Ahmad Elshenawy
Ling571 – Wi2014
February 27, 2014

Homework 5 – writeup

One problem I definitely had was getting some of the determiners to work. For example, for the determiner "every", I was running into a problem where nltk would give me an error saying "Expected logic expression", even though my entry for 'every' was structurally identical to my entry for 'all', which did not give me any errors:

```
DET[AGR=[NUM='pl'],SEM=<\Q P.all x.(Q(x) -> P(x))>] -> 'all'
DET[AGR=[NUM='sg'], SEM=<\Q P.all x.(Q(x) -> P(x))>] -> 'every'
```

At first I thought it might have to do with the AGR property I added into the entry, but I also had one for 'all', which wasn't giving me any errors, so that wasn't it. Ultimately, since I didn't really need to keep AGR for these, I removed it entirely from both, but I was still getting the same error:

```
DET[SEM=<\Q P.all x.(Q(x) -> P(x))>] -> 'all'
DET[SEM=<\Q P.all x.(Q(x) -> P(x))>] -> 'every'
```

What ended up fixing it for me was adding both words into the same line.

Another issue that came up for me was learning all of the proper symbols/terms to use so that nltk would work right. For example, for the entry for 'no', I took me some time to figure out that I needed something like "-exist", because originally I was using "!exist", which gave me the "Expected logic expression" error. Similarly for the adverb 'not', I had something like:

```
ADV[SEM=<\Q P x.(Q(x) NOT P(x))>] -> 'not'
```

But I couldn't get that to work either, so I opted to treat it as a negation function instead, which turned out better.

I struggled a bit figuring out a weird problem I had where I was somehow applying my lambda operators in the backwards order. Originally, I had entries like this:

```
V[SUBCAT=itv, SEM=<\e x.(eat(e) & eater(e,x))>] -> 'eat'
```

But I would end up with backwards parses like this:

```
\e.(eat(John) & eater(John,e))
```

I guess I must be flipping the switch somewhere down the line in my grammar, or I'm just getting myself confused and understand things backwards, but the only way I was able to fix this was by reversing the 'e' and 'x' lambdas in these entries, and in my AUX entries as well

```
V[SUBCAT=itv, SEM=<\x e.(eat(e) & eater(e,x))>] -> 'eat'
```

One thing that I am concerned about in my results file is the parse for the final sentence "John eats in Seattle", which resulted in the following semantic structure:

```
(\x e.(eat(e) & eater(e,x)) & \e.Location(e,Seattle))(John)
```

It might just be because I misunderstood something I heard Professor Levow say in lecture about floating lambdas, but I wanted to be sure I addressed this in case what I thought was right. This was the only parse I had where a variable wasn't directly applied to the lambda expression, where "John" is to the right of the expression and hasn't yet replaced any of the "X" variables in the expression. Once "John" is applied, I believe the expression works out just fine, but I thought I should bring it up. Partly this must have something to do with how I handle PP modifiers on VPs: I opted to just use these semantics:

```
VP[SEM=<?v & ?p>] -> VP[SEM=?v] PP[SEM=?p]
```

Where the PP is essentially tacked on to the existing VP semantics. I think it does the trick, but there's likely some more elegant way to handle this out there.

It goes without saying that this grammar is abhorrently small and doesn't account for all possible semantic interpretations (problems with scope, but boy am I glad we didn't need t mess around with that Cooper stuff from the book, it looked confusing indeed), nor would it really work for a number of examples. I tried to keep AGR in just as an exercise in continuity with the last homework, but some problems I had with "every" kept me from maintaining AGR throughout the grammar.