

600.465 — Intro to NLP

Assignment 4: Semantics

Prof. J. Eisner — Fall 2009
Due date: Tuesday 10 November, 2 pm

In this assignment, you will run your output parses from Assignment 3 through a post-processing script that computes their features...including semantic features. You will be asked to understand and tweak the grammar that assigns the features.

No programming is involved, just thinking.

Note: This split into two assignments is a bit artificial. Features are really part of the context-free grammar, so your parser in Assignment 3 should arguably have considered them while parsing in the first place. On the other hand, ignoring the features made your parser simpler and faster.

It is pretty common in NLP to use this kind of simple but artificial “pipeline” approach, where the input is passed through a sequence of separate programs, each of which enriches the input with some new kind of annotation before passing it on to the next program for further processing. For example, given a speech input, program #1 transcribes it into a sequence of words, program #2 divides those words into morphemes, program #3 disambiguates those morphemes with part of speech tags, program #4 picks the best parse that is consistent with those part of speech tags, and program #5 computes semantics from the parse.

This “pipeline” approach does lose some accuracy. Each stage in the pipeline outputs only its *single* best guess, and makes that guess without paying attention to the linguistic considerations at further stages. For example, the speech recognizer (program #1) might have considered other transcriptions that would have admitted a better parse (program #4)—but unfortunately, it discarded those transcriptions as suboptimal before anyone considered their syntax. For this reason, NLP pipelines are starting to go out of fashion as the community has discovered better ways to coordinate the behavior of separate modules such as #1–#5.

What is lost by the “tree first, features second” pipeline in Assignments 3–4? In principle, it could have improved accuracy if the parser had computed constituents’ features *during* parsing, as humans probably do. Then the rule probabilities could have depended on the nonterminal features. In particular, feature mismatch would have allowed the parser to rule out some options, or give them lower probabilities (e.g., it’s impossible or unlikely to combine a singular subject with a plural verb).

<code>*.grf</code>	full grammar with rule <i>frequencies</i> , features, comments
<code>*.gr</code>	simple grammar with rule <i>weights</i> , no features, no comments
<code>delfeats</code>	script to convert <code>.grf</code> \rightarrow <code>.gr</code>
<code>*.sen</code>	collection of sample sentences (one per line)
<code>*.par</code>	collection of sample parses
<code>checkvocab</code>	script to detect words in <code>.sen</code> that are missing from the grammar <code>.gr</code>
<code>parse</code>	your program from Assignment 3 that converts <code>.sen</code> $\xrightarrow{.gr}$ <code>.par</code>
<code>prettyprint</code>	script to reformat <code>.par</code> more readably
<code>buildfeats</code>	script to convert <code>.par</code> $\xrightarrow{.grf}$ a feature assignment
<code>parsefeats</code>	simple script to convert <code>.sen</code> $\xrightarrow{.grf}$ a feature assignment (calls <code>checkvocab</code> , <code>parse</code> , and <code>buildfeats</code>)
<code>simplify</code>	script that lets you experiment with lambda terms

Figure 1: Files available to you for this project. The ones above the double line were already provided in Assignment 3.

You’ll be interested to know that the parsing community has swung back and forth on whether that is a good use of runtime. Features were heavily used in the old days to reduce the number of parses. But once we started using probabilities instead to pick the best parse, people went back to simpler nonterminals without features, as in the Penn Treebank (see `wallstreet.gr`). That’s because syntactic features such as **agreement** rarely help for choosing among the *most probable* parses of a sentence.¹ Thus, most modern probabilistic English parsers compute little more than the **head** feature while parsing (since conditioning the probabilities on the head feature does help parsing accuracy). Only in the past few years have a few researchers shown parsing benefit by picking the “right” nonterminal features.²

The more detailed **semantic** features that we will consider in this assignment could theoretically help parsing as well—but only in a system that can reason about the semantics and relate it to a database of knowledge about the world in order to decide whether a constituent is plausible.

All the new files you need can be found in <http://cs.jhu.edu/~jason/465/hw4>, or in `/usr/local/data/cs465/hw4` on the `ugrad` machines. You can download the files individually as you need them, or download a zip archive that contains all of them. Read Figure 1 for a guide to the old and new files.

¹Of course, they are still needed to define grammaticality and to generate grammatical sentences with `randsent`.

²And they’ve done it not by using human-crafted features in a grammar or a treebank, but rather by *automatically* discovering features (i.e., more refined nonterminals) that help model the training data better. It’s cool that many of these automatically learned features do appear to capture linguistic notions such as singular vs. plural or noun vs. pronoun.

As in Assignment 3, you should actually look inside each file as you prepare to use it! For the scripts, you don't have to understand the code, but do read the introductory comments at the beginning of each script.

1. Your first job is to understand the notation for adding features to a grammar. A grammar with features is a `.grf` file; the corresponding `.gr` file can be produced by using `delfeats` to strip the features and comments. You have been given some simple `.grf` files that demonstrate the different features of the notation.
 - (a) Read the file `arith.grf` carefully and examine the output of the following commands, especially the part of the output that is *not* indented:

```
parse arith.gr arith.sen > arith.par
buildfeats arith.grf arith.par
```

Parsing speed will not be a big issue in Assignment 4, so you can use your `parse` from Assignment 3 rather than `parse2`. If you weren't even able to get `parse` working correctly, then contact the course staff, and we will send you a copy of the correct `arith.par` output produced by the first command.

The output of `buildfeats` for each parse is an indented trace, showing how the features for each constituent are built bottom-up. The traces for different parses are separated by ---.

At the end of a trace (not indented) is the final result: the features for the parse as a whole. This is what you should usually study, but if something is mysterious you can look earlier in the trace to see how the parse's features arose from those of smaller constituents.

- (b) Now study `arith-infix.grf` and try

```
buildfeats arith-infix.grf arith.par
```
- (c) Finally, study `arith-typed.grf` and try

```
parse arith.gr arith-typed.sen > arith-typed.par
buildfeats arith-typed.grf arith-typed.par
```

Note that you can abbreviate this process using the `parsefeats` script, which also does some other nice things for you (look at the script to see what!):

```
parsefeats arith-typed.grf arith-typed.sen
```

There is nothing to hand in for this question—just make sure you understand what's going on before it gets more confusing! Feel free to discuss with others.

2. `times(x,y)` is all very well, but to build interesting natural-language semantics we are going to have to use lambda terms. So here are some simple exercises—you don't

have to hand the answers in. Again, feel free to discuss with others, although you may want to try them on your own first.

You can check your answers using the `simplify` script, which will simplify any lambda-expression you type in. (Look at the top of the script for documentation. Start it by typing `./simplify` with *no arguments*.) But try to come up with each answer on your own first ...

A newer tool you could try is the Penn Lambda Calculator (<http://www.ling.upenn.edu/lambda/>), “an interactive, graphical, pedagogical computer program that helps students of formal semantics practice the typed lambda calculus.” It helps you through some exercises. If you try it, feel free to ask for help, and definitely let me and the class know how you like it!

- (a) Simplify $(\lambda x x * x)3$.
- (b) Simplify $(\lambda x x * x)(y + y)$.
- (c) Simplify $(\lambda x x * x)y + y$.
- (d) Simplify $(\lambda a a)(\lambda b f(b))$.
- (e) Simplify $(\lambda a 3)(\lambda b f(b))$.
- (f) Simplify $(\lambda x green(x))(y)$. Since the result holds for any y , what do you conclude about the relation between $\lambda x green(x)$ and $green$?
- (g) Simplify $(\lambda x \lambda y ate(x, y))(lemur, leopard)$.
- (h) Simplify $(\lambda x \lambda y ate(x, y))(lemur)$.
- (i) Apply the previous answer to “leopard”: simplify $(\lambda x \lambda y ate(x, y))(lemur)(leopard)$.
- (j) Simplify $(\lambda x f(x, y))(a)(b)(c(z))$.
- (k) Simplify $(\lambda x f(x, y))(a, b, c(z))$. This is just an abbreviation for the previous case.
- (l) Simplify $(\lambda f f(x))g$.
- (m) Simplify $(\lambda f f(f(f(x))))g$.
- (n) Simplify $(\lambda f f(f(f(x))))(\lambda t a(c(t)))$.
- (o) Simplify $(\lambda f f(f(f(x))))(\lambda t t * t)$.
- (p) Simplify $(\lambda f f(f(f(x))))(\lambda t a(b, c[t], d))$.

Feel free to play around more with `simplify`. You can actually do some outrageous things with it, including using lambda terms to represent integers, pairs, stacks, conditionals, recursion, loops, and in fact any Turing machine. (Can you write an expression whose simplification doesn't terminate?) More information is at the top of the file `LambdaTerm.pm`.

3. From here to the end of the assignment, you should work by yourself and hand your answers in. (Put the answers in your **README**, using the same notation used by **simplify**. You can use **simplify** to check your answers.)

Several of these are basically division problems (analogous to “If $x \cdot 3 = 21$, what is x ?”). For example, if $f(6) = 6 \cdot 6$, then what is f ? Answer: $f = \lambda x x \cdot x$. That’s all there is to it.³

(These “division” problems are related to the end of the semantics lecture. We wanted a particular meaning for “Every nation wants George to love Laura,” and worked backwards to figure out what functions f should be associated with the words. Those slides will make more sense once you’ve done this question.)

- (a) Suppose $f(\text{John}) = \text{loves}(\text{Mary}, \text{John})$. What is f ,
- written in the form $\lambda x \dots$?
 - written without any λ ?
- (For example, $(\lambda x x)(3)$ can be written as 3 . $\lambda x s(x)$ can be written as s .)
- (b) In our semantics, $\text{loves}(\text{Mary}, \text{John})$ will be the interpretation of “John loves Mary,” not vice-versa. This is just more convenient because then the VP in that sentence has a nice, compact semantics. Namely, what?
- (c) Suppose $f(\text{John}) = (\forall x \text{ woman}(x) \Rightarrow \text{loves}(x, \text{John}))$.
- What is f ?
 - Translate f and $f(\text{John})$ into English.
- Note:* To type an expression such as $\forall x \text{ woman}(x) \Rightarrow \text{loves}(x, \text{John})$ into the **simplify** script, write something like **A%x woman(x) => loves(x, John)**.⁴ Notating \forall as **A%** (or anything ending in **%**) tells **simplify** that the following x is a dummy variable, not a constant.
- (d) Suppose $f(\lambda x \text{ loves}(\text{Mary}, x)) = (\lambda x \text{ Obviously}(\text{loves}(\text{Mary}, x)))$. What is f and how would you use it in constructing the semantics of “Sue obviously loves Mary?”
- (e) Suppose $f(\lambda x \text{ loves}(\text{Mary}, x)) = (\forall y \text{ woman}(y) \Rightarrow \text{loves}(\text{Mary}, y))$.
- What is f ?
 - Translate $f(\lambda x \text{ loves}(\text{Mary}, x))$, $(\lambda x \text{ loves}(\text{Mary}, x))$, and f into English.
- (f) Let f be your answer from question **3(e)i**. Suppose $g(\text{woman}) = f$.

³Other possible answers are $f = \lambda x 6 \cdot x$, $f = \lambda x x + 30$, and $f = \lambda x 36$. These are technically correct, since $f(6) = 36$ in all these cases. But $f = \lambda x x \cdot x$ is the answer we’d be looking for.

⁴This particular expression cannot be simplified further by **simplify**, so don’t be alarmed if you type it in and it comes right back at you.

- i. What is g as a lambda term?
- ii. What English word does it represent?

Hint: Substituting $g(woman)$ for f in question 3e yields $g(woman)(\lambda x \text{ loves}(Mary, x)) = (\forall y \text{ woman}(y) \Rightarrow \text{loves}(Mary, y))$. If you replaced every other term in this equation with the English phrase of which it is the semantics, then what would you have to replace g with?

- (g) Suppose $f(\lambda x \text{ loves}(Mary, x)) = \text{loves}(Mary, Papa)$.
 - i. What is f as a lambda term?
 - ii. Why would one want to give Papa these funny semantics (rather than just `sem=Papa`, as in the original `english.grf`)? (*Hint:* Look back at question 3e, translate both expressions into English, and think “consistency.”)

4. Now you’re ready to look at a (small) English semantic grammar: study `english.grf`. The syntactic coverage is nowhere near that of the Penn Treebank’s grammar, but it does have semantics.

Try running (as in question 1c)

```
parsefeats english.grf english.sen
```

This will convert the grammar to a `.gr` file, parse some English sentences using *your* Earley parser, and then assign features with `buildfeats`.

Each of the 22 sentences in `english.sen` should have yielded a parse under `english.gr`. For each sentence, inspect its features and decide whether they are appropriate:

- For a grammatical sentence, did the system find the most plausible semantics?
- For an ungrammatical sentence, did the system print the message “there is no consistent way to assign features”?

List the sentences where you think the feature assignment may be inappropriate, and explain why. In each case, say whether it would have helped if the parser had chosen a different valid parse of the same sentence. (Remember, the parser uses probabilities but without considering features, and then `buildfeats` is stuck computing features for whatever the parser chose.) If so, what parse would have worked better?

As before, if you are not sure that your own parser from Assignment 3 is working correctly, you might want to contact the course staff for an `english.par` file containing the correct parses, which you could then run through `buildfeats`

```
english.grf english.par.
```

For example, if the sentence is

Meilin saw a bird with the telescope

then you should notice a problem if the representation is

```
Past(see(a(%x bird(x) ^ with(the(telescope),x)),Meilin))
```

since that says that the bird has the telescope. Probably this is *not* the semantics that the author of the sentence intended. A different parse would have gotten the correct semantics.

Important: Don't kill yourself. You don't have to pore over the features for hours with a monocle and tweezers. Just try to find the major problems and briefly say why they are problems. We won't penalize you for missing a few. The point of this problem is not to torture you, only to make you stare at the output long enough to Understand™ what's going on and make some intelligent comments.

Certainly you do not have to second-guess the *style* of the representations. That is,

```
Past(with(the(telescope),see(a(bird),Meilin)))
```

may not be the ideal semantic representation, since the handling of prepositions, determiners, and tense is pretty primitive. But it is reasonable enough that you needn't take issue with it.

5. In `english.grf` and `english.sen`, Papa is eating bonbons rather than caviar. This is because I couldn't figure out whether *caviar* was singular or plural. You can say "All caviar is delicious"—but *all* only combines with plural nouns, whereas *is* only combines with singular nouns . . . so how can *caviar* do both?

In fact, *caviar* (like *chocolate* and *dirt* and *camera film*) is what is called a "mass noun." Modify `english.grf` to admit *three* values for the `num` feature: `sing`, `pl`, and `mass`. Add *caviar* as a mass noun. Make sure that mass nouns work correctly both with verbs (which always treat them as singular) and with determiners (which don't).

To do this, you'll need to work out the facts about which determiners can go with which nouns. You may want to make a grid of determiners versus nouns and see which ones can combine, using the vocabulary in `english.grf`. You'll notice that mass determiners are always plural determiners as well (*all caviar* → *all bonbons*) but not vice-versa (*two bonbons* ↯ **two caviar*).⁵

Try to handle these inelegant facts elegantly, using as small and simple a system of rules as you can under the circumstances. Submit your modified `english.grf`. Run

⁵It would be nice to capture this asymmetric generalization with a rule like $N[\text{num=pl}] \rightarrow N[\text{num=mass}]$, which says that mass determiners can always be used where plural determiners are called for. One could similarly write $NP[\text{num=sing}] \rightarrow NP[\text{num=mass}]$, which says that mass NPs can be used to agree with singular verbs or singular pronouns. Unfortunately, these elegant rules introduce an extra NP node into the tree when mass nouns are involved. That would require the shape of the tree to be affected by the `num` features. So they won't work with our pipeline system, which parses *before* it looks at the features.

it on a few sentences about caviar—both grammatical and ungrammatical ones—and report what happened.

6. `english.grf` doesn't attempt any real semantics for determiners. In particular, quantifiers like “every” are left as atomic elements with no internal semantics.

`english-fullquant.grf` fixes this, reorganizing the grammar along the lines you explored in questions 3e-3g.⁶ At the end of the semantics lectures, we also handled “every nation” in this style—you could review those slides.

Try `parsefeats english-fullquant.grf english.sen` to see the new form of the output. Study `english-fullquant.grf` to see how it's done; just look at the changes, which are marked with `***`.

As before, if you are not sure that your own parser from Assignment 3 is working correctly, you might want to contact the course staff for an `english-fullquant.par` file containing the correct parses (according to the `english-fullquant` grammar, which is not quite identical to the `english` grammar). You could then run these parses through `buildfeats english-fullquant.grf english-fullquant.par`.

- (a) The new grammar gives pretty complicated semantic features to *two* and to singular and plural *the*. Justify the features it uses (i.e., explain what those lambda-terms mean). The `!` symbol means “not.”
- (b) The semantics of one rule in the new grammar has been left as `???`. It affects the sentence *Papa want -ed George to eat a pickle*. What should replace the `???`? (Try your answer out, but see if you can get it without trial and error! It's hard to wrap your brain around, I know.)

⁶This approach (due to Montague) is called the “Proper Theory of Quantification” because it says that proper nouns have the same semantic type as NPs containing quantifiers.