Natural Language Processing HW5

Yunhao Yang

Problem 2

(a) Simplify $(\lambda x\, x * x)(3)$
Answer: $3 * 3$

(b) Simplify $(\lambda x\, x * x)(y + y)$
Answer: $(y + y) * (y + y)$

(c) Simplify $(\lambda x\, x * x)y + y$
Answer: $y * y + y$
Note: This is parsed as $[(\lambda x\, x * x)y] + y$, not $(\lambda x\, x * x)(y + y)$.

(d) Simplify $(\lambda a\, a)(\lambda b\, f(b))$
Answer: $\lambda b\, f(b)$

(e) Simplify $(\lambda a\, 3)(\lambda b\, f(b))$
Answer: 3

(f) Simplify $(\lambda x\, \text{green}(x))(y)$
Answer: $\text{green}(y)$

(g) Simplify $(\lambda x\, \lambda y\, \text{ate}(x, y))(\text{lemur}, \text{leopard})$
Answer: $\text{ate}(\text{lemur}, \text{leopard})$

(h) Simplify $(\lambda x\, \lambda y\, \text{ate}(x, y))(\text{lemur})$
Answer: $\lambda y\, \text{ate}(\text{lemur}, y)$

(i) Apply the previous answer to "leopard": simplify $(\lambda x\, \lambda y\, \text{ate}(x, y))(\text{lemur})(\text{leopard})$
Answer: $\text{ate}(\text{lemur}, \text{leopard})$

(j) Simplify $(\lambda x\, f(x, y))(a)(b)(c(z))$
Answer: $f(a, y)(b)(c(z))$

(k) Simplify $(\lambda x\, f(x, y))(a, b, c(z))$
Answer: $f(a, y)(b, c(z))$

(l) Simplify $(\lambda f\, f(x))g$
Answer: $g(x)$

(m) Simplify $(\lambda f\, f(f(f(x))))g$

Answer: $g(g(g(x)))$

(n) Simplify $(\lambda f\, f(f(f(x))))(\lambda t\, a(c(t)))$
Answer: $a(c(a(c(a(c(x))))))$

(o) Simplify $(\lambda f\, f(f(f(x))))(\lambda t\, t * t)$
Answer: $((x * x) * (x * x)) * ((x * x) * (x * x))$

(p) Simplify $(\lambda f\, f(f(f(x))))(\lambda t\, a(b, c[t], d))$
Answer: $a(b, c[a(b, c[a(b, c[x], d)], d)], d)$

Problem 3

(a)
i.

$$f = \lambda x \text{ loves}(\text{Mary}, x)$$

ii.

$$f = \text{loves}(\text{Mary})$$

(b) The VP "loves Mary" has semantics λx loves(Mary, x).

(c) i. What is f?
Answer: $f = \lambda y\, (\forall x\, \text{woman}(x) \Rightarrow \text{loves}(x, y))$
ii. Translate f and $f(\text{John})$ into English.
Answer:

- $f(\text{John})$: “Every woman loves John” or “All women love John”
- f: “the property of being loved by every woman” or “is loved by every woman”

(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”?
Answer: $f = \lambda g\, \lambda y\, \text{Obviously}(g(y))$

- “loves Mary” has semantics: $\lambda x\, \text{loves}(\text{Mary}, x)$
- “obviously” has semantics: $f = \lambda g\, \lambda y\, \text{Obviously}(g(y))$
- “obviously loves Mary” has semantics: $f(\lambda x\, \text{loves}(\text{Mary}, x)) = \lambda y\, \text{Obviously}(\text{loves}(\text{Mary}, y))$
- Apply to Sue: $\text{Obviously}(\text{loves}(\text{Mary}, \text{Sue}))$

Pop/push trick: Apply g to variable y (pop), add Obviously around result, reabstract y (push).
(e) Suppose $f(\text{Mary})(\text{John}) = (\lambda e\, \text{act}(e, \text{loving}), \text{lovee}(e, \text{Mary}), \text{lover}(e, \text{John}))$. What is f?
Answer: $f = \lambda x\, \lambda y\, \lambda e\, \text{act}(e, \text{loving}), \text{lovee}(e, x), \text{lover}(e, y)$

(f) Keep f as in the previous problem. Suppose $g(f(\text{Mary}))(\text{John}) = (\lambda e\, \text{act}(e, \text{loving}), \text{lovee}(e, \text{Mary}), \text{lover}(e, \text{John}), \text{manner}(e, \text{passionate}))$. What is g?
Answer:

$$g = \lambda h\, \lambda z\, \lambda e\, h(z)(e), \text{manner}(e, \text{passionate})$$

g takes a VP semantics h (like $f(\text{Mary})$ = "love Mary"), applies it to argument z, then to event e, adds the manner predicate, and abstracts over e. This is the pop/push trick for adverbs.

(g) Suppose $f(\lambda x\, \text{loves}(\text{Mary}, x)) = (\forall y\, \text{woman}(y) \Rightarrow \text{loves}(\text{Mary}, y))$.
i. What is f?
Answer:

$$f = \lambda P\, (\forall y\, \text{woman}(y) \Rightarrow P(y))$$

ii. Translate $f(\lambda x\, \text{loves}(\text{Mary}, x))$, $(\lambda x\, \text{loves}(\text{Mary}, x))$, and f into English.
Answer:

- $f(\lambda x\, \text{loves}(\text{Mary}, x))$: "Every woman loves Mary"
- $\lambda x\, \text{loves}(\text{Mary}, x)$: "loves Mary" (VP)
- f: "every woman..." (quantifier that takes a property P and says every woman has it)

(h) Let f be your answer from question 3(g)i. Suppose $g(\text{woman}) = f$.
i. What is g as a lambda term?
Answer:

$$g = \lambda Q\, \lambda P\, (\forall y\, Q(y) \Rightarrow P(y))$$

Verification: $g(\text{woman}) = \lambda P\, (\forall y\, \text{woman}(y) \Rightarrow P(y)) = f$
ii. What English word does it represent?
Answer: g represents the word **"every"**.
g takes a noun (like woman) and returns a quantifier function. For example:

- $g(\text{woman})$ = "every woman" (a quantifier phrase)
- $g(\text{woman})(\lambda x\, \text{loves}(\text{Mary}, x))$ = "every woman loves Mary"

(i) Suppose $f(\lambda x\, \text{loves}(\text{Mary}, x)) = \text{loves}(\text{Mary}, \text{Papa})$.
i. What is f as a lambda term?
Answer:

$$f = \lambda P\, P(\text{Papa})$$

f takes a property P and applies it to Papa.
ii. Why would one want to give (NP Papa) the semantics sem=f (rather than just sem=Papa)?
Answer: **Type uniformity**. To use a single consistent rule for all sentences.
Without this approach:

- Quantified NP: "every woman" = $\lambda P\,(\forall y\,\text{woman}(y) \Rightarrow P(y))$ (a function)
- Proper noun: "Papa" = Papa (a constant)
- Need different composition rules for different NP types

With this approach:

- Quantified NP: "every woman" = $\lambda P\,(\forall y\,\text{woman}(y) \Rightarrow P(y))$
- Proper noun: "Papa" = $\lambda P\,P(\text{Papa})$
- Both have type $(e \to t) \to t$, same composition rule: S = NP(VP)

This way, both "Every woman loves Mary" and "Papa loves Mary" use the identical semantic composition pattern.

Problem 4

Sentences with inappropriate semantics:

1. "Papa ate every bonbon with a spoon."
Semantics: `past(eat(all(%x bonbon(x) ^ with(some(spoon),x)),Papa))`
Problem: This means "Papa ate every bonbon that has a spoon," but the intended meaning is "Papa used a spoon to eat every bonbon." The PP "with a spoon" attached to the NP "every bonbon" instead of the VP "ate." A different parse attaching the PP to the VP would give the correct instrumental reading.

2. "Papa wanted a pickle to eat Joe."
Semantics: `past(want(eat(Joe,some(pickle)), Papa))`
Problem: This means "Papa wanted a pickle that would eat Joe," which is absurd. The intended meaning is "Papa wanted Joe to eat a pickle." The parser made "a pickle" the subject of "to eat Joe" instead of parsing it as "Papa wanted [Joe to eat a pickle]." A different parse with the correct control structure would be better.

Question 5

I modified `english.gra` to support mass nouns (num=mass). The key changes were:

1. Modified S rules: Expanded the S rules to handle three num values explicitly. Mass nouns use singular verbs:

```
S → NP[num=sing] VP[num=sing tense=pres]
S → NP[num=pl] VP[num=pl tense=pres]
S → NP[num=mass] VP[num=sing tense=pres]
```

(Similar for past and modal tenses)

2. Added caviar as a mass noun

```
N[num=mass] → caviar
```

3. Added mass determiners Only *all*, *some*, and *the* can modify mass nouns:

```
Det[num=mass sem=all] → all
Det[num=mass sem=some] → some
Det[num=mass sem=the] → the
```

I tested the modified grammar with the following sentences:

Grammatical sentences (parsed successfully):

1. "all caviar is delicious ." → `pres(delicious(all(caviar)))`
2. "some caviar is delicious ." → `pres(delicious(some(caviar)))`
3. "the caviar is delicious ." → `pres(delicious(the(caviar)))`
4. "Papa eat -ed some caviar ." → `past(eat(some(caviar),Papa))`
5. "Papa eat -ed the caviar ." → `past(eat(the(caviar),Papa))`

All grammatical sentences parsed correctly. The mass noun *caviar* correctly agrees with singular verbs (*is*, not *are*).

Ungrammatical sentences (correctly rejected):

6. "two caviar is delicious ." → `No consistent way to assign attributes!`
7. "a caviar is delicious ." → `No consistent way to assign attributes!`
8. "every caviar is delicious ." → `No consistent way to assign attributes!`
9. "Papa eat -ed two caviar ." → `No consistent way to assign attributes!`
10. "Papa eat -ed a caviar ." → `No consistent way to assign attributes!`

Problem 6

(a) Semantics of `two`:

```
%dom %pred E%first E%second
     [first!=second ^ dom(first) ^ dom(second)]
    ^ pred(first) ^ pred(second)
```

This means: there exist two distinct individuals that both belong to the domain and both satisfy the predicate. The `!=` operator ensures we're talking about two different entities, not counting the same one twice.

For "Papa ate two sandwiches", this gives us $\exists$first $\exists$second [first $\neq$ second $\land$ sandwich(first) $\land$ sandwich(second) $\land$ ate(first, Papa) $\land$ ate(second, Papa)], which correctly says Papa ate two different sandwiches.

Semantics of singular `the`:

```
%dom %pred E%t [dom(t) ^ A%u [dom(u) ^ u!=t
    ==> salience(u) < salience(t)]] ^ pred(t)
```

This means: there's an entity `t` in the domain that's more salient than any other entity in the domain, and `t` satisfies the predicate. Salience captures which object is most prominent or contextually relevant.

For "the sandwich is delicious", we get $\exists t$ [sandwich(t) $\wedge$ $\forall u$ [sandwich(u) $\wedge$ $u \neq t \Rightarrow$ salience(u) $<$ salience(t)] $\wedge$ delicious(t)]. This captures the definiteness of "the"—both speaker and hearer can identify the specific sandwich being discussed because it's uniquely salient.

Semantics of plural `the`:

```
%dom %pred E%T [subset(T,dom) ^ |T|>one ^
    A%U [subset(U,dom) ^ |U|>one ^ U!=T
        ==> salience(U) < salience(T)]] ^ pred(T)
```

Similar idea but for sets. There's a set `T` (containing multiple elements) that's more salient than other sets, and the predicate applies to this set. For "the sandwiches are delicious", the set referred to is contextually identifiable (like all the sandwiches on a particular plate). The predicate can either distribute over elements (each sandwich is delicious) or apply collectively (the set costs $30).

(b) The rule is: `VP[sem="???"] → V[arg=npvpinf] NP VP[tense=inf]`

Used for sentences like "Papa wanted Joe to eat a pickle."

Answer: `%subj 2(%obj 1(obj, 3, subj))`

How this works: The verb `want` has three arguments: object (who should do it), predicate (what they should do), and subject (who wants it). We use the pop/push technique to handle the generalized quantifier NP. First pop `obj` to create λobj want(3(obj), subj), then apply the NP to bind `obj` to Joe, giving want(3(Joe), subj) = want(eat(some(pickle), Joe), subj). Finally push `subj` back to get λsubj want(eat(some(pickle), Joe), subj). When applied to Papa, this correctly produces want(eat(some(pickle), Joe), Papa), making Papa the wanter and Joe the one who should eat. This follows the same pop/push pattern used for transitive verbs but accommodates the extra VP argument.