Chapter 7

Semantics

When we began our study of syntax, I had to spend some time convincing you that syntax exists (that is, that there is some kind of representation of syntactic structure in your mind when you use language), but the basic shape of those representations has not been all that controversial.

By contrast, as we turn our attention to *semantics*, or meaning, I probably don't need to convince you at all that meaning exists, but in theories of semantics and computational approaches to semantics, representations of meaning come in all kinds of shapes and sizes.

Semantics is often divided into the meanings of words, known as *lexical semantics*, and the meanings of sentences, which is usually just called *semantics*. As I see it, there are three main approaches to representing meaning in NLP:

1. Vectors: whose components are features, either designed by humans or automatically learned.
2. Graphs: in which nodes are usually entities and edges are various kinds of relationships among them.
3. Logic: formulas of logics of various kinds, or SQL queries, or even computer programs.

7.1 Vectors

We've already seen lots of examples of vector representations of text. For example, we saw, in IBM Model 1, possibly the simplest way to represent the "meaning" of a text, which is to assume that each word in the text contributes a little bit of meaning, which we can represent as a one-hot vector, and that we can combine the "meanings" of words simply by adding up their vectors.

$$\text{the cat sat on the mat} \rightarrow \{\text{cat} : 1, \text{mat} : 1, \text{on} : 1, \text{sat} : 1, \text{the} : 2\}$$

In information retrieval, this is called a *term vector*, but in NLP it's more commonly known (somewhat pejoratively) as a *bag of words*. This representation completely ignores any semantic relationships between words (like synonymy) and any kind of structure of the text (like word order or syntax). Yet, in many situations, it can be very effective.

After IBM Model 1, recall that we factored the model so that it computed a d-dimensional dense representation of the sentence. This sentence representation was computed as a by-product of a model trained to carry out a particular task, like translation.

7.1.1 Text classification

One type of problem where vector representations of texts come up very naturally is *text classification*. The simplest kind of text classification model is called a *naive Bayes* classifier; here, we'll focus on classification models based on *logistic regression*, which in the world of neural networks looks like this:

$$\mathbf{h} \in \mathbb{R}^d$$

$$\mathbf{h} = \text{Encoder}(w) \tag{7.1}$$

$$\mathbf{y} = \text{SoftmaxLayer}^{\boxed{1}}(\mathbf{h}) \tag{7.2}$$

$$P(k \mid w) = \mathbf{y}_k \tag{7.3}$$

where $\text{Encoder}(\cdot)$ is some function that encodes strings as vectors.

The simplest choice of encoding would be a bag of words. That is, $d = |\Sigma|$, the words of Σ are numbered $1, \ldots, d$, and $\mathbf{h}_\sigma$ is the number of times that σ occurs in w. The parameters of the SoftmaxLayer would be weights $\mathbf{W}_{k,\sigma}$ that capture how much word σ is predictive of class k; for example, we would expect $\mathbf{W}_{\texttt{find\_hotel,stay}}$ to be high, but $\mathbf{W}_{\texttt{find\_hotel,eat}}$ to be low. This kind of classifier is usually known as *logistic regression*.

There are many imaginable neural networks that can be used as string encoders, but the one that is most often used now works as follows. Prepend a special token `CLS` to the string, so the string is now $\texttt{CLS}w_1 w_2 \cdots w_n$. Then apply a Transformer encoder, that is, look up word and position embeddings for each word, then apply a stack of alternating self-attention layers and position-wise feedforward neural networks. The result is a sequence of $(n+1)$ vectors. Take the first of these (the one corresponding to `CLS`) to be $\mathbf{h}$.

7.2 Graphs

The next broad category of semantic representation I want to talk about is semantic *graphs*. I'm including under this heading a bunch of tasks like

- Named entity recognition: Which noun phrases are names of people, places, etc., and what entity do they refer to?
- Coreference resolution: Which noun phrases (including pronouns) refer to the same thing?
- Word sense disambiguation: If a word has more than one sense, which one is being used in this context?
- Semantic role labeling: For each action in the sentence, who/what is the agent (the one doing the action), the patient (the one upon whom the action is done), etc.?

- Relation extraction: Sometimes we're interested in higher-level relationships between entities, like "x is the president of y."

Ultimately, the entities and relationships that these tasks find can be assembled into a graph – either a graph that represents the meaning of a sentence, or a big graph that represents the knowledge contained in a whole collection of text.

7.2.1 Named entity recognition (sequence labeling)

Problem

In named entity recognition (NER), the input is a sentence like

> Please find me a train from cambridge to stansted airport

and the output has all the named entities tagged like this:

> Please find me a train from [cambridge]$_{\text{LOC}}$ to [stansted airport]$_{\text{LOC}}$

In the most common NER dataset, the four types are person (PER), organization (ORG), location (LOC) and miscellaneous (MISC).

The most common way to formulate this kind of problem, where the computer has to identify a number of non-overlapping substrings of the input, is called *BIO tagging*.

O	O	O	O	O	O	B-LOC	O	B-LOC	I-LOC
Please	find	me	a	train	from	cambridge	to	stansted	airport

B stands for "begin" and is used for the first word in each slot-filler; I stands for "inside" and is used for the second and subsequent words in each slot-filler. O stands for "outside" and is used for any word that does not belong to a slot-filler. Other schemes exist, like BILOU (L for the last word an entity, U 'unit' for the only word in an entity), but this is the simplest and most common.

Now we've reduced slot-filling to a *sequence labeling* task. Other examples of sequence labeling tasks are:

- Word segmentation: Given a representation of a sentence without any word boundaries, reconstruct the word boundaries. In some languages, like Chinese, words are written without any spaces in between them. (Indeed, it can be difficult to settle on the definition of a "word" in such languages.)
- Part of speech tagging: Given a sentence, label each word with its part of speech.
- Slot filling: Given a (partial) query, identify the spans that correspond to parameters of the query. For example, in a travel booking system, the slots might be the departure and arrival airports, the number of seats, etc.

One of the hallmarks of sequence labeling problems is dependencies between the labels. For example, if we're doing named entity recognition, a model might learn that *Dame* has a high probability of being tagged I-ORG, as the last word of *Notre Dame* (and *University of Notre Dame, Cathedral of Notre Dame,* etc.). But in a sentence like

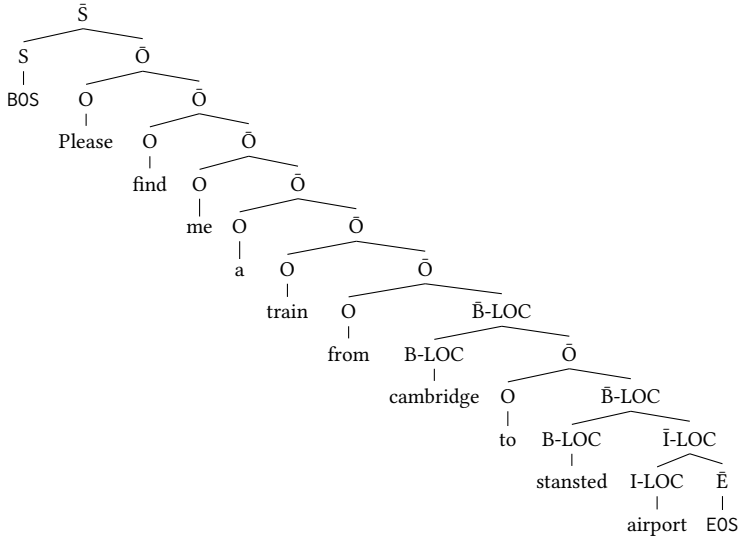
In 2004, Dame Julie Andrews voiced Queen Lillian in *Shrek 2*,

Dame should be tagged B-PER. Maybe the model can get enough clues from the surrounding words to tag it correctly, but the strongest clue should be that I-ORG absolutely cannot follow O.

Sequence labeling as parsing

The classic models used to solve sequence labeling problems are, in historical order, hidden Markov Models (HMMs), conditional random fields (CRFs), and biLSTM-CRFs. HMMs and CRFs are usually formulated as either finite automata or matrix operations. But since you have parsing with CFGs fresh in your mind, let's formulate them as CFGs. It may be overkill, but it's arguably the cleanest way to write them.

What would a parse tree look like for this task? The labels (B-*, I-*, O) would be like parts of speech, and since we don't have any other kind of structure, it only makes sense to use a purely right-branching or left-branching structure. Let's choose right-branching:



So the grammar has the following kinds of rules:

$\bar{X} \rightarrow X\ \bar{Y}$	X and Y are labels
$X \rightarrow a$	X is a label and a is a word (incl. BOS)
$\bar{\text{E}} \rightarrow \text{EOS}$	

The start symbol is $\bar{\text{S}}$ (not S). Let's call the first kind of rules *transition rules* and the last two kinds *emission rules*. (The reason we have to special-case the last rule is because it's the only emission rule whose left-hand side has a bar over it. Otherwise, it's not really that special.)

Classic models

If the grammar is a probabilistic CFG, then this is equivalent to a hidden Markov model. The probabilities of the transition rules $\bar{X} \to X\ \bar{Y}$ measure the probability of one label coming after another and are called *transition probabilities*. The emission rules, $X \to a$, measure the probability of generating a word given a label and are called *emission probabilities*.

If the grammar is a weighted (not necessarily probabilistic) CFG, then this is equivalent to a conditional random field. A rule can have any weight $p > 0$; we also call $\log p$ its *score*. Note that in order for this CFG to be equivalent to a CRF, we have to include all rules of the form $X \to a$ and $\bar{X} \to X\ \bar{Y}$, even if they were not observed in the training data.

RNN+CRFs

In the parsing chapter and in HW3, we built a neural parser by using a neural network to compute the rule scores of a weighted CFG. We can do the exact same thing here, but with a slightly different neural network.

We start off, as usual, with a sequence encoder. Let $w = w_1 \cdots w_n$ be the input string, with $w_1 = \texttt{BOS}$ and $w_n = \texttt{EOS}$. Let Γ be the set of possible labels.

$$\begin{aligned} &\mathbf{V} \in \mathbb{R}^{n \times d} \\ &\mathbf{V}_i = \text{Embedding}^{\boxed{1}}(w_i) \qquad\qquad i = 1, \ldots, n \qquad (7.4) \\ &\mathbf{H} \in \mathbb{R}^{n \times d} \\ &\mathbf{H} = \text{RNN}^{\boxed{2}}(\mathbf{V}). \qquad (7.5) \end{aligned}$$

Usually the encoder is a fancier kind of RNN called a bidirectional LSTM, but we're sticking to a simple, left-to-right RNN here. Each $\mathbf{H}_i$ is the encoding of w_i. So far, this is the same as the neural parser from before.

Next, we need to define a function that assigns a score to every rule, possibly depending on its position in the string. We define this function for the three kinds of rules as follows:

$$s(\bar{X} \to {}_i\, X\ \bar{Y}\, {}_n) = \mathbf{T}_{X,Y} \qquad 0 \le i \le n-2 \qquad (7.6)$$

$$s(X \to {}_{i-1}\, a\, {}_i) = \mathbf{O}_{i,X} \qquad 1 \le i < n \qquad (7.7)$$

$$s(\bar{\text{E}} \to {}_{n-1}\, \texttt{EOS}\, {}_n) = \mathbf{O}_{i,\text{E}} \qquad (7.8)$$

where $\mathbf{T} \in \mathbb{R}^{|\Gamma| \times |\Gamma|}$ is a matrix of learnable parameters, so that every transition rule gets an independent score; and $\mathbf{O}$ is computed from the RNN encodings as

$$\begin{aligned} &\mathbf{O} \in \mathbb{R}^{n \times |\Gamma|} \\ &\mathbf{O}_i = \text{LinearLayer}^{\boxed{3}}(\mathbf{H}_i). \qquad (7.9) \end{aligned}$$

Now both training and labeling (= parsing) can be done exactly as before. But, this is an extremely inefficient way of implementing a RNN+CRF. Since the grammar includes all rules with the forms shown above, even if they were not observed in the training data, the grammar is quite large. In the next section, we'll see how to optimize this.

RNN+CRFs made more efficient

Recall that during training, we maximize

$$
\begin{aligned}
L &= \sum_{(w,\text{tree}) \in \text{data}} \log P(\text{tree} \mid w) \\
&= \sum_{(w,\text{tree}) \in \text{data}} \log \frac{\exp s(\text{tree})}{\sum_{\text{tree}'} \exp s(\text{tree}')} \\
&= \sum_{(w,\text{tree}) \in \text{data}} \left(s(\text{tree}) - \underbrace{\log \sum_{\text{tree}'} \exp s(\text{tree}')}_{\text{partition function}} \right)
\end{aligned}
$$

and the partition function is computed using a modified CKY algorithm. And during parsing, we use the CKY algorithm.

As a reminder, here's the algorithm, where we've plugged in the rule scores computed by the neural network. The symbol $\oplus$ is a generic operator that is max if we're looking for the best parse and logaddexp if we want the total score of all parses.

for all $0 \le i < j \le n$ **do**
  for all $X \in \Gamma$ **do**
    $chart[i, j][X] \leftarrow -\infty$
    $chart[i, j][\bar{X}] \leftarrow -\infty$
  end for
end for

▷ rules of the form $X \rightarrow w_i$
for all $i \leftarrow 1, \ldots, n-1$ **do**
  for all $X \in \Gamma$ **do**
    $chart[i-1, i][X] \leftarrow \mathbf{O}_{i,X}$
  end for
end for

▷ rule $\bar{\text{E}} \rightarrow \texttt{EOS}$
$chart[n-1, n][\bar{\text{E}}] \leftarrow \mathbf{O}_{n,\text{E}}$

▷ rules of the form $\bar{X} \rightarrow X\ \bar{Y}$
for $\ell \leftarrow 2, \ldots, n$ **do**
  for $i \leftarrow 0, \ldots, n-\ell$ **do**
    $j \leftarrow i + \ell$
    for $k \leftarrow i+1, \ldots, j-1$ **do**
      for all $X \in \Gamma$ **do**
        for all $Y \in \Gamma$ **do**
          $chart[i, j][\bar{X}] \leftarrow chart[i, j][\bar{X}] \oplus (\mathbf{T}_{X,Y} + chart[i, k][X] + chart[k, j][\bar{Y}])$
        end for
      end for

```
        end for
    end for
end for
return chart[0, n][S̄]
```

Linear time. This is $O(n^3)$, but we would like to reduce this to $O(n)$. Remember that the cubic time complexity comes from the triple loop involving i, j, and k. But in the trees that our grammar generates, it's always the case that if $j - i > 1$, then $i \leq n-2$, $k = i+1$, and $j = n$. We didn't even define the rule-scoring function for other values of i and j. So the above triple loop can be rewritten as a single loop over i.

Moreover, no cell $chart[i, j]$ has entries for both X and $\bar{X}$. So we can drop the distinction between X and $\bar{X}$. This gives us the following algorithm:

```
for all 0 ≤ i < j ≤ n do
    for all X ∈ Γ do
        chart[i, j][X] ← −∞
    end for
end for
for all i ← 1, . . . , n − 1 do
    for all X ∈ Γ do
        chart[i − 1, i][X] ← O_{i,X}
    end for
end for
chart[n − 1, n][E] ← O_{n,E}
for i ← n − 2, . . . , 0 do
    for all X ∈ Γ do
        for all Y ∈ Γ do
            chart[i, n][X] ← chart[i, n][X]
                                ⊕ (T_{X,Y} + chart[i, i + 1][X] + chart[i + 1, n][Y])
        end for
    end for
end for
```

Vectorization. The other thing that is special about our grammar is that it's very *dense*, in the sense that it has a rule $X \to a$ for every single X, and a rule $\bar{X} \to X\ \bar{Y}$ for every single X and Y. Instead of thinking of $chart[i, j]$ as a hash table from labels to numbers, we think of it as a vector of numbers. Instead of all those loops over labels, we can now use vector operations.

Modified 2023-11-08 to make the pseudocode less pseudo

The loop at line 2 can be replaced with a single call to `torch.full()`. Similarly, the loop at line 7 can be replaced with a single assignment.

The double loop at line 13 can also be replaced with tensor operations. In pseudo-PyTorch, the sum $(\mathbf{T}_{X,Y} + chart[i, i+1][X] + chart[i+1, n][Y])$ becomes

$$\mathbf{C} \leftarrow \mathbf{T} + \texttt{unsqueeze}(chart[i, i+1], 1) + chart[i+1, n]$$

which has size $|\Gamma| \times |\Gamma|$. Then, during training, we compute $chart[i, n]$ as the `logaddexp` of all the columns of $\mathbf{C}$:

$$chart[i, n] \leftarrow \texttt{logsumexp}(\mathbf{C}, \texttt{dim} = 1)$$

During parsing (which in the context of sequence labeling is called decoding or inference), we want to find the elementwise max of all the columns of $\mathbf{C}$:

$$chart[i, n], back[i, n] \leftarrow \texttt{max}(\mathbf{C}, \texttt{dim} = 1)$$

PyTorch's `max` function returns a pair of tensors: the first contains the maximum values, and the second contains the indices of the maximum values (the argmaxes).

Reconstructing the best label sequence is just like reconstructing the best parse tree in CKY. Namely, $back[i, n]_X = Y$ means that the best labeling of $w_{i+1} \cdots w_n$ that starts with X continues with Y. So $back[0, n]_S$ tells you the first label in the best label sequence (call it X_1), and $back[1, n]_{X_1}$ tells you the second label in the best label sequence, and so on. Corrected 2023-11-16

One final note: It's a little weird that this algorithm runs right-to-left. If we had made our original tree left-branching instead of right-branching, the final algorithm would run left-to-right.

7.2.2 Abstract Meaning Representations

Above we mentioned various semantics-related tasks like semantic role labeling (Gildea and Jurafsky, 2000), word sense disambiguation (Brown et al., 1991), coreference resolution (Soon, Ng, and Lim, 2001), and so on. Resources like OntoNotes (Hovy et al., 2006) provided separate resources for each of these tasks.

Some more recent work in semantic processing tries to consolidate these tasks into one. For example, the Abstract Meaning Representation (AMR) Bank (Banarescu et al., 2013) began as an effort to unify the various annotation layers of OntoNotes. Others include: the Prague Dependency Treebank (Böhmová et al., 2003), DeepBank (Oepen and Lønning, 2006), and Universal Conceptual Cognitive Annotation (Abend and Rappoport, 2013). By and large, these resources are based on, or equivalent to, *graphs*, in which vertices stand for entities and edges stand for semantic relations among them.

Data format

Here, I'll focus on AMRs, just because they're the representation I'm most familiar with. AMRs can be written in a serialized form or as directed graphs. Examples of these two representations, from the AMR Bank (LDC2014T12), are reported in Figure 7.1 and Figure 7.2. Nodes are labeled, in order to convey lexical information. Edges are labeled to convey information about semantic roles. Labels at the edges need not be unique, meaning that edges impinging on the same node might have the same label. Furthermore, our DAGs are not ordered, meaning that there is no order relation for the edges impinging at a given node, as is usually the case in standard graph structures. A node can appear in more than one place (for example, in Figure 7.1, node `s2` appears six times).

The numbers (e.g., ask-01) require some explanation. These are from PropBank (Palmer, Gildea, and Kingsbury, 2005), which catalogues and numbers, for each verb, the different senses of the verb and ways it can be used. For example,

- ask-01 is for asking questions

```
(a / and
      :op1 (a2 / ask-01
            :ARG0 (i / i)
            :ARG1 (t / thing
                  :ARG1-of (t2 / think-01
                        :ARG0 (s2 / she)
                        :ARG2 (l / location
                              :location-of (w / we))))
            :ARG2 s2)
      :op2 (s / say-01
            :ARG0 s2
            :ARG1 (a3 / and
                  :op1 (w2 / want-01 :polarity -
                        :ARG0 s2
                        :ARG1 (t3 / think-01
                              :ARG0 s2
                              :ARG1 l))
                  :op2 (r / recommend-01
                        :ARG0 s2
                        :ARG1 (c / content-01
                              :ARG1 i
                              :ARG2 (e / experience-01
                                    :ARG0 w))
                        :ARG2 i))
            :ARG2 i)
      :op3 c)
```

Figure 7.1: Example AMR in its standard format, number DF-200-192403-625_0111.7 from the AMR Bank. The sentence is: "I asked her what she thought about where we'd be and she said she doesn't want to think about that, and that I should be happy about the experiences we've had (which I am)."

- ask-02 is for asking favors
- ask-03 is for asking a price
- ask_out-04 is for asking someone on a date.

Each of these senses comes with a numbered list of arguments. For example, for ask-01,

- arg0 is the asker
- arg1 is the question
- arg2 is the hearer.

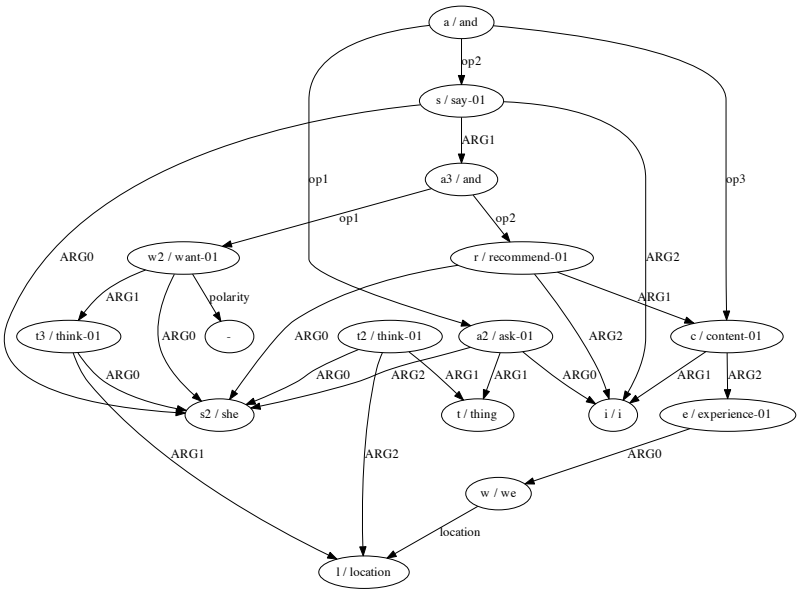


Figure 7.2: The AMR of Figure 7.1, presented as a directed graph.

AMR parsing

Semantic parsing is the task of taking a natural language sentence and mapping it to a representation of its meaning. If the semantic representation is AMR, we call this AMR parsing.

These days, it's easy to build a barebones AMR parser – just run a neural machine translation system on parallel text consisting of English sentences and their AMRs in textual format (Figure 7.1). Research on AMR parsing has gotten plenty more sophisticated than that, but here I want to focus on one improvement to the basic NMT system (which you will implement in HW4).

Unlike language-to-language translation, it's typical for the AMR to have words in common with the source text. The NMT system can do a good job learning to copy these words if they're frequent, but for rare words, it may do so less reliably, and for unknown words, it will be unable to. So we want to add a *copy* mechanism to the model.

The basic idea is to introduce a fake target word, `COPY`, which instructs the system to copy a word from the source sentence. Which word? We use the source-to-target attention, which is a distribution over source positions, to choose one source word.

We're only interested in the last two steps of the model, which were the same in our presentation of both the RNN and Transformer models. These are for the

context vector $\mathbf{c}^{(i)}$, and the output word distribution:

$$\begin{aligned}
\mathbf{g}^{(i)} &\in \mathbb{R}^d \\
\mathbf{H} &\in \mathbb{R}^{n \times d} \\
\mathbf{o}^{(i)} &\in \mathbb{R}^d \\
\mathbf{c}^{(i)} &\in \mathbb{R}^d \\
\mathbf{c}^{(i)} &= \text{Attention}(\mathbf{g}^{(i)}, \mathbf{H}, \mathbf{H}) && (7.10) \\
P(e_{i+1}) &= \text{SoftmaxLayer}^{\boxed{6}}(\mathbf{o}^{(i)}). && (7.11)
\end{aligned}$$

The equation for $\mathbf{c}^{(i)}$ computes both the attention and the weighted average, so we're going to "break it open" to get at the attention inside:

$$\begin{aligned}
\alpha^{(i)} &\in \mathbb{R}^n && (7.12) \\
\alpha^{(i)} &= \text{softmax}\, \mathbf{H}\mathbf{g}^{(i)} && (7.13)
\end{aligned}$$

The output word distribution now includes `COPY`. We modify this to:

$$\begin{aligned}
\mathbf{p}^{(i)} &\in \mathbb{R}^n && (7.14) \\
\mathbf{p}^{(i)} &= \text{SoftmaxLayer}^{\boxed{6}}(\mathbf{o}^{(i)}) && (7.15) \\
P(e) &= \mathbf{p}_e^{(i)} + \mathbf{p}_{\texttt{COPY}}^{(i)} \sum_{\substack{j=1,\ldots,n \\ f_j = e}} \alpha_j^{(i)}. && (7.16)
\end{aligned}$$

This means that there are one or more ways of choosing word e: first, we could choose it directly from the output distribution $\mathbf{o}^{(i)}$, or, for each source word f_j that is equal to e, we could copy word f_j with probability $\mathbf{p}_{\texttt{COPY}}^{(i)} \alpha_j$.

Note that

- $\alpha^{(i)}$ and $\mathbf{p}^{(i)}$ are both vectors of probabilities, not log-probabilities.
- The test $f_j = e$ compares the words *as words*, not as numbers.

7.3 Logic

The last category of semantic representations is that of logical formulas, understood broadly to include not only logics like first-order logic, but languages like SQL or even programming languages.

In these notes, I'd like to focus on a traditional approach to semantics called *Montague grammar*.

7.3.1 Logical forms

We start with a very simple example:

(7.17) a. John sees Mary.

b. $\mathit{see}(\mathit{John}, \mathit{Mary})$.

Entities are represented by constants or variables, and events are represented by predicates.

A variation (called neo-Davidsonian semantics) represents events by variables, too:

(7.18) a. John sees Mary.

b. $\exists e.see(e) \wedge agent(e, John) \wedge theme(e, Mary)$.

This is quite similar to AMR. But let's stick with events as predicates.

A key way that logical semantics differs from graph representations like AMR is in handling of quantifiers.

(7.19) a. John sees a girl.

b. $\exists g.girl(g) \wedge see(John, g)$.

(7.20) a. A boy sees Mary.

b. $\exists b.boy(b) \wedge see(b, Mary)$.

7.3.2 Compositionality

How do we compute these representations? We want to follow the principle of *compositionality*, that the meaning of any expression is computed from the meaning of its subexpressions. In other words, we want to write a recursive function that processes a syntax tree bottom-up, something like

function SEMANTICS(root)
  if root = S **and** root.children = (NP, VP) **then**
    $s_1 \leftarrow$ SEMANTICS(root.children[1])
    $s_2 \leftarrow$ SEMANTICS(root.children[2])
    build s from s_1 and s_2
    return s
  else...
  end if
end function

So we want to associate with each context-free grammar rule (e.g., S $\rightarrow$ NP VP) a little function that build the semantics of S from the semantics of NP and VP. To do that, it will be convenient to have some new notation for writing little functions.

7.3.3 Lambda calculus

A λ-expression (lambda-expression) is a self-contained way of writing a function. Many programming languages now have them:

λ-calculus	$\lambda x.x \cdot x$
Scheme/Lisp	`(lambda (x) (* x x))`
Python	`lambda x: x * x`
C++	`[](float x) { return x * x; }`

In λ-calculus, the application of a function f to an expression e is simply written as fe. So

$$(\lambda x.x \cdot x)10 \longrightarrow 10 \cdot 10 = 100.$$

Lambda expressions can do a lot of things you might not expect them to be able to do at first; here, I want to mention just one. Traditionally, λ-expressions take exactly one argument. But you can effectively write a function of two arguments as a function that returns another function, like this:

$$f = \lambda x.\lambda y.\sqrt{x^2 + y^2} \tag{7.21}$$

$$f\ 3\ 4 = (\lambda x.\lambda y.\sqrt{x^2 + y^2})\ 3\ 4 \tag{7.22}$$

$$\longrightarrow (\lambda y.\sqrt{3^2 + y^2})\ 4 \tag{7.23}$$

$$\longrightarrow \sqrt{3^2 + 4^2} \tag{7.24}$$

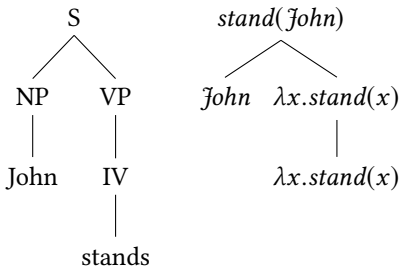
$$= 5. \tag{7.25}$$

This is called *currying* after Haskell Curry, who had nothing to do with it.

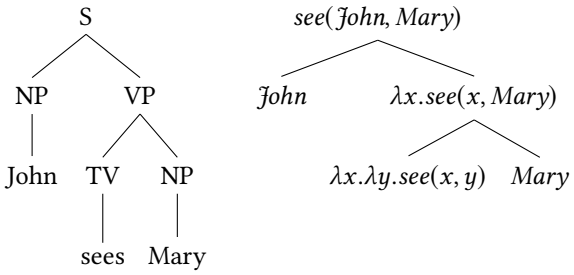
7.3.4 Examples

Here's a very simple CFG, each of whose rules is associated with a function that computes the semantics of the left-hand side (that is, the parent) in terms of the semantics of the right-hand side (that is, the children):

S $\rightarrow$ NP VP	$\lambda x.\lambda P.Px$
NP $\rightarrow$ John	John
NP $\rightarrow$ Mary	Mary
VP $\rightarrow$ IV	$\lambda P.P$
IV $\rightarrow$ stands	$\lambda x.\mathit{stand}(x)$

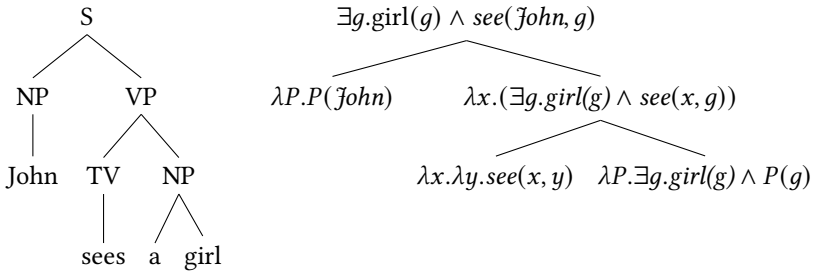


S $\rightarrow$ NP VP	$\lambda x.\lambda P.Px$
NP $\rightarrow$ John	John
NP $\rightarrow$ Mary	Mary
VP $\rightarrow$ IV	$\lambda P.P$
VP $\rightarrow$ TV NP	$\lambda P.\lambda y.\lambda x.P(x, y)$
IV $\rightarrow$ stands	$\lambda x.\mathit{stand}(x)$
TV $\rightarrow$ sees	$\lambda x.\lambda y.\mathit{see}(x, y)$



When we try to get examples like (7.19–7.20), however, problems arise. The meaning of "a" should be $\exists$, but how do we get this quantifier to appear on the very outside of the formula? The solution is to flip everything around so that the semantics for "a boy" should not just be a formula representing a boy; it should be a function that takes a predicate P about boys and returns the formula $\exists b.boy(b) \wedge P(b)$.

S $\rightarrow$ NP VP	$\lambda f.\lambda P.fP$
NP $\rightarrow$ John	$\lambda P.P(John)$
NP $\rightarrow$ Mary	$\lambda P.P(Mary)$
NP $\rightarrow$ a boy	$\lambda P.(\exists b.boy(b) \wedge Pb)$
NP $\rightarrow$ a girl	$\lambda P.(\exists g.girl(g) \wedge Pg)$
VP $\rightarrow$ TV NP	$\lambda P.\lambda f.\lambda x.f(\lambda y.Pxy)$
TV $\rightarrow$ sees	$\lambda x.\lambda y.see(x, y)$



The computation of VP is particularly complicated, so we write it out step by

step:

$$
\begin{aligned}
&\underline{(\lambda P.\lambda f.\lambda x.f(\lambda y.Pxy))(\lambda x.\lambda y.see(x,y))}(\lambda P.(\exists g.girl(g) \wedge Pg)) \\
&\longrightarrow (\lambda f.\lambda x.f(\lambda y.\underline{(\lambda x.\lambda y.see(x,y))xy}))(\lambda P.(\exists g.girl(g) \wedge Pg)) \\
&\longrightarrow (\lambda f.\lambda x.f(\lambda y.\underline{(\lambda y.see(x,y))y}))(\lambda P.(\exists g.girl(g) \wedge Pg)) \\
&\longrightarrow \underline{(\lambda f.\lambda x.f(\lambda y.see(x,y)))(\lambda P.(\exists g.girl(g) \wedge Pg))} \\
&\longrightarrow \lambda x.\underline{(\lambda P.(\exists g.girl(g) \wedge Pg))(\lambda y.see(x,y))} \\
&\longrightarrow \lambda x.(\exists g.girl(g) \wedge \underline{(\lambda y.see(x,y))g}) \\
&\longrightarrow \lambda x.(\exists g.girl(g) \wedge see(x,g)).
\end{aligned}
$$

Finally, we refine our grammar so that "a" has its own semantics.

S → NP VP	$\lambda f.\lambda P.fP$
NP → John	$\lambda P.P(\mathit{John})$
NP → Mary	$\lambda P.P(\mathit{Mary})$
NP → Det N	$\lambda d.\lambda f.df$
Det → a	$\lambda N.\lambda P.(\exists x.Nx \wedge Px)$
Det → every	$\lambda N.\lambda P.(\forall x.Nx \wedge Px)$
N → boy	$\lambda b.boy(b)$
N → girl	$\lambda g.girl(g)$
VP → TV NP	$\lambda P.\lambda f.\lambda x.f(\lambda y.Pxy)$
TV → sees	$\lambda x.\lambda y.see(x,y)$

References

Abend, Omri and Ari Rappoport (2013). "Universal Conceptual Cognitive Annotation (UCCA)". In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Sofia, Bulgaria, pp. 228–238. URL: http://www.aclweb.org/anthology/P13-1023.

Banarescu, Laura et al. (2013). "Abstract Meaning Representation for Sembanking". In: *Proceedings of the Linguistic Annotation Workshop*. Sofia, Bulgaria, pp. 178–186.

Böhmová, Alena et al. (2003). "The Prague Dependency Treebank: A Three-Level Annotation Scenario". In: *Treebanks: Building and Using Parsed Corpora*. Ed. by A. Abeillé. Kluwer, pp. 103–127.

Brown, Peter F. et al. (1991). "Word-sense disambiguation using statistical methods". In: *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics (ACL-91)*. Berkeley, CA, pp. 264–270.

Gildea, Daniel and Daniel Jurafsky (2000). "Automatic Labeling of Semantic Roles". In: *Proceedings of the 38th Annual Conference of the Association for Computational Linguistics (ACL-00)*. Hong Kong, pp. 512–520.

Hovy, Eduard et al. (2006). "OntoNotes: The 90% Solution". In: *Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers*. New York City, USA, pp. 57–60. URL: http://www.aclweb.org/anthology/N/N06/N06-2015.

Oepen, Stephan and Jan Tore Lønning (2006). "Discriminant-Based MRS Banking". In: *International Conference on Language Resources and Evaluation (LREC)*. Genoa, pp. 1250–1255.

Palmer, Martha, Daniel Gildea, and Paul Kingsbury (2005). "The Proposition Bank: An Annotated Corpus of Semantic Roles". In: *Computational Linguistics* 31.1, pp. 71–106. DOI: 10.1162/0891201053630264. URL: https://www.aclweb.org/anthology/J05-1004.

Soon, Wee Meng, Hwee Tou Ng, and Daniel Chung Long Lim (2001). "A Machine Learning Approach to Coreference Resolution of Noun Phrases". In: *Computational Linguistics* 27.4, pp. 521–544.