymbol{w}; \boldsymbol{\theta}) \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}', \boldsymbol{y}') \qquad [12.27]$$

$$= E_{\boldsymbol{z} \mid \boldsymbol{y}, \boldsymbol{w}} \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}) - E_{\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{w}} \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}) \qquad [12.28]$$

Both expectations can be computed via bottom-up algorithms like inside-outside. Alternatively, we can again maximize rather than marginalize over derivations for an approximate solution. In either case, the first term of the gradient requires us to identify derivations $\boldsymbol{z}$ that are compatible with the logical form $\boldsymbol{y}$. This can be done in a bottom-up dynamic programming algorithm, by having each cell in the table $t[i, j, X]$ include the set of all possible logical forms for $X \rightsquigarrow \boldsymbol{w}_{i+1:j}$. The resulting table may therefore be much larger than in syntactic parsing. This can be controlled by using pruning to eliminate intermediate analyses that are incompatible with the final logical form $\boldsymbol{y}$ (Zettlemoyer and Collins, 2005), or by using beam search and restricting the size of each cell to some fixed constant (Liang et al., 2013).

If we replace each expectation in Equation 12.28 with argmax and then apply stochastic gradient descent to learn the weights, we obtain the **latent variable perceptron**, a simple

Algorithm 16 Latent variable perceptron

```
1: procedure LatentVariablePerceptron($\boldsymbol{w}^{(1:N)}, \boldsymbol{y}^{(1:N)}$)
2:     $\boldsymbol{\theta} \leftarrow \boldsymbol{0}$
3:     repeat
4:         Select an instance $i$
5:         $\boldsymbol{z}^{(i)} \leftarrow \text{argmax}_{\boldsymbol{z}}\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{z}, \boldsymbol{y}^{(i)})$
6:         $\hat{\boldsymbol{y}}, \hat{\boldsymbol{z}} \leftarrow \text{argmax}_{\boldsymbol{y}', \boldsymbol{z}'}\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{z}', \boldsymbol{y}')$
7:         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{z}^{(i)}, \boldsymbol{y}^{(i)}) - \boldsymbol{f}(\boldsymbol{w}^{(i)}, \hat{\boldsymbol{z}}, \hat{\boldsymbol{y}})$
8:     until tired
9:     return $\boldsymbol{\theta}$
```

and general algorithm for learning with missing data. The algorithm is shown in its most basic form in Algorithm 16, but the usual tricks such as averaging and margin loss can be applied (Yu and Joachims, 2009). Aside from semantic parsing, the latent variable perceptron has been used in tasks such as machine translation (Liang et al., 2006) and named entity recognition (Sun et al., 2009). In **latent conditional random fields**, we use the full expectations rather than maximizing over the hidden variable. This model has also been employed in a range of problems beyond semantic parsing, including parse reranking (Koo and Collins, 2005) and gesture recognition (Quattoni et al., 2007).

12.4.3 Learning from denotations

Logical forms are easier to obtain than complete derivations, but the annotation of logical forms still requires considerable expertise. However, it is relatively easy to obtain denotations for many natural language sentences. For example, in the geography domain, the denotation of a question would be its answer (Clarke et al., 2010; Liang et al., 2013):

$$\begin{aligned}
\textbf{Text} :& \textit{What states border Georgia?} \\
\textbf{Logical form} :& \lambda x.\textsc{state}(x) \wedge \textsc{border}(x, \textsc{georgia}) \\
\textbf{Denotation} :& \{\texttt{Alabama, Florida, North Carolina,} \\
& \texttt{South Carolina, Tennessee}\}
\end{aligned}$$

Similarly, in a robotic control setting, the denotation of a command would be an action or sequence of actions (Artzi and Zettlemoyer, 2013). In both cases, the idea is to reward the semantic parser for choosing an analysis whose denotation is correct: the right answer to the question, or the right action.

Learning from logical forms was made possible by summing or maxing over derivations. This idea can be carried one step further, summing or maxing over all logical forms with the correct denotation. Let $v_i(\boldsymbol{y}) \in \{0, 1\}$ be a **validation function**, which assigns a binary score indicating whether the denotation $[\![\boldsymbol{y}]\!]$ for the text $\boldsymbol{w}^{(i)}$ is correct. We can then learn by maximizing a conditional-likelihood objective,

$$\ell^{(i)}(\boldsymbol{\theta}) = \log \sum_{\boldsymbol{y}} v_i(\boldsymbol{y}) \times \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) \tag{12.29}$$

$$= \log \sum_{\boldsymbol{y}} v_i(\boldsymbol{y}) \times \sum_{\boldsymbol{z}} \mathrm{p}(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{w}; \boldsymbol{\theta}), \tag{12.30}$$

which sums over all derivations $\boldsymbol{z}$ of all valid logical forms, $\{\boldsymbol{y} : v_i(\boldsymbol{y}) = 1\}$. This corresponds to the log-probability that the semantic parser produces a logical form with a valid denotation.

Differentiating with respect to $\boldsymbol{\theta}$, we obtain,

$$\frac{\partial \ell^{(i)}}{\partial \boldsymbol{\theta}} = \sum_{\boldsymbol{y},\boldsymbol{z}:v_i(\boldsymbol{y})=1} \mathrm{p}(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{w}) \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}) - \sum_{\boldsymbol{y}',\boldsymbol{z}'} \mathrm{p}(\boldsymbol{y}', \boldsymbol{z}' \mid \boldsymbol{w}) \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}', \boldsymbol{y}'), \tag{12.31}$$

which is the usual difference in feature expectations. The positive term computes the expected feature expectations conditioned on the denotation being valid, while the second term computes the expected feature expectations according to the current model, without regard to the ground truth. Large-margin learning formulations are also possible for this problem. For example, Artzi and Zettlemoyer (2013) generate a set of valid and invalid derivations, and then impose a constraint that all valid derivations should score higher than all invalid derivations. This constraint drives a perceptron-like learning rule.

Additional resources

A key issue not considered here is how to handle **semantic underspecification**: cases in which there are multiple semantic interpretations for a single syntactic structure. Quantifier scope ambiguity is a classic example. Blackburn and Bos (2005) enumerate a number of approaches to this issue, and also provide links between natural language semantics and computational inference techniques. Much of the contemporary research on semantic parsing uses the framework of combinatory categorial grammar (CCG). Carpenter (1997) provides a comprehensive treatment of how CCG can support compositional semantic analysis. Another recent area of research is the semantics of multi-sentence texts. This can be handled with models of **dynamic semantics**, such as dynamic predicate logic (Groenendijk and Stokhof, 1991).

Alternative readings on formal semantics include an "informal" reading from Levy and Manning (2009), and a more involved introduction from Briscoe (2011). To learn more about ongoing research on data-driven semantic parsing, readers may consult the survey article by Liang and Potts (2015), tutorial slides and videos by Artzi and Zettlemoyer (2013),[12] and the source code by Yoav Artzi[13] and Percy Liang.[14]

Exercises

1. The **modus ponens** inference rule states that if we know $\phi \Rightarrow \psi$ and ϕ, then ψ must be true. Justify this rule, using the definition of the $\Rightarrow$ operator and some of the laws provided in § 12.2.1, plus one additional identity: $\bot \vee \phi = \phi$.

2. Convert the following examples into first-order logic, using the relations CAN-SLEEP, MAKES-NOISE, and BROTHER.

   - If Abigail makes noise, no one can sleep.
   - If Abigail makes noise, someone cannot sleep.
   - None of Abigail's brothers can sleep.
   - If one of Abigail's brothers makes noise, Abigail cannot sleep.

3. Extend the grammar fragment G_1 to include the ditransitive verb *teaches* and the proper noun *Swahili*. Show how to derive the interpretation for the sentence *Alex teaches Brit Swahili*, which should be TEACHES(ALEX,BRIT,SWAHILI). The grammar need not be in Chomsky Normal Form. For the ditransitive verb, use NP_1 and NP_2 to indicate the two direct objects.

4. Derive the semantic interpretation for the sentence *Alex likes every dog*, using grammar fragment G_2.

5. Extend the grammar fragment G_2 to handle adjectives, so that the meaning of *an angry dog* is $\lambda P.\exists x \text{DOG}(x) \wedge \text{ANGRY}(x) \wedge P(x)$. Specifically, you should supply the lexical entry for the adjective *angry*, and you should specify the syntactic-semantic productions NP $\rightarrow$ DET NOM, NOM $\rightarrow$ JJ NOM, and NOM $\rightarrow$ NN.

6. Extend your answer to the previous question to cover copula constructions with predicative adjectives, such as *Alex is angry*. The interpretation should be ANGRY(ALEX). You should add a verb phrase production VP $\rightarrow$ V_{cop} JJ, and a terminal production $\text{V}_{\text{cop}} \rightarrow$ *is*. Show why your grammar extensions result in the correct interpretation.

7. In Figure 12.5 and Figure 12.6, we treat the plurals *shoots* and *leaves* as entities. Revise G_2 so that the interpretation of *Alex eats leaves* is $\forall x.(\text{LEAF}(x) \Rightarrow \text{EATS}(\text{ALEX}, x))$, and show the resulting perceptron update.

[12]Videos are currently available at `http://yoavartzi.com/tutorial/`
[13]`http://yoavartzi.com/spf`
[14]`https://github.com/percyliang/sempre`

8. Statements like *every student eats a pizza* have two possible interpretations, depending on quantifier scope:

$$\forall x \exists y \textsc{pizza}(y) \wedge (\textsc{student}(x) \Rightarrow \textsc{eats}(x, y)) \quad [12.32]$$
$$\exists y \forall x \textsc{pizza}(y) \wedge (\textsc{student}(x) \Rightarrow \textsc{eats}(x, y)) \quad [12.33]$$

   a) Explain why these interpretations really are different.
   b) Which is generated by grammar G_2? Note that you may have to manipulate the logical form to exactly align with the grammar.

9. *Modify G_2 so that produces the second interpretation in the previous problem. **Hint**: one possible solution involves changing the semantics of the sentence production and one other production.

10. In the GeoQuery domain, give a natural language query that has multiple plausible semantic interpretations with the same denotation. List both interpretaions and the denotation.

    Hint: There are many ways to do this, but one approach involves using toponyms (place names) that could plausibly map to several different entities in the model.