Lambda Calculus

Prepared by Mark on October 18, 2023

Instructor's Handout

This file contains solutions and notes.
Compile with the "nosolutions" flag before distributing.

Beware of the Turing tar pit, in which everything is possible but nothing of interest is easy.

Alan Perlis, *Epigrams of Programming*, #54

Part 1: Introduction

Lambda calculus is a model of computation, much like the Turing machine. As we're about to see, it works in a fundamentally different way, which has a few practical applications we'll discuss at the end of class.

A lambda function starts with a lambda (λ), followed by the names of any inputs used in the expression, followed by the function's output.
For example, $\lambda x.x + 3$ is the function $f(x) = x + 3$ written in lambda notation.

Let's disect $\lambda x.x + 3$ piece by piece:

- "λ" tells us that this is the beginning of an expression.
  λ here doesn't have a special value or definition;
  it's just a symbol that tells us "this is the start of a function."

- "λx" says that the variable x is "bound" to the function (i.e, it is used for input).
  Whenever we see x in the function's output, we'll replace it with the input of the same name.

  This is a lot like normal function notation: In $f(x) = x + 3$, (x) is "bound" to f, and we replace every x we see with our input when evaluating.

- The dot tells us that what follows is the output of this expression.
  This is much like $=$ in our usual function notation:
  The symbols after $=$ in $f(x) = x + 3$ tell us how to compute the output of this function.

Problem 1:
Rewrite the following functions using this notation:
- $f(x) = 7x + 4$
- $f(x) = x^2 + 2x + 1$

To evaluate $\lambda x.x+3$, we need to input a value:

$$(\lambda x.x+3)\ 5$$

This is very similar to the usual way we call functions: we usually write $f(5)$.
Above, we define our function f "in-line" using lambda notation,
and we omit the parentheses around 5 for the sake of simpler notation.

We evaluate this by removing the "λ" prefix and substituting 3 for x wheverever it appears:

$$\begin{tikzcd}[column sep=tiny] (\lambda x. & x+3) & 5 \arrow[ll, bend left] & = 5+3 = 8 \end{tikzcd}$$

Problem 2:
Evaluate the following:

- $(\lambda x.2x+1)\ 4$
- $(\lambda x.x^2+2x+1)\ 3$
- $(\lambda x.(\lambda y.9y)x+3)\ 2$
  Hint: This function has a function inside, but the evaluation process doesn't change.
  Replace all x with 2 and evaluate again.

As we saw above, we denote function application by simply putting functions next to their inputs.
If we want to apply f to 5, we write "$f\ 5$", without any parentheses around the function's argument.

You may have noticed that we've been using arithmetic in the last few problems. This isn't fully correct: addition is not defined in lambda calculus. In fact, nothing is defined: not even numbers!
In lambda calculus, we have only one kind of object: the function. The only action we have is function application, which works by just like the examples above.

Don't worry if this sounds confusing, we'll see a few examples soon.

Definition 3:
The first “pure” functions we’ll define are I and M:

- $I = \lambda x.x$
- $M = \lambda x.xx$

Both I and M take one function (x) as an input.
I does nothing, it just returns x.
M is a bit more interesting: it applies the function x on a copy of itself.

Also, note that I and M don’t have a meaning on their own. They are not formal functions. Rather, they are abbreviations that say “write $\lambda x.x$ whenever you see I.”

Problem 4:
Reduce the following expressions.
Hint: Of course, your final result will be a function.
Functions are the only objects we have!

- $I\ I$
- $M\ I$
- $(I\ I)\ I$
- $\Big(\ \lambda a.(a\ (a\ a))\ \Big)\ I$
- $\Big(\ (\lambda a.(\lambda b.a))\ M\ \Big)\ I$

Example Solution

Solution for $(I\ I)$:
Recall that $I = \lambda x.x$. First, we rewrite the left I to get $(\lambda x.x)\ I$.
Applying this function by replacing x with I, we get I:

$$I\ I = (\lambda x.x) \underset{\curvearrowleft}{\ } I = I$$

So, $I\ I$ reduces to itself. This makes sense, since the identity function doesn’t change its input!

In lambda calculus, functions are left-associative:
$(f\ g\ h)$ means $((f\ g)\ h)$, not $(f\ (g\ h))$
As usual, we use parentheses to group terms if we want to override this order: $(f\ (g\ h)) \neq ((f\ g)\ h)$
In this handout, all types of parentheses ((), [ ], etc) are equivalent.

Problem 5:
Rewrite the following expressions with as few parentheses as possible, without changing their meaning or structure. Remember that lambda calculus is left-associative.

- $(\lambda x.(\lambda y.\lambda(z.((xz)(yz)))))$
- $((ab)(cd))((ef)(gh))$
- $(\lambda x.((\lambda y.(yx))(\lambda v.v)z)u)(\lambda w.w)$

Definition 6: Equivalence
We say two functions are *equivalent* if they differ only by the names of their variables:
$I = \lambda a.a = \lambda b.b = \lambda \heartsuit.\heartsuit = ...$

Note for Instructors

The idea behind this is very similar to the idea behind "equivalent groups" in group theory: we do not care which symbols a certain group or function uses, we care about their *structure*.

If we have two groups with different elements with the same multiplication table, we look at them as identical groups. The same is true of lambda functions: two lambda functions with different variable names that behave in the same way are identical.

Definition 7:
Let $K = \lambda a.(\lambda b.a)$. We'll call K the "constant function function."

Problem 8:
That's not a typo. Why does this name make sense?
Hint: What is $K\ x$?

Solution

$Kx = \lambda a.x$, which is a constant function that always outputs x.
Given an argument, K returns a constant function with that value.

Problem 9:
Show that associativity matters by evaluating $\big((M\ K)\ I\big)$ and $\big(M\ (K\ I)\big)$.
What would $M\ K\ I$ reduce to?

Solution

$\big((M\ K)\ I\big) = (K\ K)\ I = (\lambda a.K)\ I = K$
$\big(M\ (K\ I)\big) = M\ (\lambda a.I) = (\lambda a.I)(\lambda a.I) = I$

Currying:
In lambda calculus, functions are only allowed to take one argument.
If we want multivariable functions, we'll have to emulate them through *currying*![1]

The idea behind currying is fairly simple: we make functions that return functions.
We've already seen this on the previous page: K takes an input x and uses it to construct a constant function. You can think of K as a "factory" that constructs functions using the input we provide.

Problem 10:
Let $C = \lambda f.\Big[\lambda g.\Big(\lambda x.[\ g(f(x))\ ]\Big)\Big]$. For now, we'll call it the "composer."

Note that C has three "layers" of curry: it makes a function (λg) that makes another function (λx).
If we look closely, we'll find that C pretends to take three arguments.

What does C do? Evaluate $(C\ a\ b\ x)$ for arbitary expressions a, b, and x.
Hint: Place parentheses first. Remember, function application is left-associative.

Problem 11:
Using the definition of C above, evaluate $C\ M\ I\ \star$
Then, evaluate $C\ I\ M\ I$
Note: $\star$ represents an arbitrary expression. Treat it like an unknown variable.

As we saw above, currying allows us to create multivariable functions by nesting single-variable functions. You may have notice that curried expressions can get very long. We'll use a bit of shorthand to make them more palatable: If we have an expression with repeated function definitions, we'll combine their arguments under one λ.

For example, $A = \lambda f.[\lambda a.f(f(a))]$ will become $A = \lambda fa.f(f(a))$

Problem 12:
Rewrite $C = \lambda f.\lambda g.\lambda x.(g(f(x)))$ from Problem 10 using this shorthand.

Remember that this is only notation. **Curried functions are not multivariable functions, they are simply shorthand!** Any function presented with this notation must still be evaluated one variable at a time, just like an un-curried function. Substituting all curried variables at once will cause errors.

[1] After Haskell Brooks Curry[2], a logician that contributed to the theory of functional computation.
[2] There are three programming languages named after him: Haskell, Brook, and Curry.
Two of these are functional, and one is an oddball GPU language last released in 2007.

Problem 13:
Let $Q = \lambda abc.b$. Reduce $(Q\ a\ c\ b)$.
Hint: You may want to rename a few variables.
The a, b, c in Q are different than the a, b, c in the expression!

Solution

I'll rewrite $(Q\ a\ c\ b)$ as $(Q\ a_1\ c_1\ b_1)$:

$$\begin{aligned} Q = (\lambda abc.b) &= (\lambda a.\lambda b.\lambda c.b) \\ (\lambda a.\lambda b.\lambda c.b)\ a_1 &= (\lambda b.\lambda c.b) \\ (\lambda b.\lambda c.b)\ c_1 &= (\lambda c.c_1) \\ (\lambda c.c_1)\ b_1 &= c_1 \end{aligned}$$

Problem 14:
Reduce $((\lambda a.a)\ \lambda bc.b)\ d\ \lambda eg.g$

Solution

$((\lambda a.a)\ \lambda bc.b)\ d\ \lambda eg.g$
$= (\lambda bc.b)\ d\ \lambda eg.g$
$= (\lambda c.d)\ \lambda eg.g$
$= d$

Part 2: Combinators

Definition 15:
A *free variable* in a λ-expression is a variable that isn't bound to any input.
For example, b is a free variable in $(\lambda a.a)\ b$.

Definition 16: Combinators
A *combinator* is a lambda expression with no free variables.

Notable combinators are often named after birds.[3] We've already met a few:
The *Idiot*, $I = \lambda a.a$
The *Mockingbird*, $M = \lambda f.ff$
The *Cardinal*, $C = \lambda fgx.(\ f(g(x))\ )$ The *Kestrel*, $K = \lambda ab.a$

Problem 17:
If we give the Kestrel two arguments, it does something interesting:
It selects the first and rejects the second.
Convince yourself of this fact by evaluating $(K \heartsuit \star)$.

Problem 18:
Modify the Kestrel so that it selects its **second** argument and rejects the first.

Solution

$\lambda ab.b$.

Problem 19:
We'll call the combinator from Problem 18 the *Kite*, KI.
Show that we can also obtain the kite by evaluating $(K\ I)$.

Part 3: Boolean Algebra

The Kestrel selects its first argument, and the Kite selects its second.
Maybe we can somehow put this "choosing" behavior to work...

Let $T = K \quad = \lambda ab.a$
Let $F = KI = \lambda ab.b$

Problem 20:
Write a function NOT so that (NOT T) $= F$ and (NOT F) $= T$.
Hint: What is $(T \heartsuit \star)$? How about $(F \heartsuit \star)$?

Solution

$\text{NOT} = \lambda a.(a\ F\ T)$

Problem 21:
How would "if" statements work in this model of boolean logic?
Say we have a boolean B and two expressions E_T and E_F. Can we write a function that evaluates to E_T if B is true, and to E_F otherwise?

Problem 22:
Write functions AND, OR, and XOR that satisfy the following table.

A	B	(AND A B)	(OR A B)	(XOR A B)
F	F	F	F	F
F	T	F	T	T
T	F	F	T	T
T	T	T	T	F

Solution

There's more than one way to do this, of course.

$$\begin{aligned}\text{AND} &= \lambda ab.(a\ b\ F) = \lambda ab.aba \\ \text{OR} &= \lambda ab.(a\ T\ b) = \lambda ab.aab \\ \text{XOR} &= \lambda ab.(a\ (\text{NOT}\ b)\ b)\end{aligned}$$

Another clever solution is OR $= \lambda ab.(M\ a\ b)$

Problem 23:
To complete our boolean algebra, construct the boolean equality check EQ.
What inputs should it take? What outputs should it produce?

Solution

EQ $= \lambda ab.[a\ (bTF)\ (bFT)] = \lambda ab.[a\ b\ (\text{NOT}\ b)]$

EQ $= \lambda ab.[\text{NOT}\ (\text{XOR}\ a\ b)]$