Lambda tut handout
(by: Alex Jago)

Booleans in Lambda Calculus (Q3, Q4)
True: returns first argument (of two). False: returns second argument

$T := (\lambda p.\lambda q.p)$ $F := (\lambda p.\lambda q.q)$
This encoding means False and Zero have the same representation!

$$\boldsymbol{NOT := (\lambda p.p\ F\ T)}$$

"If P is true, return false; if P is false, return true"

$$\boldsymbol{AND := \lambda p.\lambda q.p\ q\ F}$$

"If my first argument P is true then return my second argument Q, else return false" (iff Q is true then the overall expression will be true)

$$\boldsymbol{OR := \lambda p.\lambda q.p\ T\ q}$$

"If P is true, return true, else return Q (and see what it is)"
Practice: try **OR False True**

Worked example: AND True False

$$(\lambda p.\lambda q.p\ q\ F)\ T\ F$$

Substitute in "green True" for p, "blue False" for q

$$T\ F\ F$$

$$(\lambda a.\lambda b.a)\ F\ F$$

Substitute in "blue False" for a and expand

$$\left(\lambda b.(\lambda x.\lambda y.y)\right)\ F$$

Substitute in "yellow False" for b (it disappears)

$$\lambda x.\lambda y.y$$

$$F$$

Result: False

Church Encoding for Numerals (Q1)

$$\lambda f.\lambda x.f\left(f\left(f(fx)\right)\right) = N_4$$

$$\lambda f.\lambda x.x = N_0$$

$$\lambda f.\lambda x.fx = N_1$$

Successor function:

$S := \lambda n.\lambda g.\lambda w.\ g\ (n\ g\ w)$

$$S\ N_2$$

$$(\lambda n.\lambda g.\lambda w.g\ (n\ g\ w)\ )\ (\lambda f.\lambda x.f(fx))$$
$$\lambda g.\lambda w.g\ \left((\lambda f.\lambda x.f(fx))\ g\ w\right)$$
$$\lambda g.\lambda w.g\ (g\ (g\ w)) = \mathrm{N}_3$$

Practice – turn a 0 into a 1, then a 3 into a 4

Doubling a Church Numeral (Q2)

$$\lambda f.\lambda x.f\big(f(fx)\big) = N_3$$

Doubler: change of variables, put one copy inside the other.

$$D := \lambda n.\lambda g.\lambda w.n\ g\ (n\ g\ w)$$

Say what? Let's continue the worked example.

$$D\ N_3$$
$$(\lambda n.\lambda g.\lambda w.n\ g\ (n\ g\ w))\ N_3$$

N3 substitutes for n

$$(\lambda g.\lambda w.N_3\ g\ (N_3\ g\ w))$$

Apply N3 to g:

$$\lambda g.\lambda w.N_3\ g\ \left(\left(\lambda f.\lambda x.f\big(f(fx)\big)\right)\ g\ w\right)$$

Sub in g for f:

$$\lambda g.\lambda w.\left(\lambda x.g\big(g(gx)\big)\right)\ \left(\left(\lambda x.g\big(g(gx)\big)\right)\ w\right)$$

Sub in w for x in RH part inside brackets

$$\lambda g.\lambda w.\left(\lambda x.g\big(g(gx)\big)\right)\ \left(\left(\ g\big(g(gw)\big)\right)\right)$$

Apply blue to green by subbing in green for blue x:

$$\lambda g.\lambda w.\left(\ g\left(g\left(g\left(\ g\big(g(gw)\big)\right)\right)\right)\right) = N_6$$

Addition: only slightly more complicated than doubling

$$+ := \lambda m.\lambda n.\lambda g.\lambda w.m\ g\ (n\ g\ w)$$

Multiplication

$$* := \lambda m.\lambda n.\lambda g.m\ (n\ g)$$

Demo: 3 * 2

$$* \ \boldsymbol{N_3} \ \boldsymbol{N_2}$$
$$(\lambda m.\lambda n.\lambda g\ .m\ (n\ g))\ \boldsymbol{N_3}\ \boldsymbol{N_2}$$
$$\lambda g.\boldsymbol{N_3}\ (\ \boldsymbol{N_2}\ g)$$

Partially apply N2 to g:

$$\lambda g.\boldsymbol{N_3}\ (\ (\lambda f.\lambda x.f(fx))\ g)$$
$$\lambda g.\boldsymbol{N_3}\ (\ (\lambda x.g(gx))\ )$$

Partially apply N3 to blue

$$\lambda g.\left(\lambda f.\lambda y.f(f(fy))\right)\ (\ (\lambda x.g(gx))\ )$$
$$\lambda g.\left(\boldsymbol{\lambda y.}(\lambda x.g(gx))\left((\lambda x.g(gx))\left((\lambda x.g(gx))\boldsymbol{y}\right)\right)\right)$$

Sub in green y for blue x in right brackets

$$\lambda g.\left(\lambda y.(\lambda x.g(gx))\left((\lambda x.g(gx))\left((g(g\boldsymbol{y}))\right)\right)\right)$$

Recolour ...

$$\lambda g.\left(\lambda y.\ (\lambda x.g(gx))\left((\lambda x.g(gx))\left((g(g\boldsymbol{y}))\right)\right)\right)$$

Sub blue into green

$$\lambda g.\left(\lambda y.\ (\lambda x.g(gx))\left(g\left(g\left((g(g\boldsymbol{y}))\right)\right)\right)\right)$$

Sub green-blue into yellow

$$\lambda g.\left(\lambda y.\ \left(g\left(g\left(g\left(g\left((g(gy))\right)\right)\right)\right)\right)\right)$$

Clean up a lot!

$$\lambda g.\lambda y.g\left(g\left(g\left(g(g(gy))\right)\right)\right) = N_6$$

DIY: multiply one and zero

XOR gates, real quick (Q5)

The hard way (one of them, at least)

$$\boldsymbol{XOR := \lambda p.\lambda q.\ AND\ (NOT\ (AND\ p\ q))\ \ (OR\ p\ q)}$$

The easy way (conditional negation!)

$$\boldsymbol{XOR :=\ \lambda p.\lambda q.p\ (NOT\ q)\ q}$$

If-Else function (Q6)

Despite the question, this will take three arguments *p, x, y*
"IF p THEN x ELSE y"
If we think of p as a Boolean we can write this trivially:
$\text{IFELSE} := \lambda p. \lambda x. \lambda y. p\ x\ y$

Y Combinator (Q7)

We need **controlled recursion, so we need a base case.**

First, consider the M combinator $\lambda f. ff$ and its self-application:

$$(\lambda f. ff)(\lambda g. gg)$$
$$(\lambda g. gg)(\lambda g. gg)$$
$$(\lambda g. gg)(\lambda g. gg)$$

It's the same thing! We've just created an infinite recursion. We don't have loops in Lambda but we do have recursion, so this is a principle we need to develop further.

What we want is a combinator that lets us do controlled recursion, with a base case.

This is the **Y-combinator:** it results in $Y\ R \Rightarrow R\ (Y\ R)$ which for appropriate choice of R lets us do very cool things...

You shouldn't necessarily think of $Y\ R$ on its own being the whole computation – see how the Factorial operator is applied. It's a building block. **R supplies the base case and the computation, Y supplies the recursion.**

What is the Y combinator? $\boldsymbol{\left(\lambda f. \left(\lambda x. f(xx)\right)\left(\lambda x. f(xx)\right)\right)}$

Exercise:

$$Y\ R$$
$$\left(\lambda f. \left(\lambda x. f(xx)\right)\left(\lambda x. f(xx)\right)\right)\ R$$
$$\left(\lambda x. R(xx)\right)\left(\lambda x. R(xx)\right)$$
$$\left(\lambda y. R(yy)\right)\left(\lambda x. R(xx)\right)$$
$$R\left(\left(\lambda x. R(xx)\right)\left(\lambda x. R(xx)\right)\right)$$

Notice that the part in blue is identical to an earlier line:
$(\lambda x. R(xx))(\lambda x. R(xx))$

Therefore $Y\ R => R\ (Y\ R) => R\ \Big(R\ \Big(R\ (R\ (\ldots Y\ R))\Big)\Big)$
(though not quite $Y\ R$ but rather an equivalent expression)

$$Y\ R\ N$$
$$R\ (Y\ R)\ N$$

What if R was like a Boolean?

$$Y\ F$$
$$\Big(\lambda f. (\lambda x. f(xx))(\lambda x. f(xx))\Big)\ (\lambda a. \lambda b. b)$$
$$F\ (Y\ F)$$

So if we had $Y\ F\ N$ (with F just being the simplest possible "base case" for control flow.

$$F\ (Y\ F)\ N$$
$$(\lambda a. \lambda b. b)\ (Y\ F)\ N$$
$$N$$

Lists (with Mikrokosmos) (Q8)

There are a few ways to do lists in Lambda calculus. The general principle is a right-recursive pairing, a bit like a linked list:

[1, 2, 3, 4] is represented as [1, [2, [3, [4, *nil*]]]]

The ***nil*** object is just another version of False or Zero: $\lambda p. \lambda q. q$

Mikrokosmos defines three operators: ***cons*** to construct a list, ***head*** to get the "leftmost" element of the list, and ***tail*** to get the rest of the list. These are all quite complicated objects.

list = *cons 1 (cons 2 (cons 3 (cons 4 nil))))*
***head* list => 1**
tail* list => *cons 2 (cons 3 (cons 4 nil)))

(Lists continued)

https://mroman42.github.io/mikrokosmos/index.html

Exercise: using Mikrokosmos, construct a list of five items. Retrieve each element of the list using a combination of **head** ***and*** **tail**

list = cons 3 (cons 5 (cons 7 (cons 11 (cons 13 nil))))

Exercise: using Mikrokosmos, develop an **index function.**
This function should take a list S and Church numeral N and return the item at the Nth index of the list (where index 0 is given by head)

$$\mathrm{INDEX} = \lambda S.\, \lambda N.\, \mathbf{head}\ (N\ \mathbf{tail}\ S)$$

Due to the structure of the Church numeral, we get head(tail(...(tail S)...))

Just the one beta reduction

$((\lambda x.\ (\lambda y.\ yx))\ x)\ (\lambda z.\ w)$

$((\lambda \mathrm{v}.\ (\lambda y.\ y\mathrm{v}))\ x)\ (\lambda z.\ w)$

$(((\lambda y.\ y\ x)))\ (\lambda z.\ w)$

$(\lambda y.\ y\ x)\ (\lambda z.\ w)$

$((\lambda z.\ w)\ x)$

w

You can try other reductions by hand and verify them in Mikrokosmos.

Mikrokosmos doesn't like free variables, but we can substitute a number and then spot the number in the output.

Defining the predecessor function (III.1)

We don't have a way to represent negative numbers, so we can't just add negative one. We need to find the number *Y* such that *Y* + *1* == *X*.

Alternatively, just like the successor function added on one more *f*, we want to get rid of one of them.

Let's explain.

$$\text{PRED} := \lambda n.\lambda g.\lambda y.\ n\ (\lambda a.\lambda b.b\ (a\ g))\ (\lambda v.y)\ (\lambda u.u)$$

We'll apply our numeral ***n*** to the yellow and green sub-expressions, then apply the result to the blue sub-expression.

This means if $n := \lambda f.\lambda x.fx = \boldsymbol{N_1}$ then we'll have:

$$\lambda g.\lambda y.\ (\boldsymbol{\lambda f.\lambda x.fx})\ (\lambda a.\lambda b.b\ (a\ g))\ (\lambda v.y)\ (\lambda u.u)$$
$$\lambda g.\lambda y.\ \big((\lambda a.\lambda b.b\ (a\ g))\ (\lambda v.y)\big)\ \ (\lambda u.u)$$

So that means the green thing subs in for *a:*

$$\lambda g.\lambda y.\ \big((\lambda b.b\ ((\lambda v.y)\ g))\ \big)\ \ (\lambda u.u)$$

... and we can discard that inner *g*

$$\lambda g.\lambda y.\ \big((\lambda b.b\ (y))\ \big)\ \ (\lambda u.u)$$

... and drop some brackets

$$\lambda g.\lambda y.\ \ (\lambda b.b\ (y))\ \ \ (\lambda u.u)$$
$$\lambda g.\lambda y.\ \ (\lambda u.u)\ y$$
$$\lambda g.\lambda y.y = N_0$$

OK so WTF just happened?

Let's try again but with N_2

$$\lambda g.\lambda y.\ (\boldsymbol{\lambda f.\lambda x.f(fx)})\ \ (\lambda a.\lambda b.b\ (a\ g))\ (\lambda v.y)\ (\lambda u.u)$$
$$\lambda g.\lambda y.\ \big((\lambda a.\lambda b.b\ (a\ g))\ ((\lambda a.\lambda b.b\ (a\ g))\ (\lambda v.y))\big)\ \ (\lambda u.u)$$

Green subs in for *a* again

$$\lambda g.\lambda y.\ \big((\lambda a.\lambda b.b\ (a\ g))\ ((\lambda b.b\ ((\lambda v.y)\ g))\ )\big)\ \ (\lambda u.u)$$

Thus g disappears again...

$$\lambda g.\lambda y.\ ((\lambda a.\lambda b.b\ (a\ g))\ (\lambda b.b\ y))\ \ (\lambda u.u)$$

Sub yellow-green result into purple...

$$\lambda g.\lambda y.\ ((\lambda b.b\ ((\lambda b.b\ y)\ g))\ )\ \ (\lambda u.u)$$
$$\lambda g.\lambda y.\ ((\lambda b.b(gy))\ )\ \ (\lambda u.u)$$
$$\lambda g.\lambda y.\left(\left(\ (\lambda u.u)(gy)\right)\right)$$
$$\lambda g.\lambda y.g\ y = N_1$$

That gives us a little bit more insight. When we sub green into yellow, we effectively throw away a copy of g. But when we sub the *result* of that into purple, we keep purple's copy of g.
Finally, the blue identity function lets us clean up at the end.

Try it with N_3 to really lock it in

$$\lambda g.\lambda y.\ (\lambda f.\lambda x.f(f(fx)))\ \ (\lambda a.\lambda b.b\ (a\ g))\ (\lambda v.y)\ (\lambda u.u)$$

$$\lambda g.\lambda y.\ ((\lambda a.\lambda b.b\ (a\ g))((\lambda a.\lambda b.b\ (a\ g))((\lambda a.\lambda b.b\ (a\ g))(\lambda v.y))))\ \ (\lambda u.u)$$

$$\lambda g.\lambda y.\ ((\lambda a.\lambda b.b\ (a\ g))((\lambda a.\lambda b.b\ (a\ g))((\lambda b.b\ ((\lambda v.y)\ g)))))\ \ (\lambda u.u)$$

$$\lambda g.\lambda y.\ (\ (\lambda a.\lambda b.b\ (a\ g))\ ((\lambda a.\lambda b.b\ (a\ g))(\lambda b.b\ y)))\ \ (\lambda u.u)$$

$$\lambda g.\lambda y.\ (\ (\lambda a.\lambda b.b\ (a\ g))\ ((\lambda b.b\ ((\lambda b.b\ y)\ g)))\ )\ \ (\lambda u.u)$$

$$\lambda g.\lambda y.\ \left(\ (\lambda a.\lambda b.b\ (a\ g))\ (\lambda b.b\ (gy))\right)\ \ (\lambda u.u)$$

$$\lambda g.\lambda y.\ (\ (\lambda b.b\ ((\lambda b.b\ (gy))\ g))\ )\ \ (\lambda u.u)$$

$$\lambda g.\lambda y.\ (\ (\lambda b.b\ ((\ g\ (gy)))))\ )\ \ (\lambda u.u)$$

$$\lambda g.\lambda y.\ \left(\left((\lambda u.u)\ ((\ g\ (gy)))\right)\right)$$

$$\lambda g.\lambda y.g\ (g\ y) = N_2$$

Hurray!

Absolute Differences (III.2)

Note: we can test this section using Mikrokosmos. $PRED$ *is defined as* `pred` *and* EQ *as* `eq`*.*

We can use the predecessor function, applied repeatedly, to define a saturating minus:

$$\text{SATMINUS} = \lambda x. \lambda y. y \text{ PRED } x$$

But this is not quite a difference function…

IsZero (III.3)

Mikrokosmos pre-defines an EQ function, as `eq`. It is **very** complicated. We could then, quite easily, define an "is zero" function:

$$\text{ISZERO} = \lambda n. \text{EQ } 0 \; n$$

We can also exploit the identity between zero and false:

$$\text{ISZERO} = \lambda n. n \; (T \; F) \; T$$

Assuming N is a Church numeral, if zero, then it also acts like a Boolean and returns its second argument, True. If N is a non-zero numeral, then the $(T \; F)$ subs in for its first argument and the F for its second argument.Once simplified, a False will be returned.

Less than and Greater than (III.4)

We know that iff $x - y \leq 0$ then SATMINUS will return zero. Therefore we can use the result of SATMINUS to test whether $x \leq y$.

$$\text{LEQ} = \lambda x. \lambda y. \; \text{ISZERO } (\text{SATMINUS } x \; y)$$

Testing whether $x \geq y$ is a simple change to the argument order:

$$\text{GEQ} = \lambda x. \lambda y. \; \text{ISZERO } (\text{SATMINUS } y \; x)$$

Equality (III.5)

We can also write equality as "GEQ AND LEQ", which is what Mikrokosmos does too, as it turns out:

$$\text{EQ} = \lambda x. \lambda y. \; \text{AND } (\text{GEQ } x \; y) \; (\text{LEQ } x \; y)$$

Absolute Differences again

We can now put the pieces together. (GEQ x y) will give us a Boolean, which we can use to select between (x - y) and (y – x).

$$\text{ABSDIFF} = \lambda x. \lambda y. (\text{GEQ } x \; y) \; (\text{SATMINUS } x \; y) \; (\text{SATMINUS } y \; x)$$

Euclidean Algorithm for the GCD (III.6)

First, for reference, let's write the Euclidean GCD algorithm in Python...

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

So we'll need to build up from the following:

- take two arguments
- do an "is (not) zero" comparison
- modulus
- recursion

Anyway... we'll use the pre-defined modulus here, which saves a *tonne* of work. The Mikrokosmos keyword for this is simply `mod`

The real challenge is plumbing it all together...

Let's take Shakes' textbook's factorial function as an example: write

```
R = \f. \n. ISZERO(n)(1) MUL(n) (f(PRED(n)))
```

Then we get the recursion element via applying the Y combinator (spelled `fix`) to it. Clearly we have to do something similar here.

Mikrokosmos also predefines an `iszero` operator, (which we did in Q6) so we'll reuse that too. It also predefines the `Y` combinator as `fix`.

Like the factorial function in the textbook, we'll define a "kernel" function *R* and apply the *Y* combinator to it to make our *GCD* function.

```
R_inprogress = \a. \b. iszero(b) (a) (...)


R_maybe = \a. \b. iszero b a (b (mod a b))

fix R_maybe {x} {y}
```

Well, almost. We actually need to prepend and insert another argument which we'll call *f*.
This *f* we insert is just like the *f* argument which we needed in the factorial function.

(for Mikrokosmos' sanity, tell it not to pre-expand yet, by using !=):

```
R != \f. \a. \b. (iszero b) (a) (f b (mod a b))

GCD != fix R

GCD 15 6
```

We can define our own modulo in a similar way.

```
def modulo(a, b):
    # calculates a % b
    if a < b:
        return a
    else:
        return modulo(a - b, b)
```

We'll need a strict less-than:

```
lt = \x.\y. and (leq x y) (not (geq x y))
```

Now using the same principles as before:

```
MMM = \f.\a.\q. (lt a q) (a) (f (minus a q) q)
modulo != fix MMM
```