CS98-52 Lecture 2

Wednesday, September 11th, 2019

Lambda Calculus

Church-Turing Thesis

- We haven't even covered all the features of Python in 61A, like `class` and `for` statements. But you should ask: do we even need all the features of a language to write a program? Could we get rid of any single feature and still do the same thing?
- A function on the natural numbers is computable by a human following an algorithm, ignoring resource limitations, if and only if it is computable by a Turing machine.
- The Turing machine is a simple machine with a programming language that has no functions.
- Lambda calculus is a function that only has call expressions and lambda expressions.
- Comparing these two, in doing exactly the same thing despite one not having functions and the other having nothing but.
- One of the ideas was to encode numbers as functions, as part of "just for fun" in Homewokr 2, which will be covered in CS98-52 Lecture 3.
- You can also invent functions to represent True and False.
- The goal is not to build a practical programming language, but the simplest one possible.

Representation

- What do you do about all the other stuff you need?

Functions Can Represent Boolean Values

- If all we have to work with are functions and call expressions, is there any way to represent other primitive values?

```
t = lambda a: lambda b: a
f = lambda a: lambda b: b
```

- There is no real explanation for why it works this way. True and False, generally speaking, are just a choice between two things `a` and `b`, and this is the simplest way we can represent it, even if rather arbitrary.
- t and f are set up so that if you take any representation of True and False and pass them in in the right order, they will represent True and False.

```
def py_pred(p):
    """Takes in either t or f, and converts them into the standard Python True and False
    return p(True)(False)
```

- We can also build a not statenment:

```
def f_not(p):
    """Define Not.

    >>> py_pred(f_not(t))
    False
    >>> py_pred(f_not(f))
    True
    """
    return lambda a: lambda b: p(b)(a)
```

- We could also define this without lambdas in the definition:

```
def f_not(p):
    return p(f)(t)
```

Exercise

1. Write function `f_and` .

```
def f_and(p,q):
    return p(q)(f)
```

> This states that `p` will be evaluated first. If true, it will evaluate `q` , which if True, returns True. Otherwise, False.

2. Write function `f_or` .

```
def f_or(p,q):
    return p(t)(q)
```

> This states that `p` will be evaluated first. If true, then it automatically returns True. Otherwise, the value depends on whether `q` is True or False.

Environment diagrams aren't good here because it's unreadable. You can evaluate this using the substitution way of evaluation, which would be bad in normal Python because values change. In lambda calculus, names don't change, so this is good.

```
(lambda x: x + 2)(4+5)
```

is evaluated with:

```
4+5 + 2
```

directly substituting `x` with `4+5` .

> **Iteration Using Functions**
> Turns out you can write a function imitating a while statement using recursion!

If With Lambda calculus

How would we represent an if statement?

```
def f_if(p,a,b):
    return p(a)(b)
```

> **Exclusive Or**
> Exclusive or is an or statement where only one value can be True for the value returned to be True. It can be implemented by:
>
> ```
> def ex_or(p,q):
>     return p(f_not(q),q)
> ```

Lambda Calculus Notation

This is why lambda calculus is so concise to write:

- Single letters, such as `x`, instead of long names.
- Functions: Instead of lambda x: x, we write x.x
  - Also, no commas, so the first statement is: ab.a
- Assignment: write as `var f = ...`
- Application: Instead of `f(x)`, write as `(f x)`.
  - Instead of `f(x,y)` and `f(x)(y)`, write `(f x y)`