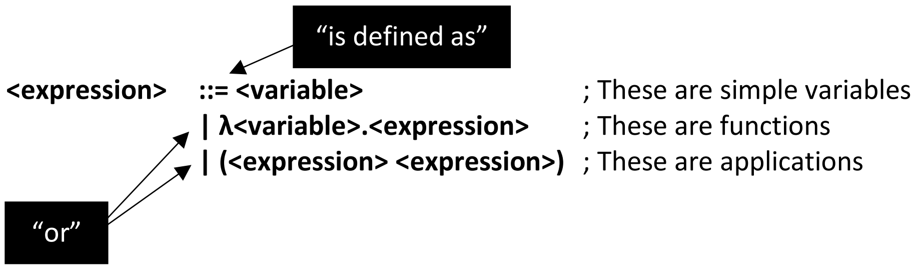
Lambda Applicator Lab

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.

Implementation Hints in Java

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 `Expressions` 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 are several other implementation choices you can make for Variables. I did not do any of 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.

Doing these 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.

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

The Features

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 enter ↵ 3 times, and then types e x i t and enter ↵ again:

```
>
>
>
> exit
Goodbye!
```

Now let's add in the ability to comment. Basically, if you find a `;` character, ignore the rest of the line.

```
>
> ; This comment should do nothing. define run (lambda y. lambda x.(x
y)) (x x)
>
> exit
Goodbye!
```

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
```

Make sure that you haven’t broken comments! Already, there is a bit of hidden complexity here, since the `a` added should not have extra spaces in its name.

```
> a     ; comment
a
```

Now let's create some `Application`s. We won’t Beta-reduce them yet, just create them. Once again, 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`.

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

This created the following tree. The `toString()` of an application would return `"(" + left + " " + right + ")"`.

```
┌─[APP]─┐
a       b

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

This created the following tree:

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

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.

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

```
> \a.a
(λa.a)
> \a.b
(λa.b)
```

Now, let’s focus on some spacing. 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))
```

You may notice that we have just introduced a non-standard character here, λ. This character does not exist in ANSII. ***This means that our project must use UTF encoding, not ASCII/ANSI.*** *Which leads us to a small, but infuriating trap, the BOM.*

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:

We can override the default order with parenthesis!

```
> (\f.f) x
((λf.f) x)
```

Expressions always stack the same way:

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

By default, lambdas (`Function`s) take in **everything** to their right. Note the parenthesis in these two:

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

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)
> \a.a b c d e \h. f g h i j
(λa.(((((a b) c) d) e) (λh.((((f g) h) i) j))))
```

Finally, extraneous parenthesis are stripped away:

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

Storing Values

We can store expressions!

```
> 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
```

We can also `run` a statement during a definition. There will be more on running later!

```
> 3 = run succ 2
Added (λf.(λx.(f (f (f x))))) as 3
```

Finally, 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

We can run an application with the command "`run`". There are three simple 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))
```

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
```

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
```

Remember variable capture? It's illegal, and we can't do it. Every lambda has a "local variable" parameter. These are called *bound variables*. In this case, the `x` in the inner lambda function is unrelated to the `x` in the outer lambda function, so the inner `x` is unaffected:

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

We also must be careful about **which** parameter a bound variable to bound to. In the example below, the bound `x` (this one: `(\x. \x . `**`x`**`)`) is actually bound to the second parameter (this one: `(\x. \`**`x`**` . x)`).

```
> run (\x. \x . x) y
(λx.x)
```

Finally, 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))
```

The right side of an application does not have to be a `Variable`. It can be *any* expression, and any free variable in the right expression can potentially conflict with a bound variable in the left expression.

```
> 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!*

Examples

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 *equivalent* to mine, they are fine! If my `f2` comes out as your `f`, 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 (λf1.(λx1.(f1 x1))) 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 (λf2.(λx2.(f2 (f2 (f2 x2))))) as 3
> 4 = run * 2 2
Added (λf1.(λx1.(f1 (f1 (f1 (f1 x1)))))) 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 (λf1.(λx1.(f1 (f1 (f1 (f1 (f1 (f1 x1)))))))) 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 (λf1.(λx1.(f1 (f1 (f1 (f1 (f1 (f1 (f1 (f1 (f1 (f1 x1))))))))))) 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 (λf3.(λx3.(f3 (f3 (f3 (f3 (f3 (f3 (f3 (f3 x3)))))))))) 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)
(λx1.(x1 (x x)))
> run (\x. \x . x) y z
z
```

Useful Functions For Copy/Paste and Testing

```
; BOOLEANS AND BRANCHING
true = λx.λy.x
false = \f.\x.x   ; same as 0
not = λp.p false true
and = λp.λq.p q p
or = λp.λq.p p q
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)
```

Typo fixed!

```
; 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)
```

Typo fixed!

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
```