NLP HW 5
Sathvik Manikantan Napa Ugandhar (333E9A)

Question 3
The below constructions are all under the assumption that loves(Mary, John) translates to `John loves Mary` and not vice-versa. The second daughter to predicate is considered the subject and the first daughter is the object in this construction.

(a) Suppose f(John) = loves(Mary, John). What is f,

i. written in the form λx · · · ?

```
**Answer:** %x loves(Mary, x)
```

ii. written without any λ ? (For example, (λx x)(3) can be written as 3. λx s(x) can be written as s.)

```
**Answer:** loves(Mary)
```

(b) In our semantics, loves(Mary, 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?

```
**Answer:**
The VP "loves Mary" describes a predicate/function that takes the subject as an argument and returns a boolean output.
**%subj loves(Mary, subj)**
```

(c) Suppose f(John) = (A%x woman(x) ⇒ loves(x, John)).

i. What is f?

```
**Answer:** f = %y A%x woman(x) ⇒ loves(x, y))
```

ii. Translate f and f(John) into English.

Under the assumption: The second daughter to predicate is considered the subject and the first daughter is the object in this construction.

```
**Answer:**
**f =>** ___ loves every woman **(or) <the argument>** loves every woman
**f(John) =>** John loves all women
```

(d) Suppose f(λx loves(Mary, x)) = (λx Obviously(loves(Mary, x))). What is f and how would you use it in constructing the semantics of "Sue obviously loves Mary"? Hint: Review the pop/push slide near the end of the semantics lecture.

````
**Answer:**
f= %k (obviously(k))
````

To construct, "Sue obviously loves Mary"

f would take in verb `loves` as an argument as the adverbs modify the verbs. The arguments of love are then passed down to the entire construct.

(obviously(loves))(Mary, Sue) or
(obviously(loves)) (Mary) (Sue)
would be the appropriate constructs.
```

(e) Let's try using a Davidsonian event variable e. Suppose f(Mary)(John) = (λe act(e, loving), lovee(e, Mary), lover(e, John)). What is f?

```

Answer:
f = (%x %y %e act(e, loving), lovee(e, x), lover(e, y))
```

(f) Keep f as in the previous problem. Suppose g(f(Mary))(John) = (λe act(e, loving), lovee(e, M ary), lover(e, John), manner(e, passionate)). What is g? Hint: Write out f(Mary), which is the meaning of "love Mary." g(f(Mary)) will be the meaning of "passionately love Mary." Again, think about the pop/push trick.

```

Answer:
g = %V %s %e V(s)(e), manner(e,passionate)
```

(g) Suppose f(λx loves(Mary, x)) = (∀y woman(y) ⇒ loves(Mary, y)).

i. What is f?

```

Answer:
f= %V A%y woman(y) => V(y)
```

ii. Translate f(λx loves(Mary, x)), (λx loves(Mary, x)), and f into English.

``` **Answer:**
Translate f(λx loves(Mary, x)): all women love Mary.
Translate (λx loves(Mary, x)): <The subject (argument x)> loves Mary.
Translate f: the operator that turns a VP into 'every woman VP`
All women <does something>
```

(h) Let f be your answer from question 3(g)i. Suppose g(woman) = f.

i. What is g as a lambda term?

```

Answer:
g = %dom %V A%y dom(y) => V(y)
```

ii. What English word does it represent? Hint: Substituting g(woman) for f in question 3g yields g(woman)(λx loves(Mary, x)) = (∀y woman(y) ⇒ 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?

```

Answer:
It represents the the determiner "**every**"
```

(i) Suppose f(λx loves(Mary, x)) = loves(Mary, Papa).

i. What is f as a lambda term?
```

```

**Answer:**
f=%V V(Papa)
```

ii. Why would one want to give (NP Papa) the semantics sem=f (rather than just sem=Papa, as in the original english.gra)? (Hint: Look back at question 3g, translate both expressions loves(Mary, Papa) and (∀y woman(y) ⇒ loves(M ary, y)) into English, and remember that we'd like to treat similar phenomena in a consistent way.)

```

**Answer:**

- In order to loosely couple and treat **NP** as a predicate with more functionality and to not hard code it to a specific proper nouns. Doing it this way, gives the possibility to different kinds of subjects or objects **NP** can take.
- NP can now allow more variations of Det N (every, all, some, a, the) in addition to pure proper nouns/proper nouns.
- Hardcoding makes it restrictive and tight to a specific kind of NPs.
- For example, 3gi has "every woman" as NP construct instead of a hard coded proper noun such as Papa.

```

Question 4

Now you're ready to look at a (small) English semantic grammar: study english.gra. The syntactic coverage is nowhere near that of the Penn Treebank's grammar, but it does have semantics. Try running (as in question 1c) parseattrs english.gra english.sen This will convert the grammar to a .gr file, parse some English sentences using your Earley parser, and then assign attributes with buildattrs. 1Other possible answers are f = λx 6 · x, f = λx x + 30, and f = λx 36. These are technically correct, since f(6) = 36 in all these cases. But f = λx x · x is the answer we'd be looking for. 4 As before, you can use either your parse.py or parse2.py from Homework 4, or if you're not sure that those are working correctly, you can use the Homework 1 parser.

Each of the 22 sentences in english.sen should have yielded a parse under english.gr. For each sentence, inspect its attributes 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 attributes"?

List the sentences where you think the attributes 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 attributes, and then buildattrs is stuck computing attributes for whatever the parser chose.) If so, what parse would have worked better? 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 attributes 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 UnderstandTM 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.

Answers Below

1) Joe love -s Jill . [Seems Correct]

2) he love -s her . [Seems correct]

3) him love -s she . [seems incorrect]

Parse: assert(pres(love(her,him)), speaker)

```
This seems to be the effect of parsing without the attributes and attaching attributes later.  The sentence should have conditioned with attributes to take only the subject.
        S[=2 sem=pres(2(1))]    NP[num=N] VP[num=N tense=pres]
should instead be something like
        S[=2 sem=pres(2(1(subj)))]      NP[num=N case=subj] VP[num=N tense=pres])
```

4) Papa sleep -s with a spoon . [Seems correct]

5) Papa eat -ed with a spoon . [Seems fine]

6) Papa sleep -s every bonbon with a spoon . [Ungrammatical]

(Failed to attach semantic attributes since sleeps is intransitive as per grammar)

7) Papa eat -ed every bonbon with a spoon . [seems incorrect]

Parse: assert(past(eat(all(%x bonbon(x) ^ with(some(spoon),x)),Papa)), speaker)

```
Correct parse should have been:
assert(past(all(bonbon) (%x with(some(spoon), eat(x, Papa))), speaker)
```

8) have a bonbon ! [seems incorrect]

Parse: command(possess(some(bonbon))(hearer), speaker)

```
- This means to possess a bonbon according to the grammar. It doesn't mean `eat` a bonbon like it pragmatically conveys.
- In order to disambiguate `have` in this context of eating, we can add a finegrained semantic attribute to NP say, `eatable` to distinguish that the object is an eatable.
```

We define eatable Nouns as

1 N[=1 num=sing eatable=Y] bonbon

1 N[=1 num=sing eatable=Y] sandwich

1 N[=1 num=sing eatable=Y] pickle

1 NP[=2 num=N sem=1(2) eatable=Y] Det[num=N] N[num=N eatable=Y]

1 VP[=1 head=1 sem="%subj 2(%obj eat(obj, subj))"] V[head=have arg=np] NP[eatable=Y]

For plural Nouns

1 N[=1 num=pl eatable=Y] N[=1 num=sing eatable=Y] -s

The new parse would look like

command(some(bonbon) (%obj eat(obj, hearer)), speaker)```

9) a bonbon on the spoon entice -s . [ambiguous]
Parse: assert(pres(entice(something,some(%x bonbon(x) ^ on(the(spoon),x)))), speaker)
```
The verb `entice`, according to Merriam Webster, is a transitive verb. Under the assumption that entice is transitive, the parse is incorrect.
In that case,
`1 V[=1 tense=stem arg=none sem="%subj 1(something,subj)"]  entice   # intransitive` the rule should simply be removed from our gra file.

- **If entice is considered to be ambitransitive, the parse seems about fine**
``
10) a bonbon on the spoon entice -0 . [seems correct]
11) the bonbon -s on the spoon entice -0 . [seems correct]
**12) Joe kiss -ed every chief of staff .** [seems incorrectly parsed]
Parse: assert(past(kiss(all(chief_of_staff), Joe)), speaker)
    ```
    the quantifier all(chief of staff) should move before the VP
    assert(past(all(chief_of_staff) (%s kiss(s, Joe))), speaker)
    ````
**13) Jill say -s that Joe might sleep on the floor !** [Seems ambiguous]
Parse: exclaim(pres(on(the(floor),say(might(sleep(Joe)), Jill))), speaker)
    ```
    There are two interpretations to the sentence
    1. exclaim(pres(on(the(floor) ^  says(might(sleep(Joe)))), Jill)), speaker) - signals that Jill mentioned that "Joe might sleep" on the floor (say, a stage).

    2. exclaim(pres( say(might( on( the(floor), sleep(Joe))), Jill )), speaker) - signals that Jill mentioned that "Joe might sleep on the floor"

    Making a slight adjustment to the rules can help attain the second semantic representation
    1 VP[=1 sem="%subj 1(subj) ^ 2(subj)"]  VP PP
    ```
14) the perplexed president eat -ed a pickle . [seems correct]
15) Papa is perplexed . [seems correct]
16) Papa is chief of staff . [seems correct]
17) Papa want -ed a sandwich . [Seems correct]
18) Papa want -ed to eat a sandwich . [seems correct]
19) Papa want -ed Joe to eat a pickle . [seems correct]
20) Papa want -ed a pickle to eat Joe . [seems correct]
21) Papa would have eat -ed his sandwich -s . [seems correct]
**22) every sandwich was go -ing to have been delicious .** [seems incorrect]
```
The NP semantics seems wrong since every sandwich [all(sandwich)] is a generalized quantifier and must apply to a predicate of entities, not be an argument to delicious.

The appropriate parse with semantics might be:
assert(past(all(sandwich)(%s prog( will( perf(delicious(s)))))), speaker)
```
```

23) the fine and blue woman and every man must have eat -ed two sandwich -s and sleep -ed on the floor . [seems incorrect]
**sem:** assert( must( on( the(floor), perf( eat( two(sandwich) , and(the(%x woman(x) ^ (fine(x)^blue(x))),all(man))) ^ sleep(and(the(%x woman(x) ^ (fine(x)^blue(x))),all(man)))))), speaker)

```
The reason the parse seems incorrect since `on(the(floor))` seems to be attached to both eat -ed .. and sleep -ed ...

The correct parse instead should have been closer the following:
assert((the(%x woman(x) ^ fine(x) ^ blue(x)), all(man) )(%subj must( perf( eat( two(sandwich),  subj) ) ^ perf( on( the(floor), sleep(subj) ) ))), speaker)
```

**Question 5**
In english.gra 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.gra to admit three values for the num attribute: 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). 5 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.gra. You'll notice that mass determiners are always plural determiners as well (all caviar → all bonbons) but not vice-versa (two bonbons /→ *two caviar).2 Try to handle these inelegant facts elegantly, using as small and simple a system of 11 rules as you can under the circumstances. Submit your modified english.gra. Run it on a few sentences about caviar—both grammatical and ungrammatical ones—and 12 report what happened.

**Answer:**

- Mass nouns take plural Determiner forms and Singular verb forms. For example: "All caviar is delicious ."
- In order to encode these rules, we can approach it in two ways:
    1. Add a new "**num = mass"** and add the appropriate Determiners with the "**num = mass**" attribute and tack singular V and VP forms. The mass nouns are not countable and hence Plural determiners like Two are invalid. This might not be the most elegant method as there are duplicated rules. [Submitted on gradescope]
    2. Other way to handle this problem would be to add more attributes to rules to avoid rule duplication. The new attributes added are
        a. verb_agreement=sg|pl on V and VP.
        b. num=mass alongside sing/pl
        c. can_count=Y|N on N (nouns)
        d. Determiner selection constraints with "req=sing|nonsing" and "req_count=Y"
        e. NP construction split by number and countability
            i. Count singular: NP[num=sing] → Det[req=sing req_count=Y] N[num=sing can_count=Y]
            ii. Count plural: NP[num=pl] → Det[req=nonsing req_count=Y] N[num=pl can_count=Y]

iii. Mass with Det: NP[num=mass] → Det[req=nonsing] N[num=mass can_count=N req_count=N]
iv. Bare mass NP: NP[num=mass] → N[num=mass can_count=N req_count=N] with sem=the(1).

f. S-rule expansion by number + agreement
i. NP[num=pl|sing|mass] pairing with VP[verb_agreement=pl|sg]

g. refactored Determiner
i. a/every/each marked **req=sing req_count=Y**.
ii. all/some/the (nonsingular) marked **req=nonsing** (compatible with plural and mass).
iii. Possessives (his/her/its/their) as **req=nonsing** forms.
iv. two marked **req=nonsing req_count=Y** (plural count only).

I tried the above grammar modifications with the following test sentences that exhibited expected behavior:

```
```

**Grammatical**

- all caviar is delicious .
- some caviar is delicious.
- the caviar is fine .
- all caviar is nice .
- all rice is delicious .
- All sand is fine .
- she eats some rice .

**Non Grammatical**

- every caviar is delicious .
- the caviar are perplexed .
- two caviar are fine .
- two sand -s is delicious .
- two rice is delicious .

```
```

**Question 6**

**english.gra** doesn't attempt any real semantics for determiners. In particular, quantifiers like "every" are left as atomic elements with no internal semantics. english-fullquant.gra fixes this, reorganizing the grammar along the lines you explored in questions 3g–3i. 3 At the end of the semantics lectures, we also handled "every nation" in this style—you could review those slides. Try parseattrs english-fullquant.gra english.sen to see the new form of the output. Study english-fullquant.gra to see how it's done; just look at the changes, which are marked with ***. As before, you can use either your parse.py or parse2.py from Homework 4, or if you're not sure that those are working correctly, you can use the Homework 1 parser. Note that the parses under the english-fullquant grammar will be slightly different from the parses under the english grammar.

(a) The new grammar gives pretty complicated semantic attributes to two and to 13 singular and plural the. Justify the attributes it uses (i.e., explain what those lambda-terms mean). The ! symbol means "not."

‘’’

**Answer:**

NP isn't restricted to a single entity anymore with the new constructs. The new grammar loosely couples and treats **NP** as a predicate with more functionality and to not hard code it to a specific proper noun or entity. Doing it this way, gives the possibility to different kinds of subjects or objects **NP** can take.

- **Two**

**1 Det[=1 num=pl  sem="%dom %pred E%first E%second [first!=second ^ dom(first)^dom(second)] ^ pred(first) ^ pred(second)"] two**

- There exists two elements first and second, where first and second elements of the same noun's domain and take on the same predicate but are distinct and not one and the same.
- However, this can also mean "At least two".
- An attempt of mentioning exactly two could be

**"A%z [dom(x) ^ pred(x) ^ dom(y) ^ pred(y) ==> (x=first ^ y=second and x!=y)]"**

- **'the' in singular form**

**1 Det[=1 num=sing sem="%dom %pred E%t [dom(t) ^ A%u [dom(u) ^ u!=t ==> salience(u) < salience(t)] ] ^ pred(t)"] the**

- There exists an element `t` that takes the singular noun's domain which takes a verb predicate which is in agreement with the singular noun form.
- The contextual saliency/prominence of the element t is more than all the other elements denoted by u.
- For example, in a set of mugs, the prominence of one specific mug is higher than the other in context. We call it `The mug`. This represents this phenomenon.

- **'the' in plural form**

**1 Det[=1 num=pl sem="%dom %pred E%T [subset(T,dom) ^ |T|>one ^ A%U [subset(U,dom) ^ |U|>one ^ U!=T ==> salience(U) < salience(T)] ] ^ pred(T)"] the**

- There exists a subset of elements from a plural noun's domain, where the cardinality of subset is greater than 1, where the salience/prominance of the subset is higher than that of the other subsets of the plural nouns in context.
- This subset is modified by the predicate which is in agreement with the plural noun form.
- For example, from a universal set of countries, the prominence of a subset of countries is higher than the other countries in context. We call it `the countries`.

**Other interesting `Det` semantics:**

- **"a"**

**1 Det[=1 num=sing sem="%dom %pred E%b [dom(b)] ^ pred(b)"]  a**

- There exists an element b which takes the singular noun's domain and is modified by predicate that's attached by VP later.
- Example: past(E%x [Woman(x) ^ marry(x, Jack)])
- For example: Jack marries a woman

- **“all”**

**1 Det[=1 num=pl sem="%dom %pred A%c [dom(c)] ==> pred(c)"] all**

- All elements c take on the plural noun's domain and is modified by the predicate of appropriate num attribute.
- For example: A%c (dog(c) => loyal(c)) | under num=pl
- “All dogs are loyal”

- **“every”**

**1 Det[=1 num=sing sem="%dom %pred A%d [dom(d)] ==> pred(d)"] every**

- All elements d take on the singular noun's domain and is modified by the predicate of appropriate num attribute.
- Example: A%c (dog(c) => loyal(c)) **under num=pl**
- Every dog is loyal

- **“each”**

**1 Det[=1 num=sing sem="%dom %pred A%i [dom(i)] ==> pred(i)"] each**

- Similar to every. All elements “i” take on the singular noun's domain and are modified by the predicate of appropriate num attribute.
- Example: **E%p [person(p) ^ unique(p)]**
- For example: Each person is unique .

**Q6 (b)** The semantics of one rule in the new grammar has been left as ???. It affects 14 the sentence Papa want -ed Joe 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.)

**Answer:**

- **Old `english.gra`:**
  1 VP[=1 tense=1 sem="1(2,3)"] V[arg=npvpinf] NP VP[tense=inf] # "want him to eat"
- **Modified `english-fullquant.gra`**
  Replace **???** with %subj 2(%obj 1(obj,3)(subj))

**Modified Rule:** 1 VP[=1 tense=1 sem="%subj 2(%obj 1(obj,3)(subj))"] V[arg=npvpinf] NP VP[tense=inf]