601.465/665 — Natural Language Processing
Homework 5: Semantics

Prof. Jason Eisner — Fall 2025
Due date: Monday 27 October, 11 pm

In this homework, you will run your output parses from Homework 4 through a post-processing script that computes the attributes (i.e., nonterminal features) of each constituent.

You will be asked to understand and tweak the grammar that assigns the attributes. No programming is involved, just thinking. Our main focus is on the `sem` attribute, whose value is a semantic interpretation of the constituent, such as a λ-term.

Homework goals: This short homework is the main homework that deals with formal representations of meaning. After doing it, you should be comfortable

- manipulating λ-calculus expressions formally
- building them up compositionally using semantic attachments to syntactic rules
- considering whether they appropriately capture the meaning of a phrase

Collaboration: You may discuss problems 1–2 with anyone; there is nothing to hand in for those. However, you should do problems 3–6 alone, in order to ensure that you are personally fluent with λ-calculus. Of course, feel free to ask the course staff for help.

Reading: Read the handout attached to the end of this homework!

Materials: All the new files you need can be found in http://cs.jhu.edu/~jason/465/hw-sem. The scripts require you to have installed Perl (and preferably rlwrap as well). If you don't want to install anything, these files are also available on the `ugrad` machines, under `/usr/local/data/cs465/hw-sem`, and the scripts should run there with no trouble. Read Figure 1 for a guide to the old and new files.

1. Your first job is to understand the notation for adding attributes to a grammar. A grammar with attributes is a `.gra` file. The corresponding `.gr` file can be produced by using `delattrs`, which strips the attributes and comments, as well as normalizing the probabilities for you. You have been given some simple `.gra` files that demonstrate the notation.

   (a) Read the file `arith.gra` carefully and examine the output of the following commands, especially the part of the output that is *not* indented:

   ```
   parse.py arith.gr arith.sen > arith.par
   buildattrs arith.gra arith.par
   ```

   Parsing speed will not be a big issue in Homework 5, so you can use your `parse.py` from Homework 4 rather than `parse2.py`. If you weren't even able to get `parse.py` working correctly, then you could instead use the `parse` program supplied with Homework 1 (which has a slightly different command-line syntax—use `parse -h` to see the help text).

`*.gr`	simple grammar with rule *weights*, no attributes, no comments
`*.sen`	collection of sample sentences (one per line)
`*.par`	collection of sample parses
`checkvocab`	script to detect words in `.sen` that are missing from the grammar `.gr`
`parse.py`	your program from Homework 4 that converts `.sen` $\xrightarrow{\texttt{.gr}}$ `.par`
`prettyprint`	script to reformat `.par` more readably
`*.gra`	full grammar with rule *frequencies*, attributes, comments
`delattrs`	script to convert `.gra` $\longrightarrow$ `.gr`
`buildattrs`	script to convert `.par` $\xrightarrow{\texttt{.gra}}$ an attribute assignment
`parseattrs`	simple script to convert `.sen` $\xrightarrow{\texttt{.gra}}$ an attribute assignment (calls `checkvocab`, `parse`, and `buildattrs`)
`simplify`	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 Homework 4.

The output of `buildattrs` for each parse is an indented trace, showing how the attributes 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 attributes 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 attributes arose from those of smaller constituents.

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

```
buildattrs arith-infix.gra arith.par
```

(c) Finally, study `arith-typed.gra` and try

```
parse.py arith.gr arith-typed.sen > arith-typed.par
buildattrs arith-typed.gra arith-typed.par
```

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

```
parseattrs arith-typed.gra arith-typed.sen
```

> If you are using the `parse` command from Homework 1, then you'll have to edit `parseattrs` to use that command, with its command-line syntax.

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 (described in reading section B), which will simplify any lambda-expression you type in. But try to come up with each answer on your own first ...

(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?)

3. From here to the end of the homework, you should work by yourself and hand in your answers as a PDF submission to Gradescope. Use 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.[1]

   (These "division" problems are related to the end of the semantics lecture. We wanted a particular meaning for "Every nation wants Joe to love Jill," 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 $f(John) = loves(Mary, John)$. What is f,

   i. written in the form $\lambda x\ \cdots$?

   ii. 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, $loves(Mary, John)$ 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?

(c) Suppose $f(John) = (\forall x\ woman(x) \Rightarrow loves(x, John))$.

i. What is f?

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

(d) Suppose $f(\lambda x\ loves(Mary, x)) = (\lambda x\ Obviously(loves(Mary, x)))$. 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.

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

(f) Keep f as in the previous problem. Suppose $g(f(Mary))(John) = (\lambda e\, act(e, loving), lovee(e, Mary), lover(e, John), manner(e, passionate))$. 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.

(g) Suppose $f(\lambda x\ loves(Mary, x)) = (\forall y\ woman(y) \Rightarrow loves(Mary, y))$.

i. What is f?

ii. Translate $f(\lambda x\ loves(Mary, x))$, $(\lambda x\ loves(Mary, x))$, and f into English.

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

i. What is g as a lambda term?

ii. What English word does it represent?

Hint: Substituting $g(woman)$ for f in question 3g yields $g(woman)(\lambda x\, loves(Mary, x)) = (\forall y\ woman(y) \Rightarrow loves(Mary, y))$. 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?

(i) Suppose $f(\lambda x\ loves(Mary, x)) = loves(Mary, Papa)$.

i. What is f as a lambda term?

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 $(\forall y\ woman(y) \Rightarrow loves(Mary, y))$ into English, and remember that we'd like to treat similar phenomena in a consistent way.)

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`.

[1] Other 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.

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 Understand$^{\text{TM}}$ 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.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).

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 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 report what happened.

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 singular and plural *the*. Justify the attributes 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 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.)

[2] It would be nice to capture this asymmetric generalization with a rule like `N[num=pl]` → `N[num=mass]`, which says that mass determiners can always be used where plural determiners are called for. One could similarly write `NP[num=sing]` → `NP[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` attributes. So they won't work with our pipeline system, which parses *before* it looks at the attributes.

[3] This approach (due to Montague 1973) is called the "Proper Theory of Quantification" because it says that proper nouns have the same semantic type as NPs containing quantifiers.

601.465/665 — Natural Language Processing
Reading for Homework 5: Semantics

Prof. Jason Eisner — Fall 2025

We don't have a required textbook for this course. Instead, handouts like this one are the main readings. This handout accompanies homework 5, which refers to it.

A Pipelined NLP versus Joint NLP

The split into Homework 4 and Homework 5 is a bit artificial. After all, attributes are really *part* of the context-free grammar, so your parser in Homework 4 should arguably have considered them while parsing in the first place. On the other hand, ignoring the attributes 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 program enriches the input with some new kind of annotation before passing it on to the next program for further processing. For example:

1. transcribe a speech input into a sequence of words
2. divide those words into morphemes
3. disambiguate those morphemes with part of speech tags
4. pick the best parse that is consistent with those part of speech tags
5. compute 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, using feedback or joint optimization.

Specifically, what is lost by our "tree first, attributes second" pipeline in Homeworks 4–5? In principle, it could have improved accuracy if the parser had computed constituents' syntactic attributes *during* parsing, as humans probably do. Then the rule probabilities could have depended on the nonterminal attributes. In particular, attribute 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).

You'll be interested to know that the parsing community has swung back and forth on whether that is a good use of runtime. Attributes 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 attributes, as in the Penn Treebank (see `wallstreet.gr`). That's because syntactic attributes such as **agreement** rarely help for

choosing among the *most probable* parses of a sentence.[1] As the following partly-obscured sentence shows, you can often guess the right parse without knowing whether the nouns and verbs are singular or plural, so maybe a parser can too:

`The girl[redacted] think[redacted] that the flower[redacted] smell[redacted] nice.`

Thus, many modern probabilistic English parsers compute little more than the **head** attribute while parsing (since conditioning the probabilities on the head attribute does help parsing accuracy). Only in the past several years have a few researchers (e.g., the Berkeley parser) shown benefit by picking the "right" nonterminal attributes.[2]

The more detailed **semantic** attributes that you will consider in Homework 5 could theoretically help parsing as well—but only in a system that can reason about the semantics and relate it to knowledge about the world in order to decide whether a constituent is plausible.

B The `simplify` script

B.1 Running the script

Run the script as `./simplify` with *no arguments*. Then type some λ-calculus expressions. It will simplify each expression that you enter. If you have `rlwrap` installed on your system, then you can use the up-arrow to recall previous expressions, and edit them.

To get started, look inside the script itself. It has comments with some sample expressions that you can type.

B.2 What does the script do?

`simplify` just keeps on simplifying the expression until it can't be simplified any more. This is *always* a matter of replacing a function application with the result of applying that function.

That is, `simplify` looks for sub-expressions where a λ-term is applied to an argument. Whenever it finds one, it replaces the sub-expression with its result—basically, it calls the function. If there are two such sub-expressions, it can replace either one first—that doesn't affect the final answer. It keeps doing this until it can't anymore.

Example: The function application $f(plus(2,3))$ can't be simplified any more.

Example: However, $(\lambda x\ times(x,x))(plus(2,3))$ can be simplified, because the function is expressed as a λ-term. The result is $times(plus(2,3),plus(2,3))$, which is just the function's **body** $times(x,x)$ but with the function's **parameter** x replaced with the **argument** $plus(2,3)$.

This kind of simplification step is formally known as β-reduction. To avoid confusion between two variables of the same name, it might need to rename variables, which is technically known as α-conversion.

[1] Of course, a grammar with attributes is still needed to *define* grammaticality and to *generate* grammatical sentences with `randsent`.

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

B.3 Notation used by the script

- The script interprets `f(x,y)` as an abbreviation for `f(x)(y)`.
- You must type `^` rather than `,` to indicate conjunction. This is because the script assumes that comma is *only* used for constructions like `f(x,y)`.
- The `%` character is used to represent λ.[3] So you can write `(%x x*x)(3)`, which simplifies to `3*3`.
- $\forall x\ woman(x) \Rightarrow loves(x, John)$ is written as `A%x woman(x) => loves(x,John)`.[4]

  Notating the quantifier $\forall$ as `A%` (or anything ending in `%`) tells `simplify` that the following x is a variable name, not a constant. Variable names are arbitrary: that expression has exactly the same meaning as `A%foobar woman(foobar) => loves(foobar,John)`.

  In the same way, you can write $\exists$ as `E%`.

More detailed documentation can be found at the top of the file `LambdaTerm.pm`.

C Tips, tricks, and examples

C.1 Functions of functions

The argument to a function application can be another function. Don't be confused!—the rules are the same.

Example: The function application $(\lambda f\ f(2,3))(plus)$ can be simplified, because the function is expressed as a λ-term. The result is $plus(2,3)$, which is just the function's body $f(x,x)$ with the function's parameter f replaced with the argument $plus$.

More interesting example: The function application

$$(\lambda f\ f(x+1, x+2))(\lambda y\ \lambda z\ plus(sin(y), cos(z)))$$

can be simplified, because the function is expressed as a λ-term. The result is the function's body, $f(x+1, x+2)$, with the function's parameter f replaced with the argument $\lambda y\ \lambda z\ plus(sin(y), cos(z))$. This gives

$$(\lambda y\ \lambda z\ plus(sin(y), cos(z)))(x+1, x+2)$$

Do you see how the parameter f was actually being applied as a function in the subexpression $f(x+1, x+2)$, and we replaced it with a λ-term saying what specific function to apply? But simplify isn't done yet ... it can now call that function!

Recall that the above result is shorthand for

$$((\lambda y\ \lambda z\ plus(sin(y), cos(z)))(x+1))(x+2) \quad [*]$$

But that contains another function application we can simplify—namely $(\lambda y\ \lambda z\, plus(sin(y), cos(z)))(x{+}1)$. The result is the function's body, $\lambda z\, plus(sin(y), cos(z))$, with the function's parameter y replaced with the argument $x+1$. This gives

$$\lambda z\ plus(sin(x+1), cos(z))$$

[3] Perhaps we should have used backslash, which is often used because it looks more like λ.

[4] This 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.

Using this result in the original expression marked with [*], we get

$$(\lambda z\ plus(sin(x+1), cos(z)))(x+2)$$

But this is yet another function application we can simplify! The result is the function's body $plus(sin(x+1), cos(z))$ with the function's parameter z replaced with the argument $x+2$. So our final answer is

$$plus(sin(x+1), cos(x+2))$$

Hopefully that's not too surprising as a simplification of the original expression $(\lambda f\ f(x+1, x+2))(\lambda y\ \lambda z\ plus(sin(y), cos(z)))$.

C.2 Functions are opaque

A λ-term is an expression that denotes a particular function. Just as $3+5$ is an expression that denotes a number.

The API for functions is very simple—the only thing you can do with a function is to apply it to an argument. You *can't* get at the function's source code, i.e., the expression that was used to describe it.

Analogy: In the same way, if you write $f(3+5)$ in C, the f function only sees the number 8. f can examine 8 through the usual API for numbers, and thus determine that it is positive and composite and a power of 2. But it can't get at the original expression and say "Oh, this 8 is the result of an addition."

In particular, $f(3+5)$ can't systematically give the result $9+5$, where it squares the first argument to the addition. There is no way for $f(3+5)$ to give a different result from $f(2*4)$. All f sees is the 8.

C.3 How to manipulate functions even though they're opaque

Suppose $myfunc = \lambda x\ f(x, x)$ and you want to write an expression in terms of $myfunc$ that gives $\lambda x\ g(f(x, x))$.

Note that the goal here is to "slip g underneath the outer λx." But as we just discussed, your expression has no ability to inspect or edit the text of $myfunc$. It just sees a function, not the expression $\lambda x\ f(x, x)$!

- The *only* way to manipulate $myfunc$ is to apply it to something. What should we apply it to? Oh, anything—let's say y, where this is just a new variable that could stand for anything.

  So let's write $myfunc(y)$, which equals $f(y, y)$ because of the definition of $myfunc$. Well, that's making progress: it got rid of the outer λx.

- We can now add the g in the desired place by writing $g(myfunc(y))$, which equals $g(f(y, y))$.

  That's still sort of an ill-formed expression that doesn't yet mean anything: y is a variable, not a constant like 4 or π or e. If you tried to evaluate $g(f(y, y))$ in the real world, you'd get something like "Undefined variable error."

- But if we abstract away the y, we do get a meaningful expression again: $\lambda y\ g(myfunc(y))$. This is a function, which equals $\lambda y\ g(f(y, y))$ because of the definition of $myfunc$.

  In fact that's the same function we wanted, since $\lambda y g(f(y, y))$ denotes the same function as $\lambda x\ g(f(x, x))$.

- So $\lambda y\ g(myfunc(y))$ is the expression you wanted in terms of $myfunc$. But if you wanted to write a similar expression in terms of $theirfunc$, would you have to do the same thing all over again?

  Hopefully not. To make our solution general, let's abstract out the choice of $myfunc$ and just call it F.

  That is, $\lambda F\ \lambda y\ g(F(y))$ is a general function that you can apply to $myfunc = \lambda x\ f(x,x)$ to get the answer above (namely $\lambda y\ g(myfunc(y)) = \lambda y\ g(f(y,y))$), but which you can also apply to other functions like $theirfunc$ to transform them similarly.

D Other Resources

Alternatives to the `simplify` script: Linguists have developed nice graphical tools for their students to practice the λ-calculus. You could try the impressive and elegant Jupyter Lambda Notebook from Prof. Kyle Rawlins at Johns Hopkins, or the Lambda Calculator (which helps you through some exercises) from a team at U. Penn. If you try one of these, feel free to ask for help, and definitely let us know on Piazza how you like it!

Bret Victor draws lambda terms as families of colored alligators. During simplification, the grand alligator of one family eats the whole next alligator family. It dies from overeating, but its eggs hatch into copies of the eaten family. Check out his drawings at Alligator Eggs!