CSCI3390-Lecture 12: λ-Calculus

October 23, 2018

λ-calculus is historically the first model of universal computation to be studied. Where Turing machines and counter machines can be viewed as abstractions of a processor executing machine code, and partial recursive functions as abstractions of programs in procedural languages like Python, λ-calculus is an idealized functional programming language, and is in fact the basis for all such languages.

I am rather new to λ-calculus, after having spent many years teaching theory of computing while remaining blissfully ignorant about it. (It's the programming language types, more than the theory of computing people, who are particularly enamored of λ-calculus.) I found the mini-version of λ-calculus built into Python, in spite of its many limitations, to be very helpful in learning this stuff.

1 What is a computation?

A computation takes an *input* and gives you an *output* as the result of applying some *rule*. We can write rules as expressions like

$$\lambda x.x^2,$$

which means, 'substitute the input for x in the piece after the dot, and you get the output'. We can apply this rule to various inputs, to compute outputs, and write, for instance,

$$(\lambda x.x^2)3 = 9.$$

Python incorporates this notation in something called *anonymous functions* where you can write

```
lambda x: x*x
```

and carry on this kind of dialogue in the Python shell:

```
>>> (lambda x:x*x)(3)
9
```

Python is very nice for this purpose, because it lets you apply functions not just to numbers, but also to other functions, which as we will see is very important in what we are doing.

Writing $\lambda x.x^2$ for the squaring function is meant to give you an idea of how the notation is used, but is a bit dishonest for a first example, because in fact in this world we are creating, we don't have at our immediate disposal numbers, or addition or multiplication, or strings, or *anything*. All we have are rules.

Furthermore, all our rules take only one input. Python's `lambda` notation lets you define anonymous functions of several variables and apply them to pairs, as in

```
>>> (lambda x,y:x+y)(3,4)
7
```

but real λ-calculus does not have functions of several variables.It does let you write things (once we have numbers and addition) like

$$\lambda x.\lambda y.x + y,$$

which you should parse as

$$\lambda x.(\lambda y.x + y),$$

and, if you apply this to 3, and then to 4, you get

$$\begin{aligned}((\lambda x.(\lambda y.x + y))3)4 &= (\lambda y.(3 + y))4 \\ &= 3 + 4 \\ &= 7\end{aligned}$$

and that *also* works in Python:

```
>>> (lambda x:(lambda y:x+y))(3)(4)
7
```

But that is not the same thing as the Python expression we saw earlier. What we are doing is faking a function of two-variables by composing functions of single variables. In effect, we are exploiting the isomorphism between

$$Z^{X\times Y},$$

the set of all functions from the Cartesian product $X \times Y$ into Z and

$$(Z^Y)^X,$$

the set of functions from X into the set Z^Y, which is itself a set of functions, from Y into Z. Here $\lambda x.(\lambda y.x + y)3$ is itself a function, namely $\lambda y : 3 + y$, which adds 3 to its input. We do in fact write

$$\lambda xy.x + y$$

as an abbreviation for $\lambda x.\lambda y.x + y$, but this is not 'really' a function of two variables,. In λ-calculus, all we have is these one-variable computational rules, and we have to fake everything else.

2 Building λ-expressions

If all we have are rules for turning inputs into outputs, what do we apply these rules *to*? What do we use as inputs? what do we get as outputs? The answer is, other rules. Here, more precisely, is how we build the expressions of λ-calculus. Variable symbols like x and y and the like are expressions. These really don't represent specific rules, but you can substitute rules for these variables in larger contexts, just as x in an algebraic expression is not a number, but a placeholder into which you can drop a number.

Given a variable, like x, and an expression M, then we can form the expression

$$\lambda x.M,$$

Furthermore, given two expressions M and N, we form the expression MN which means, 'apply the rule represented by M to N'. In ordinary function notation we would write this as $M(N)$.

That's it. All λ-expressions are built in this manner, by starting with variables, shoving a $\lambda x.$ in front of an expression, and juxtaposing two expressions. We should also introduce parentheses to disambiguate: $(MN)J$ and $M(NJ)$ are not the same thing: The first means, 'apply rule M to N (giving a rule) and apply the resulting rule to J', whereas the second means, 'apply rule M to the result of applying rule N to J.' These rule applications are not quite the same thing as ordinary function composition, which is associative. In fact we do allow you to write MNJ, but the interpretation of this is $(MN)J$. If we want to say $M(NJ)$, we

have to write the parentheses. (In Python, we would write $M(N)(J)$ for MNJ, and $M(N(J))$ for $M(NJ)$.)

As we've already mentioned, we allow $\lambda xy.M$ as an abbreviation for $\lambda x.(\lambda y.M)$, and similarly for three or more variables.

3 Examples–Reduction and Normal Form

x is a λ-expression, and so are xx and xy, but as mentioned above, these are like the variables in normal algebraic expressions. Such expressions evaluate to numbers when numbers are substituted for the variables, and these λ-expressions evaluate to rules when rules are substituted for the variables.

$\lambda x.x$ is a λ-expression, and so is $\lambda y.x$. To apply the first of these to an argument, we simply substitute the argument for every occurrence of x in...x, so we get the input argument back out as output. In other words, for all expressions α,

$$(\lambda x.x)\alpha = \alpha.$$

So $\lambda x.x$ is the identity function. Now the only thing we can apply our rules to is other λ-expressions, so we have, in particular

$$(\lambda x.x)(\lambda x.x) = \lambda x.x.$$

Note how λ-calculus makes application of functions to functions a very simple affair. The whole programs-as-data concept, which requires some gymnastics (*e.g.,* encoding functions as numbers, or machines as strings) when working with recursive functions or Turing machines, is simply built in to the fabric of λ-calculus. We can mimic these calculations in Python, which lets you apply functions to functions.

```
>>> I=lambda x:x
>>> I(3)
3
>>> J=I(I)
>>> J(3)
3
```

We will henceforward use **I** to denote $\lambda x.x$. What about $\lambda y.x$? If we apply this to any input α, it ignores the input and just gives us x:

$$(\lambda y.x)\alpha = x.$$

The result is not a rule but just a variable. Python complains about this:

```
>>> U=lambda y:x
>>> U(3)

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    I(3)
  File "<pyshell#8>", line 1, in <lambda>
    I=lambda x:y
NameError: global name 'y' is not defined
```

The distinction is this: expressions like x, and xy, and $\lambda x.y$ contain *free variables*. The symbols λ *binds* a variable, much as operators like

$$lim_{x\to 0}$$

and

$$\int_0^4 \cdot dx$$

bind variables in what you usually call 'calculus'. So for example

$$\lim_{x\to 0} \frac{\sin x}{x}$$

and

$$\int_0^4 x^3 dx$$

do not have free variables—you cannot substitute anything for the variable x in these expressions. However, the occurence of the variable y in

$$\lim_{x\to 0} \frac{\sin(x+y)}{x+y},$$

and in

$$\int_0^4 x^3 y^2 dx$$

and in

$$\lambda x.y$$

are free. We can substitute numbers (in the first two cases) or rules (in the second) for them to get a number or a rule. Or, we can bind them with another variable-binding operation.

One other point that is important to make: The expressions

$$\int_0^4 x^3 dx, \int_0^4 y^3 dy$$

are *the same thing*. I mean, they are exactly the same integral—the choice of the names x or y for the variable in integration is completely arbitrary, and we perform exactly the same operations when we evaluate them. For the same reason, $\lambda x.x$ is the same thing as $\lambda u.u$ and $\lambda y.y$. They describe the identical rule, computed in the identical manner. This will be important later on.

Here is another example:

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

which we abbreviate

$$\lambda xy.x.$$

We have bound the free variable x in $\lambda y.x$ with another λ. Let's see what happens when we apply it to our earlier rule $\mathbf{I}$. Observe how we rename variables, as described above, to make the computation more readable.

$$\begin{aligned}(\lambda xy.x)\mathbf{I} &= (\lambda xy.x)(\lambda u.u) \\ &= (\lambda x.(\lambda y.x))(\lambda u.u) \\ &= \lambda y.(\lambda u.u) \\ &= \lambda yu.u\end{aligned}$$

The original expression was simplified by applying the basic operation in all these rules: $(\lambda z.M)N$ is *reduced* by substituting N for every free occurrence of z in M. That is what we did in moving from the second to the third line above. Computation in λ-caclulus consists of repeated application of this reduction procedure until we arrive at an expression which contains no more occurrences of subexpressions like $(\lambda z.M)N$. Such terminal expressions are said to be in *normal form*.

We can use our principle of renaming variables and write the result of the above calculation as

$$(\lambda xy.x)\mathbf{I} = \lambda xy.y.$$

4 Booleans

Now we're going to take another important step. We gave the name $\mathbf{I}$ to the expression $\lambda x.x$ because it computes the identity function. We will give the name $\mathbf{T}$ to $\lambda xy.x$ and $\mathbf{F}$ to $\lambda xy.y$, because...

Because. Remember, λ-calculus has nothing except rules. No numbers, no booleans, no sets. If we want to have booleans, we have to fake them, and the faking is to a large degree arbitrary. There is nothing 'true' about the λ-expression $\mathbf{T}$, and nothing 'false' about $\mathbf{F}$. We literally could have faked our booleans with *any* two distinct λ-expressions. These just happen to be convenient choices, because they make the stuff you usually do with booleans easy to compute.

So we can write our calculation above as

$$\mathbf{TI} = \mathbf{F}.$$

We already know

$$\mathbf{IT} = \mathbf{T}, \mathbf{IF} = \mathbf{F}.$$

Let's try a few others:

$$\begin{aligned} \mathbf{FI} &= (\lambda xy.y)(\lambda u.u) \\ &= (\lambda x.(\lambda y.y))\lambda u.u \\ &= \lambda y.y \\ &= \mathbf{I} \end{aligned}$$

Do you see what happened? $\mathbf{F}$ *ignores* whatever its input is and gives $\mathbf{I}$ as a value. In other words, for *any* expression α,

$$\mathbf{F}\alpha = \mathbf{I}.$$

Also, we can generalize the previous calculation to see that for all α,

$$\mathbf{T}\alpha = \lambda x.\alpha^{1}$$

From the above rules, we get

$$\mathbf{TTT} = (\mathbf{TT})\mathbf{T} = (\lambda x.\mathbf{T})\mathbf{T} = \mathbf{T},$$

$$\mathbf{TFT} = (\mathbf{TF})\mathbf{T} = (\lambda x.\mathbf{F})\mathbf{T} = \mathbf{F},$$

$$\mathbf{FTF} = (\mathbf{FT})\mathbf{F} = \mathbf{IF} = \mathbf{F},$$

$$\mathbf{FFF} = (\mathbf{FF})\mathbf{F} = \mathbf{IF} = \mathbf{F}.$$

[1] But be careful here: This is supposed to denote the 'constant function' that returns α on any input, so it is essential that α not contain a free occurrence of x! You must first rename the variables.

In other words, if we substiute $\mathbf{T}, \mathbf{F}$ in any manner for the variables u, v in the expression uvu we will get the result $\mathbf{T}$ if both u and v are $\mathbf{T}$, and $\mathbf{F}$ in all other cases. So

$$\lambda uv.uvu$$

works like the *and* function.

Here is the implementation of these expressions in Python:

```
>>> true = lambda x: lambda y: x
>>> false = lambda x:lambda y: y
>>> AND = lambda x: lambda y: x(y)(x)
```

Note that we use lower and upper-case to avoid conflicting with Python reserved words `True`,`False`,`and`.

We'd like to verify that AND as defined above has the desired properties. But we can't do this directly:

```
>>> AND(true)(false)
<function <lambda> at 0x10d00ab70>
```

Python does not carry out the reduction to normal form, and instead just tells us that `AND(true)(false)` is a function. So in order to see that it is doing the right thing, we need to apply it to concrete objects like Python booleans or ints that will actually display a value. If the result we're looking for is supposed to be a boolean, then we can coax out the correct value by applying the function to the Python boolean `True`, and then to `False`.

```
>>> AND(true)(false)
<function <lambda> at 0x10d00ab70>
>>> AND(true)(true)(True)(False)
True
>>> AND(false)(true)(True)(False)
False
>>> AND(false)(false)(True)(False)
False
```

The reason this works is that $\mathbf{T}$ applied to two arguments as $\mathbf{T}xy = \mathbf{T}(x)(y)$ gives the first argument x as a result, while $\mathbf{F}$ gives y. So the calculation above tells us that AND is doing the right thing.

Try to build OR this way.

What about *not*? This has to turn $\mathbf{T}$ into $\mathbf{F}$, and vice-versa. Now for any expression α and β, we have

$$\mathbf{T}\alpha\beta = \alpha = \mathbf{F}\beta\alpha.$$

So defining *not* as

$$\lambda zxy.zyx$$

does the trick. Just to check this:

$$\begin{aligned}(\lambda zxy.zyx)\mathbf{T} &= (\lambda z.(\lambda xy.zyx))\mathbf{T} \\ &= \lambda xy.\mathbf{T}yx \\ &= \lambda xy.y \\ &= \mathbf{F}.\end{aligned}$$

The reduction when $\mathbf{F}$ is the argument of course works exactly the same way. In Python this looks like

```
>>> NOT=lambda z:lambda x: lambda y:z(y)(x)
>>> NOT(true)(True)(False)
False
>>> NOT(false)(True)(False)
True
```

Before we leave these elementary examples, let's look at what happens with another rule: $\lambda x.xx$. If we apply this to itself, we get

$$\begin{aligned}(\lambda x.xx)(\lambda x.xx) &= (\lambda x.xx)(\lambda y.yy) \\ &= (\lambda y.yy)(\lambda y.yy) \\ &= (\lambda x.xx)(\lambda x.xx)\end{aligned}$$

Hmm... If you try this with Python's `lambda` expressions, you would write

```
>>> (lambda x:x(x))(lambda x:x(x))
```

and get a nasty result! This expression has no normal form—no matter how often we perform a reduction, we get exactly the same thing back and have the same reduction to perform. This is an infinite loop—the λ-calculus version of a program that runs forever.

5 Numbers

There are a number of different ways of faking natural numbers. The oldest of these methods goes back to Church. Essentially it is this: We call these Church numerals $\bar{0}, \bar{1}, \bar{2}$, *etc.* If f is any function, then

$$\bar{n}f = \underbrace{f \circ \cdots \circ f}_{n \text{ times}}.$$

This translates into λ-expressions as follows:

$$\bar{0} = \lambda fx.x = \mathbf{F}.$$

$$\bar{1} = \lambda fx.fx$$

$$\bar{2} = \lambda fx.f(fx)$$

etc.

By means of these we can define the successor function, since we would have to have

$$\overline{n+1}f = f \circ (\bar{n}f),$$

so

$$\mathbf{succ} = \lambda nfx.f(nfx).$$

(Keep in mind that in the right-hand side above, n is just a letter, and any other variable name will do as well.)

We will use Python lambda-expressions to define some of the numbers $\bar{n}$. To test them we will apply them to the increment function and 0, so that applying $\bar{n}$ to (the normal Python int) 0 will give (the normal Python int) n.

```
>>> zero=false
>>> one = lambda f:lambda x:f(x)
>>> two=lambda f:lambda x:f(f(x))
>>> three = lambda f:lambda x:f(f(f(x)))
>>> inc=lambda x: x+1
>>> zero(inc)(0)
0
>>> one(inc)(0)
1
>>> two(inc)(0)
```

```
2
>>> three(inc)(0)
3
>>> succ=lambda n:lambda f:lambda x:f((n(f))(x))
>>> succ(two)(inc)(0)
3
```

We also get addition: We must have

$$(\overline{n+m})f = (\bar{n}f) \circ (\bar{m}f)$$

so that we define

$$\mathbf{add} = \lambda nmfx.(nf)(mfx).$$

In other words, given f and x, **add** $\bar{n}\bar{m}$ applies f m times to x, and then applies f n times to the result. So that f is applied $n+m$ times to x, which is what $\overline{n+m}$ does.

Here it is in Python:

```
>>> two=lambda f: lambda x: f(f(x))
>>> three=lambda f:lambda x:f(f(f(x)))
>>> add = lambda n:lambda m: lambda f: lambda x: n(f)(m(f)(x))
>>> inc = lambda x:x+1
>>> add(two)(three)(inc)(0)
5
```

6 Pairs

Just as we have to fake booleans, we have to fake some method of bundling two values into an ordered, pair, and extracting the components of the pair. The following work:

$$\begin{aligned}\mathbf{pair} &= \lambda xyz.zxy\\ \mathbf{first} &= \lambda x.x\mathbf{T}\\ \mathbf{second} &= \lambda x.x\mathbf{F}.\end{aligned}$$

Just to see how this works:

$$\begin{aligned}\mathbf{first}(\mathbf{pair}\alpha\beta) &= (\lambda x.x\mathbf{T})\lambda z.z\alpha\beta\\ &= (\lambda z.z\alpha\beta)\mathbf{T}\\ &= \mathbf{T}\alpha\beta\\ &= \alpha\end{aligned}$$

and the identical calculation shows

$$\textbf{second}(\textbf{pair}\alpha\beta) = \beta.$$

We can do k-tuples and projections from k-tuples onto their components by the same trick.

Here is the calculation in Python:

```
>>> pair = lambda x: lambda y:lambda z: (z(x))(y)
>>> true = lambda x:lambda y:x
>>> false=lambda x: lambda y:y
>>> first=lambda x: x(true)
>>> second=lambda x: x(false)
>>> first(pair('cat')('dog'))
'cat'
>>> second(pair('cat')('dog'))
'dog'
```

7 Perspective

This was just a little taste of λ-calculus. In the exercises, you will have a chance to see how to use pairing and Church numerals to implement both the predecessor function and primitive recursion (the `for` of Tiny Python).

If we are able to implement recursive function calls as well, then we obtain all the partial functions that can be computed by Turing machines, or equivalently, counter machines, or partial recursive functions. Universal computation is everywhere, in numerous disguises.