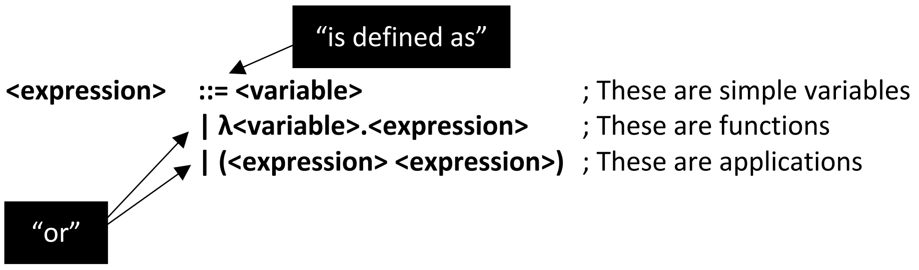
Lambda Applicator Lab

- Add in pre-parse, post-tokenize parens around lambdas
- Add in normal order

The target of this lab is to create a Lambda solver in Java. This first section contains some review of what that means.

The Lambda solver solves *expressions*. An expression can be made up of different, other *expressions*. There are three types of *expressions*:



<expression> **::= <variable>** ; These are simple variables
| λ<variable>.<expression> ; These are functions
| (<expression> <expression>) ; These are applications

Example: `((λy.(x y)) food)`

- The whole `((λy.(x y)) food)` is an expression — more specifically, an `Application`, because it has two parts: `(λy.(x y))` and `food`.
- `(λy.(x y))` is a `Function` because it starts with a `λ`.
- `(x y)` is an `Application` because, again, it contains two parts: `x` & `y`.
- Finally, `x`, `y`, & `food` are all `Variable`s because they are single variables.

In the next section, I have provided specifications (and hints), along with many, many test cases and their desired outputs. In the meantime, I suggest you re-familiarize yourself with Lambda Calculus. I would go back over the PowerPoint before proceeding any further.

- Additionally, one great resource you can use to see the applications at work is https://www.easycalculation.com/analytical/lambda-calculus.php. It walks through any calculations you put in, one step at a time, allowing you to watch each step taking place.

A note about Windows

This lab works in every flavor of *NIX, including Macs. However, native Windows consoles do not play nicely with extended Unicode characters such as λ. Windows does not pass such characters into console-based applications properly. As a result, most Windows-based IDEs (such as VS Code, IntelliJ, etc., etc.) will simply not work for this lab.

There are two easy workarounds. One option is Eclipse, which is a (genuinely good) IDE made for enterprise-level work, so it has its own built-in console that passes lambda inputs just fine. Another option is to use *nix based websites, such as replit or codingrooms to do your development.

Implementation Hints in Java

The most important thing I can say here is that you are not bound in *any way* by the suggestions that follow in this section. Many students with very successful projects have ignored all of this, taken just a bit of my starter code, and went on to have very rewarding experiences and excellent final products.

That said, if you feel like your OOP is a little rusty, or just aren't sure how to start, the suggestions that follow form a very idiomatic, native Java-esque approach to the problem. They are a fine starting place, and you are absolutely welcome to simply follow them as a guide if you wish.

Because we are working with `Objects` in Java, and we have an *is-a* relationship, this recommends either using inheritance or an interface, with `Expression` being either the `abstract` parent class or the implemented `Interface`. (Why would it be `abstract`?) I used an `Interface`, because I felt that very few actions were exactly the same from one type of Expression to the next, but you should not feel constrained by my decision.

Whether we use inheritance or interfaces, an `Expression` should be able to be a `Variable`, a `Function`, or an `Application`.

A Function *has-a* `Variable` and an `Expression`. (How do we work with *has-a* relationships in Java?) This sub expression can take any of the three forms mentioned above.

An `Application` contains two `Expressions` : a left `Expression` and a right `Expression`. Keep in mind that both of these `Expression`s can also take any of the three forms mentioned above.

A Variable has only its name, which can be accessed with `toString()`. The method `toString()` is useful for `Expression`s because it allows you to simply `System.out.print(myExpression)`.

There is another implementation choices you can make for Variables. I did not do the following, but you may wish to:

Have a variable store whether it is free, bound, or a parameter. If you do this, you may want to create three subclasses of `Variable`. `FreeVar` would have no `setName()` method, `BoundVar` would have `setName()`, and `Parameter` would also store a `List` of `BoundVar`s. One hint is to think about what useful functions `Parameter` could have.

Doing this will **not** ultimately make the code any simpler, but it will shift some of the complexity from *running* expressions (which you do in the second part of the lab) into *creating* the expressions (which you do in the first part of the lab). My suspicion is that the first part of the lab is a somewhat easier than the second part, so you may find that doing this early helps to reduce the complexity of the second part. However, I would re-emphasize that I did not do it, it is not necessary, and it will not result in a simpler lab overall. What it *might* do for you is to spread the difficulties out more evenly as you create your project.

Getting Started

Okay, done with your review? I am providing a *lot* of test cases here, along with explanations that I hope will help you understand what you need to actually accomplish. At the end, I have the tests again in an easily copyable/pastable format.

If your final submission passes all of the tests below, then you will be in the higher grading rubric.

I strongly recommend implementing the features in the order they are introduced below.

Creation Guide: Tests for some Test-Driven Development

I'm going to be giving you a series of tests to pass. Each test is meant to represent the next smallest piece that you can add to your code. One of the rules of TDD is to **always go back and try out the previous tests to make sure that nothing has broken as you work on the next step**. This process can be automated with tools like `JUnit`, though this project is small enough that you can just go back through the tests manually as you proceed. (Check the end of the document for a copy-paste-ready list of all of the tests in the implementation steps.)

That said, if you would like to experiment with `JUnit` on your own, this is an excellent lab to do it with, since I have broken the lab requirements down into tests for you!

CATEGORY: Console

1. Take as many blank lines as are entered, place a > at the beginning of each line, and exit when the user types exit. In this case, the user presses $\boxed{\text{enter} \hookleftarrow}$ 3 times, and then types $\boxed{\text{e}}$ $\boxed{\text{x}}$ $\boxed{\text{i}}$ $\boxed{\text{t}}$ and $\boxed{\text{enter} \hookleftarrow}$ again:
```
>
>
>
> exit
Goodbye!
```

2. Now we'll create the simplest `Expression`s, `Variable`s. It may look like we are echoing back the word put in, but we are instead parsing the argument passed in, and forming it as a `Variable`. The `toString()` of a `Variable` would return its name.
```
> a
a
> cat
cat
```

CATEGORY: Lexer

3. This would also be a good time to create your **lexer** and your **parser**. First, the **lexer**. A **lexer** is also called a **tokenizer**, because that's what it does: it isolates the *tokens* out some string. In fact, "tokenize(…)" might be a good method name to use in your Lexer class!

   You can remember what a lexer does because **"lex"** is the Greek root for "word", and we use this often in English. *Lex*ophiles love words! *Lex*icographers make dictionaries! A *lex*icon is a word list! And Lex Luthor talks too much! (This is his weakness. He talks instead of just beating Superman once and for all, and then misses his chance.)

   Let's look at a slightly crazy input to get a sense of what a lexer does:
```
 > (    (\bat  .bat flies)cat   );     comment
```

In order for this to be processed, we need to find the **tokens**, which are the smallest units with meaning. And that is exactly what the lexer does for us! A **lexer** converts an input into *the tokens that will need to be interpreted*. In this case, it would output this list:

`["(", "(", "\", "bat", ".", "bat", "flies", ")", "cat", ")"]`.

Here is another example:

```
> ((cu (a λboo.a boo) )) day
```

becomes

```
["(", "(", "cu", "(", "\", "boo", ".", "a", "boo", ")", ")", ")",
"day"]
```

Notice that longer variable names like "cu" and "boo" were not broken apart, and that no spaces are included in the list of tokens. Just as in languages like Java and Python, spaces are really instructions to the tokenizer.

A few more things to note:

- You eventually want some sort of list, perhaps an ArrayList<String>, of tokens.
- **No effort is made to interpret the symbols in any way.** That's not the lexer's job. It just finds and isolates the words (tokens) for us.
- Nothing after `;` makes it past the tokenizer. After all, comments are not meant to be interpreted. (How will you make sure that comments are excluded?)
- The strings in the output have no spaces around them. The meaningful parts have been fully isolated.
- There are no useless elements (like empty strings) in the final list.
- Comments are not meaningful symbols, so they do not make it into the lexer's final list.
- This should be O(n), as you can do this all in a single pass, going through the input string letter by letter. The decision points that you need to consider are what to do when you arrive at any of the 8 special characters in our programming language: `)`, `(`, ` `, `;`, `\`, `λ`, `.`, and `=`.

CATEGORY: Parsing

4. This is not required, but I'd recommend a **pre-parser** step – you can add a call to it into main, in the console, after the tokens are returned. The goal is to add parenthesis around all lambda functions. The reason for this is that, while an expression like `a b c d` has four top-level items, and an expression like `a (b c) d` has three, an expression like `a \b.c d` should only contain two, because lambdas automatically consume everything to their right.

   Thus, you would want that expression to become the list `[a, (, \, b, ., c, d, )]`. The only two other situations to be aware of is when a lambda is already surrounded by parenthesis (such as `a (\b.c d) e`), in which case you should make no modifications, and when your lambda is blocked at the end by parenthesis out of its scope, such as `(a \b.c d) e)`, which should become `(a (\b.c d)) e)`.

All of this will help your parser to make sense of where everything belongs, and will also make sure to isolate each Function so that you don't process it until you are really ready to.

5. Now for the **parser**. A **parser** takes the output of a lexer, and interprets the tokens to figure out its structure. The parser does **not** "run" the expression or perform any applications. Instead, it simply decides which Applications are contained within which Functions, where the variables go, etc. In other words, it creates a *tree* that represents the *structure* of whatever the user typed in.

   We call this output a **parse tree**, meaning the tree that results from parsing an input. In our case, a parser should take the `ArrayList<String>` from the `Lexer` and output an `Expression`.

   This should all be done in a `Parser` class. This is because your parser may start out very simply, but will get quite complex before it's finished, and you will want a place to put all of your helper methods together. Parsers are complex, and we will break it up into multiple sub-steps in this lab.

   Your first step, then, is to create the `Parser` class, and the `parse` method that returns an `Expression`. Then route everything (such as turning your lexer output into a variable, which you did earlier) through it.

6. Now make sure that you can get the prior steps to run, but routed properly through the `Lexer` and the `Parser` classes.

7. Since Variables are now being created by the parser, let's see if we can coax it to generate some `Application`s. We won't Beta-reduce them yet, just create them.

   Take a look at this output first! It may look like we are just inserting parenthesis, but we are creating trees. An `Application` is composed of a left `Expression` and a right `Expression`. What about the input string tells us that this should be an application and not simply a variable?

   ```
   > a b
   (a b)
   ```

   Then the **parser** would receive the list `["a", "b"]`, and create the following tree.

   ```
      ┌─── [APP]───┐
   [VAR: a]     [VAR: b]
   ```

   Where do the parenthesis come from in the output, then? From `toString()`. The `toString()` of an `Application` would return `"(" + left + " " + right + ")"`.

   Hint: **Are you still going back to make sure that all of the prior tests are still passing? If so, then you have already realized that you will need your `Parser` to figure out when you are looking at a `Variable` and when you are looking at an `Application`.**

8. Now, let's do order of operations for applications:

```
> a b c d
(((a b) c) d)
```

This should create the following tree:

```
        ┌─[APP]─┐
    ┌─[APP]─┐   d
┌─[APP]─┐   c
a       b
```

Please note that the input provided here has 4 *top level* items. That is vital information that you will need to create that tree. Make sure that you are finding that number somehow.

Hint: once you know that there are four top-level items, there are then two different ways that you can roll out the tree above. You can either process the tokens right-to-left or left-to-right. How would you go about creating the tree using either direction?

9. Make sure that you are determining what the top-level arguments are, and processing them recursively. For instance, in…

```
> a b c (d e)
(((a b) c) (d e))
> exit
Goodbye!
```

… `(d e)` is the right side of an application that has `((a b) c)` as its left. You can see that it goes exactly where `d` went in the prior challenge. Think carefully about how your parser should deal with parenthesis. Things are going to get more complicated at this point, and you may find that the ease you developed with recursion this year is quite helpful here.

The key here is to (again) determine that there are 4 top-level items in the input.

10. Make sure that extraneous *matching* parenthesis are stripped away. In the first case here, the parenthesis match, and should be recursively stripped away. In the second case, your parser can note that this is an `Expression`, and create something like this pseudo-code: `new Expression(parse(left), parse(right))`. In that case, it would be up to `parse(left)` to notice the matching parenthesis around `a`, and `parse(right)` to notice the matching parenthesis around `b`.

```
> (((a)))
a
> (((a))) (((b)))
(a b)
```

11. Here are our simplest `Function`s. A `Function` is composed of a `Variable` and an `Expression`.

```
> \a.a
(λa.a)
> \a.b
(λa.b)
```

12. Now, let's test how well your lexer is holding up! All of these should evaluate the same way. Make sure that yours do.

```
> \f.f x
(λf.(f x))
> \f . f x
(λf.(f x))
> \f.(f x)
(λf.(f x))
> λf.f x
(λf.(f x))
> λf . f x
(λf.(f x))
> λf.(f x)
(λf.(f x))
>    (λf.(f x))
(λf.(f x))
> \    f     .      f       x      ; comment
(λf.(f x))
```

IMPORTANT ASIDE for anyone who chose not to use my starter code: *You may notice that we have just introduced the character* `'λ'`*. This character does not exist in ASCII or ANSII.* ***This means that our project must use UTF encoding, not ASCII/ANSI.*** *This leads us to a small, but infuriating trap: the BOM. I handled this for you in the starter code, but if you are creating your lambda lab from scratch, read on…*

Some computers store bytes left-to-right, and others store them right-to-left. This wasn't a problem until computer networks made it so that they had to talk to each other! In UTF, characters can be many bytes long, so the computer will often put a special character in your string called a Byte Order Marker, which tells computers which direction to read the bytes in your string. (They could be read top to bottom, or bottom to top, and different systems approach them differently by default.) The standard Byte Order Marker is the number 65,279, which makes more sense when you see it in binary: 1111 1110 1111 1111 (hex FEFF). The idea is that if the receiving program reads the BOM as FFFE, it will know to reverse the order it reads the bytes.

This BOM can make a mess of our input strings, and we are not doing network transmissions, so I recommend removing it using the String command `replaceAll("\uFEFF", "")`*. I would do this any time you read in user input.*

With that aside over, let's resume our show:

13. We can override the default order with parenthesis!

```
> (\f.f) x
((λf.f) x)
```

14. Expressions always stack the same way:

```
> \a.a b c d e
(λa.((((a b) c) d) e))
```

15. By default, lambdas (`Function`s) take in **everything** to their right. Think carefully about the trees that form as a result. Note the parenthesis in these two:

```
> \a.a \b.b  ; function of application of variable and function
(λa.(a (λb.b)))
> \a.a \b.b c
(λa.(a (λb.(b c))))
> \a.a b c d e \h. f g h i j
(λa.(((((a b) c) d) e) (λh.((((f g) h) i) j))))
```

16. Again, we can override this with our own parenthesis:

```
> (\a.a) (\b.b) c
(((λa.a) (λb.b)) c)
> \a.a (\b.b) c
(λa.((a (λb.b)) c))
> (\a.a \b.b) c
((λa.(a (λb.b))) c)
```

17. We need to store expressions! Make sure the code that you've written so far correctly interprets such stored definitions. Do you remember what structure we use in Java to store dictionaries?

    You may choose to store either the fully parsed expression or just the tokens array in your dictionary. (If you choose to store only the tokens, your "Added…" strings will be a bit different. That's not a problem as long as the storage is correct.)

    Please note that it is up to your parser to determine that `0` in line 3 and `1` in line 11 are pre-stored values, and not just new one-character variables. That means that the Parser will need access to the dictionary.

```
> 0 = \f.\x.x
Added (λf.(λx.x)) as 0
> 0
(λf.(λx.x))
> 0 = \a.b
0 is already defined.
> 1 = λf.λx.f x
Added (λf.(λx.(f x))) as 1
> succ = \n.\f.\x.f (n f x)
Added (λn.(λf.(λx.(f ((n f) x))))) as succ
> 2 = succ 1
Added ((λn.(λf.(λx.(f ((n f) x))))) (λf.(λx.(f x)))) as 2
```

18. Finally, test out the lexer again to make sure that we don't require spaces (except for where absolutely necessary).

```
> 4=\f.\x.(f(f(f(f x))))
Added (λf.(λx.(f (f (f (f x)))))) as 4
```

Speaking of which…

Running Applications: Getting Started

19. We can run an application with the command "`run`". There will be a lot of code and many helper methods by the time you are done with run. I recommend creating a `Runner` class, with a public static (what is a static method again?) method, `run(…)`. The first target is just to route anything that starts with "run" through that method. Here are three base cases:

(1) Running a `Variable` results in itself.

```
> run cat
cat
```

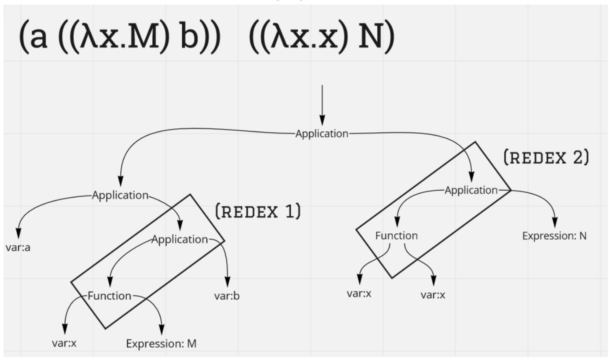
(2) Running an `Application` ***without*** lambdas also results in itself:

```
> run (x y)
(x y)
```

(3) Running a single `Function` also results in itself.

```
> run \x.x
(λx.x)
> run \x.x y
(λx.(x y))
```

Running Beta-reductions on Applications: Redexes



20. So far, so good, because none of those made any changes. The only things that change when they run are `Application`s with `Function`s on the left:

```
> run (\x.x) y
y
```

The above is the literally simplest possible runnable thing. Think for a moment about how you could detect the situations in which you would actually do something. Some might find it helpful to create a helper method. The structure above is:

```
              ┌────[APP]────┐
     ┌───[FUNC]───┐      [VAR: y]
[VAR: x]      [VAR: x]
```

This structure is called a **redex** in Lambda Calculus. **Redexes** are the only things that can actually run. Of course, redexes can be more complex. We could, for instance, have

```
> run (\x.(x y x)) (y y)
(((y y) y) (y y))
```

This is because a **redex** really comes in the form:

```
              ┌────[APP]────┐
     ┌───[FUNC]───┐      RedexInsert
 RedexVar     RedexReturn
```

(Note that, for purposes of clarity, I will use the names **RedexVar**, **RedexReturn** and **RedexInsert** for the remainder of the lab writeup). And what they return is some modified **RedexReturn**, with **RedexInsert** substituted in wherever **RedexVar** is found. Your job now is to be able to run the simple redexes above.

A word of caution! If a function has a sub-function with the same parameter name, any variables inside that inner function will no longer be bound to the outer function:

```
> run (\x. x (\x.x)) y
(y (λx.x))
```

Similarly,

```
> run (\x. \x . x) y
(λx.x)
```

21. But running an application should also run the result of the lambda expression. In this case, we are passing the identity function `(λv.v)` into `(λx.(x y))`, which should bring us `(λv.v) y`, which will resolve to `y`:

    ```
    > run (\x . x y) (\v.v)
    y
    ```

    Order matters here. We always want to run the left-most application possible in the toString. (In this case, `\x` would run before `\v`.) Again, it helps to look at the tree. How would you find the FIRST application to run just by looking at the tree itself?

    ```
                  ┌──────────────[APP]──────────────┐
         ┌───[FUNC]─────┐                   ┌───[FUNC]───┐
    [VAR: x]      ┌────[APP]────┐       [VAR: v]      [VAR: v]
            [VAR: x]        [VAR: y]
    ```

    You must always run the leftmost runnable redex, so you can't just recurse down willy-nilly. My hint for you is to use some sort of while loop to detect the next redex and run it. Stop when there are no more redexes to run.

You'll have to set up some sort of loop that looks like this in order to run each algebraic step:

```
while (there is a redex left to do) {
    Expression e = calculateModifiedRedexReturn()
    Expression f = insertBetaReductionAtCorrectLocation(e, originalExpression)
}
```

To break this pseudo-code down a bit, both functions find the leftmost redex. Using the expression `(((λx.x) y) z)` as an example, we ultimately want `(y z)`, but there are two steps to get there. The first step is to figure out that `((λx.x) y)` is properly `y`, and the second step is to reinsert that into the original expression to arrive at `(y z)`.

That pseudo-code is very pseudo, of course. Each run through the loop will do exactly one beta reduction. *This gives you a nice advantage if you wish to use it!* Many students will want to make some command like "ALGEBRA" or "DEBUG" that you can type at the console that will trigger a print at every turn through this loop, so that you can see each *algebraic* step take place. Such a mode can be very useful when you are debugging things later.

22. We can't just use shallow copies when substituting expressions. We'll need to make deep copies of the functions we pass in:

```
> run (\x.(x a) (x b))  (\i.i (\v. i))
((a (λv.a)) (b (λv.b)))
```

23. And, of course, we can `run` while setting a variable:

```
> ball = run (\x.x) bounce
Added bounce as ball
```

24. **Remember variable capture? It's illegal, and we can't do it.** Every lambda has a "local variable" parameter. The variables with the same name inside the Function's expressions are the Function's *bound variables*.

This is one of the most complicated things to get right! The rule is that if any of the *free variables* being passed in are crossing the threshold of a *bound variable* of the same name, the *bound variable* must change its name.

To solve this, there are alpha-reductions, wherein we are forced to change the names of variables: We are always free to change the name of bound variables, as long as we also change their children.

If a *free* variable conflicts with a *bound* variable during an application, we change the name of the *bound* variable (and its children). **We can NEVER change the name of a free variable under any circumstances.**

In the example below, because the free `x` and the bound `x` share the same name, we change the name of the bound `x`. So you can more easily see what is going on below, my hint is that `(λy.λx1.(x1 y)) x` would be an intermediate step here:

```
> run (λy.λx.(x y)) x
(λx1.(x1 x))
```

So here are some questions you will need to ask: (1) what are the free variables in **RedexInsert**? (2) As I am searching for the substitutions I need to make within **RedexReturn** during a regular beta reduction, do I enter into any subfunctions that have name clashes with those free variables? (3) If I do enter into a subfunction with a name clash, are there any actual variables in there bound to the original r**RedexVar**?

If the answer to that last question is YES, then you need an alpha reduction.

The last question, then, is (4) what name can I choose for my alpha reduction?

Some people give their variables an increment method, so that z becomes z1, z2, z3, etc. This is probably a great approach, but you still need to be sure that there is no similarly named variable in the list of free variables in **RedexInsert**, otherwise you will still be making an illegal capture.

```
> run (λy.λx.(x y)) (x x)
(λx1.(x1 (x x)))
```

I chose to use numbers for creating new names, but any scheme that you choose is fine as long as conflicts are avoided. *Any η-equivalents of my answers are absolutely fine. If my answer is* `(λy.λx1.(x1 y))` *and yours is* `(λy.λa.(a y))`*, that does not mean that there are any problems.*

Please note that this step may require a large amount of refactoring. How can you even detect which variables are free and which are bound? (This is one place where students who chose to create `FreeVars`, `BoundVars`, and `Parameters` get a real payoff for all of the tricky work they did earlier in locating and deep copying!) There will be many helper methods. It also may be useful to go through and edit your `Variable` classes.

Appendix 1: Example Run

Here is a sample run of inputs and outputs that you might find useful while creating and testing your code. I provide them here as a gift to you because they were useful to me while testing my own version. I stress again that if your answers are *eta-equivalent* to mine, they are fine! If my `f` comes out as your `f2`, you have nothing to worry about.

```
> ; just a blank line
>
> x
x
> x;withcommentnospaces
x
> 0 = \f.\x.x
Added (λf.(λx.x)) as 0
> succ = \n.\f.\x.f (n f x)
Added (λn.(λf.(λx.(f ((n f) x))))) as succ
> 1 = run succ 0
Added (λf.(λx.(f x))) as 1
> + = λm.λn.λf.λx.(m f) ((n f) x)
Added (λm.(λn.(λf.(λx.((m f) ((n f) x)))))) as +
> * = λn.λm.λf.λx.n (m f) x
Added (λn.(λm.(λf.(λx.((n (m f)) x))))) as *
> 2 = run succ 1
Added (λf.(λx.(f (f x)))) as 2
> 3 = run + 2 1
Added (λf.(λx.(f (f (f x))))) as 3
> 4 = run * 2 2
Added (λf.(λx.(f (f (f (f x)))))) as 4
> 5 = (λf.(λx.(f (f (f (f (f x)))))))
Added (λf.(λx.(f (f (f (f (f x))))))) as 5
> 7 = run succ (succ 5)
Added (λf.(λx.(f (f (f (f (f (f (f x))))))))) as 7
> pred = λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)
Added (λn.(λf.(λx.(((n (λg.(λh.(h (g f))))) (λu.x)) (λu.u))))) as pred
> 6 = run pred 7
Added (λf.(λx.(f (f (f (f (f (f x)))))))) as 6
> - = λm.λn.(n pred) m
Added (λm.(λn.((n (λn.(λf.(λx.(((n (λg.(λh.(h (g f))))) (λu.x)) (λu.u)))))) m))) as -
> 10 = run succ (+ 3 6)
Added (λf.(λx.(f (f (f (f (f (f (f (f (f (f x))))))))))))) as 10
> 9 = run pred 10
Added (λf.(λx.(f (f (f (f (f (f (f (f (f x)))))))))))) as 9
> 8 = run - 10 2
Added (λf.(λx.(f (f (f (f (f (f (f (f x))))))))))) as 8
> true = λx.λy.x
Added (λx.(λy.x)) as true
> false = 0
Added (λf.(λx.x)) as false
> not = λp.p false true
Added (λp.((p (λf.(λx.x))) (λx.(λy.x)))) as not
> even? = λn.n not true
Added (λn.((n (λp.((p (λf.(λx.x))) (λx.(λy.x))))) (λx.(λy.x)))) as even?
> odd? = \x.not (even? x)
```

```
Added (λx.((λp.((p (λf.(λx.x))) (λx.(λy.x)))) ((λn.((n (λp.((p (λf.(λx.x)))
(λx.(λy.x))))) (λx.(λy.x)))) x))) as odd?
> run even? 0
true
> run even? 5
false
> run (λy.λx.(x y)) (x x)
(λx.(x (x x)))
> run (\x. \x . x) y z
z
```

Appendix 2: The Tests from the Creation Guide

```
; PART 1
a
cat
; This comment should do nothing. run (\y.\x.(x y)) (x x)
a     ; comment
a
; spaces before the comment should still do nothing!
(   (\bat  .bat flies)cat   );     comment
a b
a b c d
a b c (d e)
(((a)))
((((a))) (((b))))
\a.a
\a.b
\f.f x
\f . f x
\f.(f x)
λf.f x
λf . f x
λf.(f x)
(λf.(f x))
\    f     .      f       x      ; comment
(\f.f) x
\a.a b c d e
\a.a \b.b
\a.a \b.b c
(\a.a) (\b.b) c
\a.a (\b.b) c
(\a.a \b.b) c
\a.a b c d e \h.f g h i j
0 = \f.\x.x
0
0 = \a.b
1 = \f.\x.f x
succ = \n.\f.\x.f (n f x)
2 = run succ 1
4=\f.\x.(f(f(f(f x))))
run cat
run (x y)
run \x.x
run \x.x y
run (\x.x) y
run (\x . x y) (\v.v)
run (\x.(x a) (x b))  (\i.i (\v. i))
ball = run (\x.x) bounce
run (\x. x (\x.x)) y
run (λy.λx.(x y)) x
run (λy.λx.(x y)) (x x)

; PART 2
; just a blank line
```

```
x
x;withcommentnospaces
0 = \f.\x.x
succ = \n.\f.\x.f (n f x)
1 = run succ 0
+ = λm.λn.λf.λx.(m f) ((n f) x)
* = λn.λm.λf.λx.n (m f) x
2 = run succ 1
3 = run + 2 1
4 = run * 2 2
5 = (λf.(λx.(f (f (f (f (f x)))))))
7 = run succ (succ 5)
pred = λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)
6 = run pred 7
- = λm.λn.(n pred) m
10 = run succ (+ 3 6)
9 = run pred 10
8 = run - 10 2
true = λx.λy.x
false = 0
not = λp.p false true
even? = λn.n not true
odd? = \x.not (even? x)
run even? 0
run even? 5
run (λy.λx.(x y)) (x x)
run (\x. \x . x) y z
```

Appendix 3: Other Useful Functions For Copy/Paste and Testing

```
; BOOLEANS AND BRANCHING
true = λx.λy.x
false = \f.\x.x   ; same as 0
and = λp.λq.p q p
or = λp.λq.p p q
not = λp.p false true
xor = \p.\q.p (not q) q
if = λb.λT.λF.((b T) F)

; NUMBER OPERATIONS
succ = \n.\f.\x.f (n f x)
pred = λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)
+ = λm.λn.λf.λx.(m f) ((n f) x)
* = λn.λm.λf.λx.n (m f) x
- = λm.λn.(n pred) m
even? = λn.n not true
odd? = \x.not (even? x)
zero? = \n.n (\x.false) true
leq? = \m.\n.zero?(- m n)     ; "less than or equal to"
lt? = \a.\b.not (leq? b a)
gt? = \a.\b.not (leq? a b)
eq? = \m.\n.and (leq? m n) (leq? n m)
neq? = (not (eq? a b)) ; "not equal"
geq? = \a.\b.(leq? b a)


; GENERATING NUMBERS WITH RUN
0 = \f.\x.x    ; same as false
1 = run succ 0
2 = run succ 1
3 = run + 2 1
4 = run * 2 2
5 = (λf.(λx.(f (f (f (f (f x)))))))
7 = run succ (succ 5)
6 = run pred 7
10 = run succ (+ 3 6)
9 = run pred 10
8 = run - 10 2


; LINKED LISTS
cons = λx.λy.λf.f x y   ; makes a cons pair (x y)
car = λp.p true
cdr = λp.p false
null = \x.true
null? = λp.p (λx.λy.false) ; true if null, false if a pair, UNDEFINED otherwise


; Y COMBINATOR
Y = λf. (λx. f(x x)) (λx. f(x x))
```

```
; FUN FUNCTIONS THAT USE Y
factorial = Y \f.\n.(if (zero? n) 1 (* n (f (- n 1))))
; divpair returns a cons box of the quotient and remainder of a division
divpair = Y (λg.λq.λa.λb. lt? a b (cons q a) (g (succ q) (- a b) b)) 0
/ = λa.\b. car (divpair a b)
mod = λa.\b. cdr (divpair a b)

; Now we can make statements like
; run factorial 3
; run + 2 (factorial 3)
; run (/ (* 3 6) 2)
```

Appendix 4: Extra Credits

EXTRA CREDIT 1

If the result of a run is a defined expression, provide that expression instead of the lambda result. Given these definitions:

succ = \n.\f.\x.f (n f x)
0 = \f.\x.x
1 = run succ 0
2 = run succ 1
3 = run succ 2
4 = run succ 3
5 = run succ 4
6 = run succ 5
7 = run succ 6
8 = run succ 7
9 = run succ 8
10 = run succ 9
11 = run succ 10
12 = run succ 11
+ = λm.λn.λf.λx.(m f) ((n f) x)

```
> run + 3 4
7
```

If there are two defined terms that are η-equivalent, your program may report either *or both*:

```
> 0 = \f.\x.x
Added (λf.(λx.x)) as 0
> false = \a.\b.b
Added (λa.(λb.b)) as false
> true = \a.\b.a
Added (λa.(λb.a)) as true
> not = λp.p false true
Added (λp.((p (λf.(λx.x))) (λx.(λy.x)))) as not
> run not true
0
false
```

EXTRA CREDIT 2

Populate the Church Numerals in a specified range using the command `populate`. (The actual subtraction here would take an extremely long time!)

```
> pred = λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)
Added (λn.(λf.(λx.(((n (λg.(λh.(h (g f))))) (λu.x)) (λu.u)))))
> - = λm.λn.(n pred) m
(λm.(λn.((n (λn.(λf.(λx.(((n (λg.(λh.(h (g f))))) (λu.x)) (λu.u))))))
m))) as -
> populate 0 250
Populated numbers 0 to 250
> run - 240 238
2
```