An Introduction to Formal Computational Semantics

CS224N/Ling 280 – Christopher Manning

May 23, 2000; revised 2005

1 Motivation

We have seen ways to extract and represent elements of meaning in statistical and other quantitative ways (word sense disambiguation, semantic similarity, information extraction templates). But there are many NLP domains, such as automated question answering, dialog systems, story understanding, and automated knowledge base construction, where we need to deal with the meanings of sentences in a more sophisticated and precise (yet robust) way. Characterizing what we mean by *meaning* is a difficult philosophical issue, but the only thing implementation-ready as a system of representation seems to be the tradition of logical semantics, appropriately extended to natural language (Montague 1973). I believe that a promising approach for many domain-embedded applications is to use the benefits of statistical models for disambiguation at a lexical/syntactic level, and then to use logical semantic representations for detailed interpretations.

In computational semantics, we can isolate the following questions:

1. How can we automate the process of associating semantic representations with expressions of natural language?

2. How can we use semantic representations of natural language expressions in the process of drawing inferences?

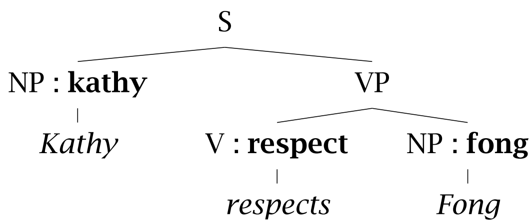
We will concentrate on the first goal. This is for two reasons. Firstly, this issue is normally ignored in logic or knowledge representation and reasoning classes, whereas the second is covered in depth. Secondly, it's the mapping from language to a meaning representation that is most interesting. One way to do inference with a meaning representation is using logical proofs, but we can also explore quite different methods such as probabilistic models.

This presentation attempts to be self-contained, but moves quickly and assumes that you've seen some logic before. It covers some ideas that you don't normally see in a quarter course on logic/semantics.

2 Compositionality, rule-to-rule translation, and the need for λ-calculus

We'll start with a version of First-Order Predicate Calculus (FOPC), which contains constants like **fong** and **kathy** (which we'll sometimes abbreviate as **f** and **k**), variables ranging over constants like x and y, and some n-ary functions like **respect**, which evaluate to truth values (**0** or **1**, as in C).

We want to have a meaning representation for a sentence like *Kathy likes Fong*. Just like in a logic class, we'll want the meaning representation to be **likes**(**k**, **f**). where **k** and **f** are *Kathy* and *Fong*, respectively, and **respect** captures the respecting relation. Now, if we give those meanings to the words concerned, things look kind of close:



But how do we work out the meaning of the whole sentence from the words? And how do we know that it comes out as **respect**(**k**, **f**), and not **respect**(**f**, **k**) – or even **respect**(**k**, **f**) $\wedge$ **respect**(**f**, **k**)?

Our goal is a system to build up a logical proposition out of the components of the sentence. We exploit the fact that the syntax and the semantics are actually pretty parallel in defining a rule-to-rule translation. Consider:

(1) a. A bookshop is across the street.

b. A strange man is across the street.

The syntactic observation here is that *[A bookshop]* and *[A strange man]* are interchangeable without affecting the grammaticality of the sentence. Due in part to this observation, syntacticians posit that the two sequences of words are *constituents* of a common type–in this case, NP. The semantic observation is that the shared parts of these two sentences seem to have a common meaning, and that the contribution made by *[A bookshop]* and *[A strange man]* to sentence meaning is parallel. This first idea is the *Principle of compositionality* (usually attributed to Frege): The meaning of an expression is determined by the meanings of its parts and the way in which they are combined. Among other things, this means that knowing the words is a good start.

But we need to provide more structure as to the way the meanings of parts are combined. What we'll do to capture this is to define a semantics in parallel to a syntax. In every syntactic rule, each category will be annotated with a semantic value, (typically) determining how the semantics of the mother is built out of the semantics of the daughters. These notations are called *semantic attachments*, and the method of building up a compositional semantics using semantic attachments of this sort is referred to as the idea of *rule-to-rule translation* (Bach 1976).

The third thing we'll need is a secret weapon that gives us an algebra for meaning assembly. It's called the λ-calculus (Church 1940), and you may have seen it in the more enlightened functional programming languages. It allows us to "hold out" positions within a FOPC expression and to "fill" them later, by applying the new expression to another expression. The first fundamental rule to remember for λ-formulae is the idea of function application.

$$(\lambda x.P(\cdots,x,\cdots))(Z) \quad \Rightarrow \quad P(\cdots,Z,\cdots) \qquad \beta \text{ reduction [application]}$$

This just says that you can apply an argument to a lambda function, and it gets substituted in. When doing application, you have to make sure in the conversion rules that variable names don't accidentally overlap, but I'll leave this informal.

3 A first example

Putting this together, we'll have lexical items and a grammar that look something like this:

Lexicon	**Grammar**
Kathy, NP : **kathy**	S : $\beta(\alpha) \to$ NP : α VP : β
Fong, NP : **fong**	VP : $\beta(\alpha) \to$ V : β NP : α
respects, V : $\lambda y.\lambda x.\mathbf{respect}(x, y)$	VP : $\beta \to$ V : β
runs, V : $\lambda x.\mathbf{run}(x)$	

Here the ':' is read 'means'. In the lexicon, note the reversal of the arguments of *likes*, since it will combine first with its object in the syntax. In the syntax, the phrase structure rules show how to combine meanings; here, just by function application. In general, head-nonhead syntactic composition corresponds to functional application. However, in certain cases, semantic heads deviate from syntactic heads. This occurs principally with determiners, and modifiers (adjectives and adverbs). We will see examples below.

With this syntax & semantics, we get a logical proposition (something whose meaning is a truth-value) for the root node S of the sentence. Here's one of the 6 sentences in the language:

- S : $\mathbf{respect}(\mathbf{kathy}, \mathbf{fong})$
  - NP : $\mathbf{kathy}$
    - *Kathy*
  - VP : $\lambda x.\mathbf{respect}(x, \mathbf{fong})$
    - V : $\lambda y.\lambda x.\mathbf{respect}(x, y)$
      - *respects*
    - NP : $\mathbf{fong}$
      - *Fong*

We have:

$$[_{\text{VP}} \text{ respects Fong}] : [\lambda y.\lambda x.\mathbf{respect}(x, y)](\mathbf{fong}) = \lambda x.\mathbf{respect}(x, \mathbf{fong}) \qquad [\beta \text{ red.}]$$

$$[_{\text{S}} \text{ Kathy respects Fong}] : [\lambda x.\mathbf{respect}(x, \mathbf{fong})](\mathbf{kathy}) = \mathbf{respect}(\mathbf{kathy}, \mathbf{fong})$$

If the semantics always precisely followed the syntax, then it would be this easy (and sometimes this is all introductory textbooks show). But, as we will see below, there are also many constructions where sentence meaning and its compositional units *differs* from syntactic units. At this point you have a choice: to complicate the syntax with extraneous stuff which will allow you to more easily generate a parallel semantics, or else to add more combinatory power to the formulae that generate semantic representations, so that correct semantic representations can be derived without complicating the syntax. We lean towards the latter approach.

4 Some more formal stuff

But before we get on to the hard part, let us just touch on more technical issues of proof theory, model theory, and typing.

4.1 Typed λ calculus (Church 1940)

Once we get into more complicated stuff below (that is, higher order terms), in order to keep track of things (and to avoid certain paradoxes), we will use a typed λ calculus. This means each semantic value will have a *type* (just like the strong typing in most current programming languages). The result is an ω-order logic (which just means that you can do any finite order quantification over types). Then we have λ terms of various types. There are two basic types, **Bool** = truth values and **Ind** = individuals, which correspond to the type of *Kathy* or *Fong*, above. (Some proposals use more basic types, but we'll stick to just those two.) Other types are made up as higher-order functional types out of simpler types. So we have:

Bool	truth values (**0** and **1**)
Ind	individuals
Ind → **Bool**	properties
Ind → **Ind** → **Bool**	binary relations

A type like the last is to be interpreted right associatively as **Ind** → (**Ind** → **Bool**) – as in the simple example, above, we standardly convert a several argument function into embedded unary functions. This is referred to as *currying* them. From now on our various semantic forms have a definite type:

k and **f** are **Ind**
run is **Ind** → **Bool**
respect is **Ind** → **Ind** → **Bool**

Once we have types, we don't need λ variables just to show what arguments something takes, and so we can introduce another operation of the λ calculus:

$$\lambda x.(P(x)) \quad \Rightarrow \quad P \qquad \eta \text{ reduction [abstractions can be contracted]}$$

This means that instead of writing:

$\lambda y.\lambda x.\mathbf{respect}(x, y)$

we can just write:

respect

You just have to *know* that **respect** is something of type **Ind** → **Ind** → **Bool**.

I'm not going to get into all the formal details, but a Church-Rosser theorem corollary is that the order of reductions of typed λ-calculus terms doesn't matter – and so there are usable proof-theoretic techniques (two terms are logically equivalent iff they are reducible to the same form). There are normal forms (built using η-reductions [removing lambdas] and α-reduction [renaming of variables]). The first form we introduced is called the β, η long form, and the second more compact representation (which we use quite a bit below) is called the β, η normal form. Here are a few more examples:

β, η normal form: **run**, **every**(**kid**, **run**), **yesterday**(**run**)
β, η long form: $\lambda x.\mathbf{run}(x)$, $\mathbf{every}(\lambda x.\mathbf{kid}(x))$, $(\lambda x.\mathbf{run}(x))$, $\lambda y.\mathbf{yesterday}(\lambda x.\mathbf{run}(x))(y)$

Our major syntactic categories will have corresponding types: nouns and verb phrases will be properties (**Ind** →**Bool**), noun phrases are **Ind** – though, we will see later that they are commonly type-raised to (**Ind** → **Bool**) → **Bool**, adjectives are (**Ind** → **Bool**) → (**Ind** → **Bool**). This is because adjectives modify noun meanings, that is properties. Intensifiers modify adjectives: e.g, *very* in *a very happy camper*, so they're ((**Ind** → **Bool**) → (**Ind** → **Bool**)) → ((**Ind** → **Bool**) → (**Ind** → **Bool**)) [honest!]. We will find that we can question and quantify over many of these higher-order types.

4.2 Logical Inference and Meaning Postulates

Once we have these predicate logic forms, we can do reasoning on them directly, using proof theory. Reasoning works pretty straightforwardly on the familiar propositional logic operators $\vee, \wedge, \Rightarrow, \neg$. For a system of lexical meaning representation, we might want to code in *meaning postulates* background knowledge relating predicates to each other. For example, we might argue that

$$\mathbf{admire(x)(y)} \Rightarrow \mathbf{respect(x)(y)}$$

or, perhaps less controversially, if **give(x)(y)(z)** means "Z gives X Y", and **give(y)(z)** means "Z gives X", then

$$\mathbf{give(x)(y)(z)} \Rightarrow \mathbf{give(y)(z)}$$

Such auxiliary facts are useful for doing inferences involving linguistic facts and facts about the world, for instance, in a question answering system.

4.3 Model-theoretic Interpretation

So far we've got a *compositional* system of semantics, but you might note that really all we have is a lot of fancy syntax. How do we actually ground things in meanings? We'll touch on that now. We'll make our semantics *truth conditional*. By *truth-conditional*, we mean that a sentence's meaning is what the world would have to be like in order for it to be true. It's difficult to prove formal theorems about the world, so we build a mathematic model of aspects of the world. This gives us a *model-theoretic* semantics (Tarski 1935): the logical language we use to build up meanings is distinguished from its *interpretation* in the domain, which is a mathematical model of "real-world" objects and substantive relations. A model-theoretic approach to natural language meaning was pioneered by Montague (1973). We seek meaning in a notion of model denotation, not simply by translating one symbol string into a different symbol string.

We build a *model* $\mathcal{M} = \langle \mathbf{Dom}, [\![\cdot]\!] \rangle$ as a pair of a frame that gives a domain for basic types (with functional types defined in the obvious manner), and an interpretation function $[\![\cdot]\!] : \mathbf{Con} \to \mathbf{Dom}$, which tells us the denotation of semantic expressions, that is, it tells us what constants in the semantic representation refer to in the model. The denotation of a constant is taken to be an *individual*, whereas the denotation of a function is taken to be a *set*, and the meaning of a sentence is a truth value (Boolean). An assignment function similarly tells us the meaning of variables.

For our example in the previous section, we might use the following model:

$$\begin{aligned}
&\mathbf{Dom} = \{k, f, b\}\\
&[\![\mathbf{kathy}]\!] = k\\
&[\![\mathbf{fong}]\!] = f\\
&[\![\mathbf{run}]\!] = \{f\}\\
&[\![\mathbf{respect}]\!] = \{(f,k), (k,k), (k,f), (b,f)\}
\end{aligned}$$

The domain has three individuals, and f is running, and there are 4 elements in a respect relation. The denotation of **fong** is f, which is an object in the model.

The truth value of an n-ary function applied to an ordered n-tuple of constants is **1** if the n-tuple is in the function's meaning; it is **0** otherwise. Assuming the above model, the sentence we built in the previous section has a truth value of **1**, since $(f,k) \in [\![\mathbf{respect}]\!]$. We can equivalently think of the denotation of a curried function like this:

$$[\![\mathbf{respect}]\!] = [\![\lambda y.\lambda x.\mathbf{respect}(x,y)]\!] = \begin{bmatrix} f \mapsto \begin{bmatrix} f \mapsto \mathbf{0} \\ k \mapsto \mathbf{1} \\ b \mapsto \mathbf{0} \end{bmatrix} \\ k \mapsto \begin{bmatrix} f \mapsto \mathbf{1} \\ k \mapsto \mathbf{1} \\ b \mapsto \mathbf{0} \end{bmatrix} \\ b \mapsto \begin{bmatrix} f \mapsto \mathbf{1} \\ b \mapsto \mathbf{0} \\ r \mapsto \mathbf{0} \end{bmatrix} \end{bmatrix}$$

$$[\![\lambda x.\lambda y.\mathbf{respect}(y)(x)(b)(f)]\!] = \mathbf{1}$$

A property, that is, a one-argument function, is an indicator function, which simply picks out a subset of the elements of the domain.

Exercise: Assuming f and b are kids, what would the function **kid** look like?

It is important to realize that this mapping is *not* trivial; it need not be 1–1, not even for the case of individuals. For example, it would be perfectly possible that $[\![\mathbf{kathy}]\!] = [\![\mathbf{fong}]\!] = k$; in the case that **kathy** and **fong** *were two different names for the same individual*. You can think of the values **kathy** and **fong** as being the more linguistic entities "that person who is referred to as Kathy (resp. Fong)." Much philosophical mileage may be gotten out of this distinction, and perhaps some computational mileage as well. The idea here would be to represent a database of knowledge in model-theoretic terms, and then evaluate the truth value of statements/queries (see a later section) directly against this database. Lots of times, multiple terms (such as *Dubya*, *Bush*, or *George W. Bush*) refer to the same database record.

The meaning of categories will be a lambda expression. This can be thought of as just a symbolic stand-in for the domain element it denotes in some model. However, in computational work, in order to use proof-theoretic techniques for reasoning in computational applications, we need to be able to manipulate these terms. Indeed, in computational work the main focus is on manipulation of the semantic values, ignoring the model-theoretic interpretation.

Exercise: How can logical constants like **and** and *quantifiers* like $\mathbf{every}_{\mathbf{Ind}}$ be represented in this model theory? The intended interpretation is that **and** will be true only if its two arguments are true, and $\mathbf{every}_{\mathbf{Ind}}$ requires some property to hold of every individual in the model: $\mathbf{every}_{\mathbf{Ind}}$ is of type $(\mathbf{Ind} \to \mathbf{Bool}) \to \mathbf{Bool}$.

5 The problem of quantifiers motivates use of higher order logic

"Seems pretty easy," you might say, "is that all there is to it?" Not quite. As soon as you get away from the domain of proper names and into noun phrases containing determiners or quantifiers, things start to change. When doing a predicate logic course, have you ever wondered how come *Kathy runs* is $\mathbf{run}(k)$, but *no kid runs* is $\neg(\exists x)(\mathbf{kid}(x) \wedge \mathbf{run}(x))$? Bertrand Russell did (1905). Or you might have noticed that the following argument doesn't hold:

(2) Nothing is better than a life of peace and prosperity.
A cold egg salad sandwich is better than nothing.
____
A cold egg salad sandwich is better than a life of peace and prosperity.

But it would, given the semantics in the previous section, and assuming transitivity of **better** and that the semantic rules for VP and S presented in the previous section applied straightforwardly to semantic values for *nothing*, *a life of peace and prosperity*, and *a cold egg salad sandwich* as NPs (why?). The problem is that *nothing* is a *quantifier*.

There are several ways to deal with quantifiers; the one we'll look at crucially uses our logical formalism with a rich system of higher order types, among which *type-shifting* can occur. We will now allow variables and quantification over relations (over relations).

Crucially, now, we can perform λ-abstraction over arbitrary types, not just constants as we could before. This is necessary to capture the fundamental insight that the semantics of a quantifier or determiner contains the semantics of the verb, but the semantics of the verb contains the semantics of the noun next to the determiner. In order to get this result compositionally, a λ-expression must be applied to a non-constant expression. We'll see this soon.

6 A first grammar fragment

But for now, we'll take a look at the nice fact that this higher-order type system allows us to do modification of VPs and NPs very easily. We expand somewhat the grammar from the previous section, write it down in one place, and take a look at the results. We give the type for each constant or function in the lexicon as a subscript for reference. We won't usually include it in derivations.

Kathy, NP : $\mathbf{kathy}_{\mathbf{Ind}}$
Fong, NP : $\mathbf{fong}_{\mathbf{Ind}}$
Palo Alto, NP : $\mathbf{paloalto}_{\mathbf{Ind}}$
car, N : $\mathbf{car}_{\mathbf{Ind} \to \mathbf{Bool}}$
overpriced, Adj : $\mathbf{overpriced}_{(\mathbf{Ind} \to \mathbf{Bool}) \to (\mathbf{Ind} \to \mathbf{Bool})}$
outside, PP : $\mathbf{outside}_{(\mathbf{Ind} \to \mathbf{Bool}) \to (\mathbf{Ind} \to \mathbf{Bool})}$
red, Adj : $\lambda P.(\lambda x.P(x) \wedge \mathbf{red'}(x))$
in, P : $\lambda y.\lambda P.\lambda x.(P(x) \wedge \mathbf{in'}(y)(x))$
the, Det : ι
a, Det : $\mathbf{some^2}_{(\mathbf{Ind} \to \mathbf{Bool}) \to (\mathbf{Ind} \to \mathbf{Bool}) \to \mathbf{Bool}}$
runs, V : $\mathbf{run}_{\mathbf{Ind} \to \mathbf{Bool}}$
respects, V : $\mathbf{respect}_{\mathbf{Ind} \to \mathbf{Ind} \to \mathbf{Bool}}$
likes, V : $\mathbf{like}_{\mathbf{Ind} \to \mathbf{Ind} \to \mathbf{Bool}}$
sees, V : $\mathbf{see}_{\mathbf{Ind} \to \mathbf{Ind} \to \mathbf{Bool}}$

S : $\beta(\alpha) \to$ NP : α VP : β
NP : $\beta(\alpha) \to$ Det : β N′ : α
N′ : $\beta(\alpha) \to$ Adj : β N′ : α
N′ : $\beta(\alpha) \to$ N′ : α PP : β
N′ : $\beta \to$ N : β
VP : $\beta(\alpha) \to$ V : β NP : α
VP : $\beta(\gamma)(\alpha) \to$ V : β NP : α NP : γ
VP : $\beta(\alpha) \to$ VP : α PP : β
VP : $\beta \to$ V : β
PP : $\beta(\alpha) \to$ P : β NP : α

$\mathbf{in'}$ is $\mathbf{Ind} \rightarrow \mathbf{Ind} \rightarrow \mathbf{Bool}$
$\mathbf{in} \stackrel{\text{def}}{=} \lambda y.\lambda P.\lambda x.(P(x) \wedge \mathbf{in'}(y)(x))$ is $\mathbf{Ind} \rightarrow (\mathbf{Ind} \rightarrow \mathbf{Bool}) \rightarrow (\mathbf{Ind} \rightarrow \mathbf{Bool})$
$\mathbf{red'}$ is $\mathbf{Ind} \rightarrow \mathbf{Bool}$
$\mathbf{red} \stackrel{\text{def}}{=} \lambda P.(\lambda x.(P(x) \wedge \mathbf{red'}(x))$ is $(\mathbf{Ind} \rightarrow \mathbf{Bool}) \rightarrow (\mathbf{Ind} \rightarrow \mathbf{Bool})$

Using this grammar, you might note that for intersective adjectives (like *red*) and prepositions we get spurious attachment ambiguities – that is that there are two syntactic derivations that yield the same resulting semantics.

- $\mathrm{N}' : \lambda x.\mathbf{car}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x) \wedge \mathbf{red'}(x)$
  - $\mathrm{Adj} : \lambda P.(\lambda x.P(x) \wedge \mathbf{red'}(x))$
    - *red*
  - $\mathrm{N}' : \lambda x.(\mathbf{car}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x))$
    - $\mathrm{N}' : \mathbf{car}$
      - $\mathrm{N} : \mathbf{car}$
        - *car*
    - $\mathrm{PP} : \lambda P.\lambda x.(P(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x))$
      - $\mathrm{P} : \lambda y.\lambda P.\lambda x.(P(x) \wedge \mathbf{in'}(y)(x))$
        - *in*
      - $\mathrm{NP} : \mathbf{paloalto}$
        - *Palo Alto*

- $\mathrm{N}' : \lambda x.\mathbf{car}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x) \wedge \mathbf{red'}(x))$
  - $\mathrm{N}' : \lambda x.(\mathbf{car}(x) \wedge \mathbf{red'}(x))$
    - $\mathrm{Adj} : \lambda P.(\lambda x.P(x) \wedge \mathbf{red'}(x))$
      - *red*
    - $\mathrm{N}' : \mathbf{car}$
      - $\mathrm{N} : \mathbf{car}$
        - *car*
  - $\mathrm{PP} : \lambda P.\lambda x.(P(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x))$
    - $\mathrm{P} : \lambda y.\lambda P.\lambda x.(P(x) \wedge \mathbf{in'}(y)(x))$
      - *in*
    - $\mathrm{NP} : \mathbf{paloalto}$
      - *Palo Alto*

But for non-intersective adjectives (such as *overpriced*, we don't; we get two different meanings. We'll get:

overpriced(in(paloalto)(house))
in(paloalto)(overpriced(house))

Non-intersective adjectives are precisely those whose denotation depends on the term they are modifying in a way that cannot be described as set intersection. (In the example above, we leave out the determiner from the noun phrase; we'll return to that topic soon.)

Here's an example of a preposition modifying a VP:

- $\mathrm{S} : \mathbf{run}(\mathbf{kathy}) \wedge \mathbf{in'}(\mathbf{paloalto})(\mathbf{kathy}))$
  - $\mathrm{NP} : \mathbf{kathy}$
    - *Kathy*
  - $\mathrm{VP} : \lambda x.(\mathbf{run}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x))$
    - $\mathrm{VP} : \mathbf{run}$
      - $\mathrm{V} : \mathbf{run}$
        - *runs*
    - $\mathrm{PP} : \lambda P.\lambda x.(P(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x))$
      - $\mathrm{P} : \lambda y.\lambda P.\lambda x.(P(x) \wedge \mathbf{in'}(y)(x))$
        - *in*
      - $\mathrm{NP} : \mathbf{paloalto}$
        - *Palo Alto*

Note that because a VP and an N′ are both a property, a PP can modify either. On the other hand, note that the intersective semantics for *in* that I have used isn't necessarily correct here – it gives a "subject-centered" analysis rather than an "event-centered" analysis.

For such semantic forms, it is straightforward to evaluate them in a model or against a database or a knowledge base (which instantiates a model).

7 Database/knowledgebase interfaces

We will assume that the predicates of our logical representations are available as tables in a database. We'll assume a straightforward mapping of one table per predicate. In practice, there would often be more indirect mappings (for instance, a unary **red** predicate could come from a table that lists objects and their colors), but we'll assume we have any necessary *views* set up to make the mapping straightforward. Let's do the easiest example again:

S : **respect**(**fong**)(**kathy**)

NP : **kathy** — VP : **respect**(**fong**)

Kathy — V : **respect** — NP : **fong**

respects — *Fong*

Here we assume that **respect** is a table Respect with two fields respecter and respected, and that **kathy** and **fong** are IDs in the database: k and f. If this were asserting a statement, we might evaluate the form **respect**(**fong**)(**kathy**) by doing an insert operation:

insert into Respects(respecter, respected) values (k, f)

Or if we wish to check that we agree with a statement, we might instead do a select from the database, and say whether we agree that the statement is true or not. In particular, below, we will focus on questions like the corresponding *Does Kathy respect Fong* for which we will use the relation to ask:

select 'yes' from Respects where Respects.respecter = k and Respects.respected = f

(we interpret "no rows returned" as **0**).

8 Generalized Quantifiers

We have a reasonable semantics for *red car in Palo Alto* as a property from **Ind** → **Bool**, but what about when we want to represent noun phrases like *the red car in Palo Alto* or *every red car in Palo Alto*? Let's start with *the* to which we gave without explanation the translation ι above. $[\![\iota]\!](P) = a$ if ($P(b) = \mathbf{1}$ iff $b = a$). We use this as a semantics for *the* following Russell, for whom *the x* meant the unique item satisfying a certain description (in reality things get a little more complex, I might note). ι is thus only a partial function, returning undefined if there is no such object (undefined represents a presupposition failure which is technically different from a false statement).

Let's continue the example from above, doing now *the red car in Palo Alto*:

- NP : $\iota(\lambda x.\mathbf{car}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x) \wedge \mathbf{red'}(x))$
  - Det : ι
    - *the*
  - N′ : $\lambda x.\mathbf{car}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x) \wedge \mathbf{red'}(x)$
    - *red car in Palo Alto*

For *red car in Palo Alto* we have the property semantics from before, which we render in SQL as:

```
select Cars.obj from Cars, Locations, Red where Cars.obj = Locations.obj AND
Locations.place = 'paloalto' AND Cars.obj = Red.obj
```

(here we assume the unary relations have one field, obj). The effect of ι would then be rendered via a having clause:

```
select Cars.obj from Cars, Locations, Red where Cars.obj = Locations.obj AND
Locations.place = 'paloalto' AND Cars.obj = Red.obj
having count(*) = 1
```

Here's another example:

- S : $\mathbf{run}(\iota(\mathbf{overpriced}(\mathbf{car})))$
  - NP : $\iota(\mathbf{overpriced}(\mathbf{car}))$
    - Det : ι
      - *the*
    - N′ : $\mathbf{overpriced}(\mathbf{car})$
      - Adj : $\mathbf{overpriced}$
        - *overpriced*
      - N : $\mathbf{car}$
        - *car*
  - VP : $\mathbf{run}$
    - V : $\mathbf{run}$
      - *runs*

What then of *every red car in Palo Alto*? If we want to translate a sentence like *Every red car in Palo Alto is expensive*, then we need to make a quantificational statement that involves *red cars in Palo Alto* and *expensive things*. A generalized determiner is a relation between two properties, one contributed by the restriction from the N′, and one contributed by the predicate quantified over:

$$(\mathbf{Ind} \to \mathbf{Bool}) \to (\mathbf{Ind} \to \mathbf{Bool}) \to \mathbf{Bool}$$

(In other literature, these are generally called "generalized quantifiers", but we take quantifiers to take a single property argument, and to map from $(\mathbf{Ind} \to \mathbf{Bool}) \to \mathbf{Bool}$, that is to be things of the type at the bottom of the type-shifting triangle below. A generalized determiner maps from a property to a quantifier.) Here are some other determiners:

$$\mathbf{some}^2(\mathbf{kid})(\mathbf{run}) \equiv \mathbf{some}(\lambda x.\mathbf{kid}(x) \wedge \mathbf{run}(x))$$
$$\mathbf{every}^2(\mathbf{kid})(\mathbf{run}) \equiv \mathbf{every}(\lambda x.\mathbf{kid}(x) \to \mathbf{run}(x))$$

Generalized determiners are implemented via the quantifiers:

$$\mathbf{every}(P) = \mathbf{1} \text{ iff } (\forall x)P(x) = \mathbf{1}\text{; i.e., if } P = \mathbf{Dom_{Ind}}$$
$$\mathbf{some}(P) = \mathbf{1} \text{ iff } (\exists x)P(x) = \mathbf{1}\text{; i.e., if } P \neq \emptyset$$

Exercise:

$\mathbf{no}^2(\mathbf{kid})(\mathbf{run}) \equiv$
$\mathbf{two}^2(\mathbf{kid})(\mathbf{run}) \equiv$

A central insight of Montague's PTQ (proper theory of quantification) was that the same type could be applied to indviduals, representing them as quantifiers (which are type-raised individuals):

$Kathy : \lambda P.P(\mathbf{kathy})$

The syntactic category of noun phrases can then be realized uniformly in the semantic dimension, by making all noun phrases be quantifiers. This was both good, and bad – everything was always raised to the most complicated type needed for anything of the category. We'd prefer to allow more flexible *type-shifting* for nominal expressions in circumstances where it is needed. The diagram below shows common patterns of nominal type shifting:

$$\begin{tikzcd}
\mathbf{Ind} \arrow[rr, shift left, "\mathbf{Q}"] \arrow[dr, "\mathbf{R}"'] & & \mathbf{Ind} \to \mathbf{Bool} \arrow[ll, dashed, shift left, "\iota"] \arrow[dl, "\mathbf{some}^2"] \\
& (\mathbf{Ind} \to \mathbf{Bool}) \to \mathbf{Bool} &
\end{tikzcd}$$

$\mathbf{R}(\mathrm{x}) = \lambda P.P(x)$
$\mathbf{some}^2(\mathrm{P}) = \lambda Q.(Q \cap P) \neq \emptyset$
$\mathbf{Q}(\mathrm{x}) = \lambda y.x = y$

In this diagram, **R** is exactly this basic type-raising function for individuals.

9 Noun phrase scope

Next, I'll illustrate how this approach can handle quantifier scope ambiguities. We'll first do *Every student runs*. We could just redefine things to allow the subject to take the verb phrase as its argument semantically, but we can maintain the usual idea of what is the functor (the verb) and what is the argument (the noun phrase) by alternatively doing **argument raising** on the VP. This applies the ideas of nominal type-raising and lowering which we saw earlier within functional types (that is, it changes the types of the NP arguments). We will use rules as follows (Hendriks 1993, simplified):

Value raising raises a function that produces an individual as a result to one that produces a quantifer. If $\alpha : \sigma \to \mathbf{Ind}$ then $\lambda x.\lambda P.P(\alpha(x)) : \sigma \to (\mathbf{Ind} \to \mathbf{Bool}) \to \mathbf{Bool}$

Argument raising replaces an argument of a boolean function with a variable and applies the quantifier semantically binding the replacing variable. If $\alpha : \sigma \to \mathbf{Ind} \to \tau \to \mathbf{Bool}$ then $\lambda x_1.\lambda Q.\lambda x_3.Q(\lambda x_2.\alpha(x_1)(x_2)(x_3)) : \sigma \to (\mathbf{Ind} \to \mathbf{Bool}) \to \mathbf{Bool} \to \tau \to \mathbf{Bool}$

Argument lowering replaces a quantifier in a boolean function with an individual argument, where the semantics is calculated by applying the original function to the type raised argument. If $\alpha : \sigma \to ((\mathbf{Ind} \to \mathbf{Bool}) \to \mathbf{Bool}) \to \tau \to \mathbf{Bool}$ then $\lambda x_1.\lambda x_2.\lambda x_3.\alpha(x_1)(\lambda P.P(x_2))(x_3) : \sigma \to \mathbf{Ind} \to \tau \to \mathbf{Bool}$

Here is an example showing argument raising of the verb:

- S : $\mathbf{every}^2(\mathbf{student})(\mathbf{run}) \equiv \mathbf{every}(\lambda x.\mathbf{student}(x) \to \mathbf{run}(x))$
  - NP : $\mathbf{every}^2(\mathbf{student})$
    - Det : $\mathbf{every}^2$
      - *every*
    - N′ : $\mathbf{student}$
      - N : $\mathbf{student}$
        - *student*
  - VP : $\lambda Q.Q(\lambda x.\mathbf{run}(x))$
    - VP : $\mathbf{run}$
      - V : $\mathbf{run}$
        - *runs*

Following Hendriks (1993), we can use argument type shifting – argument raising and lowering – to do (a fair amount of) the analysis of scopal ambiguities, for example, the two readings of *Some kid broke every toy* – one where one particular kid broke them all, and the other where different kids broke different toys:

- S : $\mathbf{every}^2(\mathbf{toy})(\lambda y_o.\mathbf{some}^2(\mathbf{kid})(\lambda x_s.\mathbf{break}(y_o)(x_s)))$
  - NP : $\mathbf{some}^2(\mathbf{kid})$
    - Det : $\mathbf{some}^2$
      - *some*
    - N′ : $\mathbf{kid}$
      - N : $\mathbf{kid}$
        - *kid*
  - VP : $\lambda S'.\mathbf{every}^2(\mathbf{toy})(\lambda y_o.S'(\lambda x_s.\mathbf{break}(y_o)(x_s)))$
    - V : $\lambda O.\lambda S'.O(\lambda y_o.S'(\lambda x_s.\mathbf{break}(y_o)(x_s)))$
      - V : $\lambda x_o.\lambda S.S(\lambda x_s.\mathbf{break}(x_o)(x_s))$
        - V : $\lambda y.\lambda x.\mathbf{break}(y)(x)$
          - *broke*
    - NP : $\mathbf{every}^2(\mathbf{toy})$
      - Det : $\mathbf{every}^2$
        - *every*
      - N′ : $\mathbf{toy}$
        - N : $\mathbf{toy}$
          - *toy*

- S : $\mathbf{some}^2(\mathbf{kid})(\lambda y_s.\mathbf{every}^2(\mathbf{toy})(\lambda x_o.\mathbf{break}(x_o)(y_s)))$
  - NP : $\mathbf{some}^2(\mathbf{kid})$
    - Det : $\mathbf{some}^2$
      - *some*
    - N′ : $\mathbf{kid}$
      - N : $\mathbf{kid}$
        - *kid*
  - VP : $\lambda S.S(\lambda y_s.\mathbf{every}^2(\mathbf{toy})(\lambda x_o.\mathbf{break}(x_o)(y_s)))$
    - V : $\lambda O'.\lambda S.S(\lambda y_s.O(\lambda x_o.\mathbf{break}(x_o)(y_s)))$
      - V : $\lambda x_s.O(\lambda x_o.\mathbf{break}(x_o)(x_s))$
        - V : $\lambda y.\lambda x.\mathbf{break}(y)(x)$
          - *broke*
    - NP : $\mathbf{every}^2(\mathbf{toy})$
      - Det : $\mathbf{every}^2$
        - *every*
      - N′ : $\mathbf{toy}$
        - N : $\mathbf{toy}$
          - *toy*

10 Questions with answers!

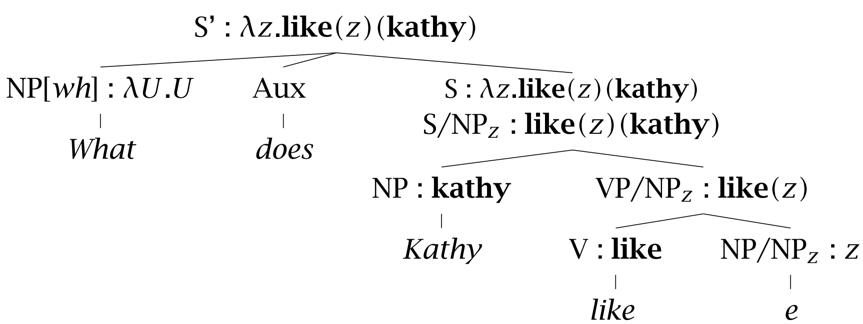
We would like to be able to work with more complex examples. I'll show some (hopefully correct!) SQL, but we will only work through the translation from English to logical representations. The translation to SQL is then "in principle" straightforward, since relational databases implement a simple form of logic (cf. Datalog).

Questions will not actually have a special logical representation. A yes/no question (*Is Kathy running?*) will be something of type **Bool**, but if it is a question we will interrogate our database to see if it is true or not, rather than either asserting or checking the fact as for a statement. We could put something like a question mark before it to indicate this interpretation, but I won't. A content question (*Who likes Kathy?*) will be an *open proposition*, that is something semantically of the type *property* (**Ind** → **Bool**), and operationally we will consult the database to see what individuals will make the statement true. For the kind of semantic forms we produce, it is fairly straightforward to evaluate them in a model or against a database or a knowledge base.

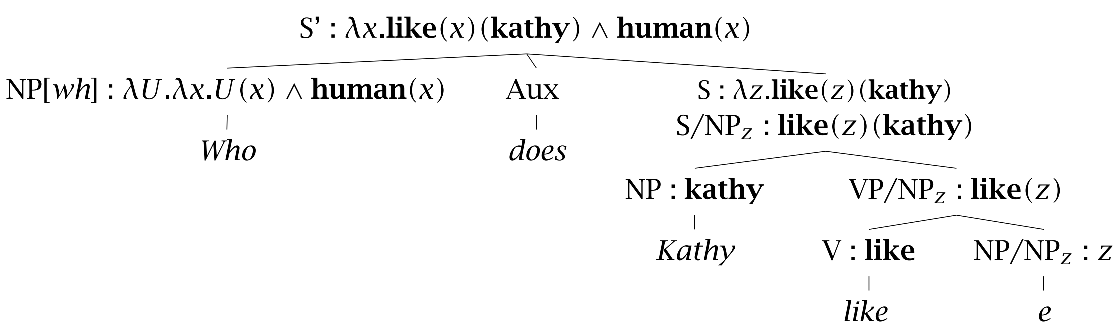
We will need new syntax rules for questions, new lexical items, and empty missing NPs, which include a means of *gap threading* to link them up with their filler, the *wh*-phrase. I won't give formal rules, but empty gapped elements are carried up the tree, until the variable is reintroduced at some (sentential) node. (This idea of gap threading is formalized in various syntactic theories like GPSG/HPSG.) Under our analysis, auxiliaries make no contribution to meaning – that's reasonable enough as what meaning contribution they do make is in areas like tense, which we aren't dealing with.

$\mathrm{S}' : \beta(\alpha) \to \mathrm{NP}[wh] : \beta \quad \mathrm{Aux} \quad \mathrm{S} : \alpha$	*who*, $\mathrm{NP}[wh] : \lambda U.\lambda x.U(x) \wedge \mathbf{human}(x)$
$\mathrm{S}' : \alpha \to \mathrm{Aux} \quad \mathrm{S} : \alpha$	*what*, $\mathrm{NP}[wh] : \lambda U.U$
$\mathrm{NP}/\mathrm{NP}_z : z \to \mathrm{e}$	*which*, $\mathrm{Det}[wh] : \lambda P.\lambda V.\lambda x.P(x) \wedge V(x)$
$\mathrm{S} : \lambda z.F(\ldots z \ldots) \to \mathrm{S}/\mathrm{NP}_z : F(\ldots z \ldots)$	*how_many*, $\mathrm{Det}[wh] : \lambda P.\lambda V.\lvert\lambda x.P(x) \wedge V(x)\rvert$

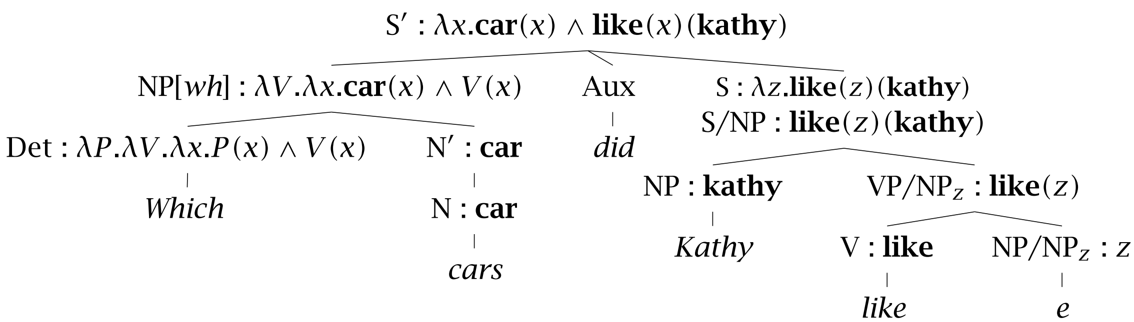
Where $|\cdot|$ is the operation that returns the cardinality of a set (count).



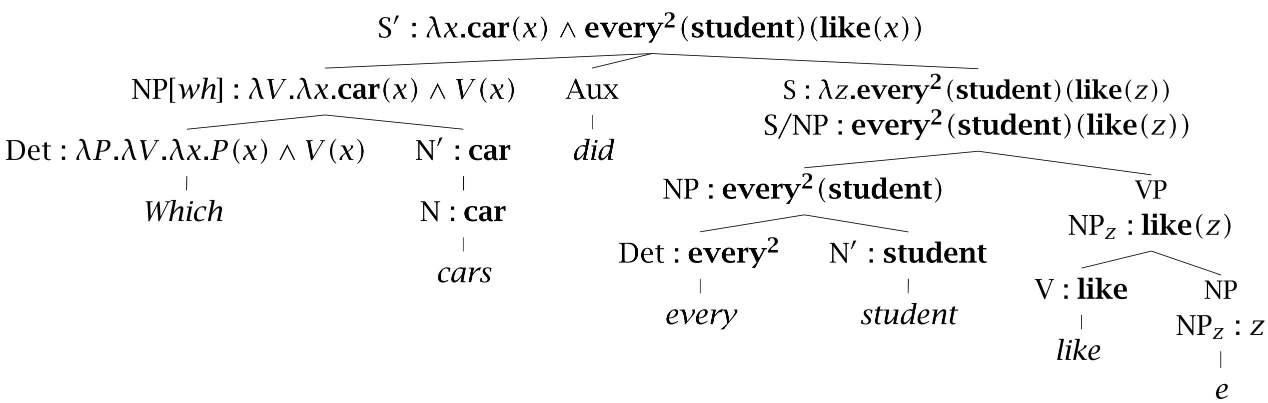
select liked from Likes where Likes.liker='Kathy'

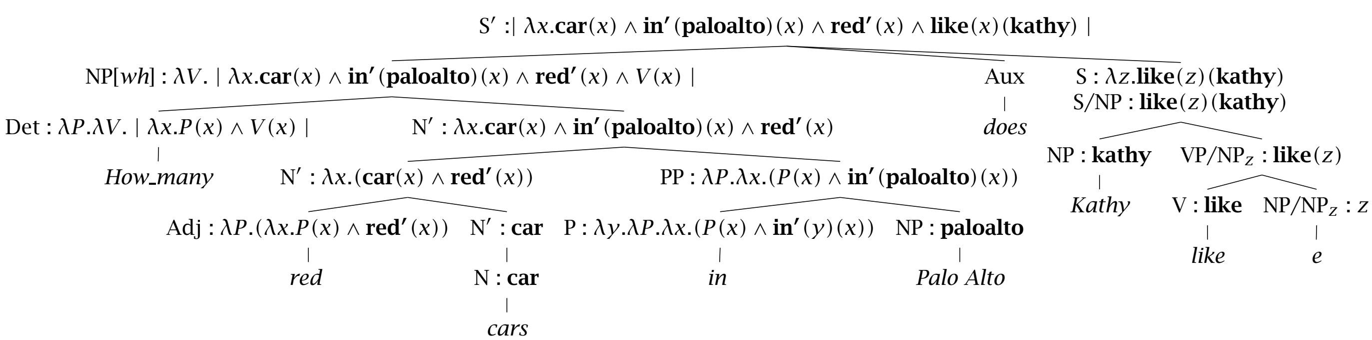


select liked from Likes,Humans where Likes.liker='Kathy' AND Humans.obj=Likes.liked

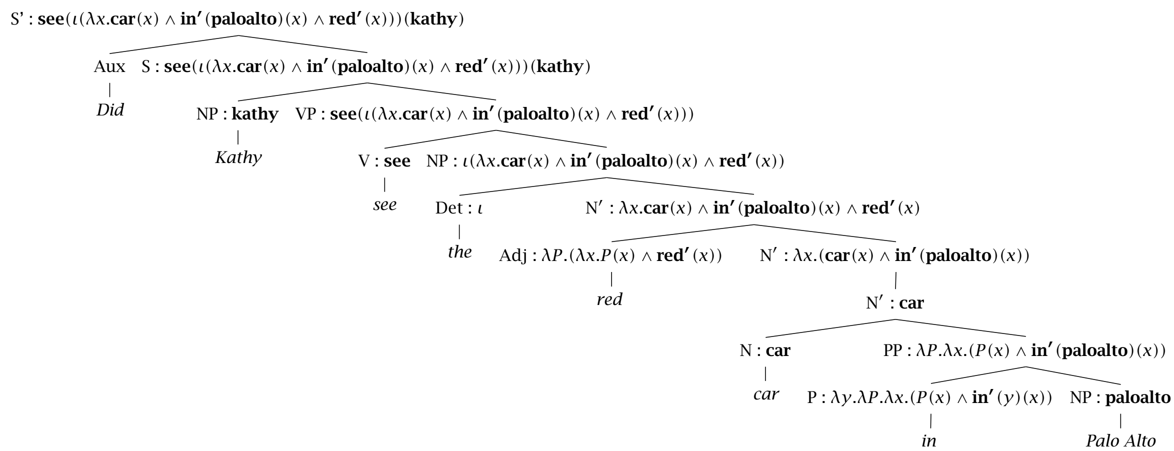



```
select liked from Cars,Likes where Cars.obj=Likes.liked AND Likes.liker='Kathy'
```






```
select count(*) from Likes,Cars,Locations,Reds where Cars.obj = Likes.liked AND
Likes.liker = 'Kathy' AND Red.obj = Likes.liked AND Locations.place = 'Palo Alto'
AND Locations.obj = Likes.liked
```



select 'yes' where Seeings.seer = k AND
Seeings.seen = (select Cars.obj from Cars, Locations, Red where Cars.obj =
Locations.obj AND Locations.place = 'paloalto' AND Cars.obj = Red.obj
having count(*) = 1)

References

There's at last an up-to-date book length treatment of computational semantics: Patrick Blackburn & Johan Bos, *Representation and Inference for Natural Language: A First Course in Computational Semantics*, CSLI Publications, 2005. They also provide implementations of their models in Prolog. Many other places you might look are not as useful as you might hope. Jurafsky & Martin present some of the basic implements for doing computational semantics, but it is hard to take seriously a fragment that can't even handle a sentence like *Who did you call?* [and they just do translation to markerese]. Russell and Norvig also avoid dealing properly with quantifiers. The presentation here draws most closely from Bob Carpenter, *Type-Logical Semantics* (MIT Press 1999), but reformulates things so that I don't have to introduce categorial grammar. (Many semanticists like categorial grammar because of the nice connections it provides between the syntactic and semantic theories. In particular, things we have specified, like type-raising rules, are theorems of the Lambek calculus.) There are other good intros to semantics from a linguistic perspective, including De Swart, *Introduction to Natural Language Semantics* (CSLI), Gamut, *Logic, Language and Meaning* (University of Chicago Press, 1991), and Heim and Kratzer, *Semantics in Generative Grammar* (Blackwell, 1997), but they don't mention computational issues.