Lecture #3: Lambda Calculus

Simplifying Python

- Python is full of features. Most are there to make programming concise and clear. Some are there for speed.
- But if we can put up with some hardship, the same *computations* can be carried out with much less.

Removing Assignment to Variables (I)

```
def dist(x1, y1, x2, y2):
    dx = x1 - x2
    dy = y1 - y2
    return sqrt(dx * dx + dy * dy)
print(dist(3, 4, 9, 10))
```

Can be rewritten as

```
(lambda dist:
    print(dist(3, 4, 9, 10)))\
 (lambda x1, y1, x2, y2: \
   (lambda dx, dy: sqrt(dx*dx + dy*dy))(x1-x2, y1-y2))
```

What About Recursion?

- In Python, we write

```
def fact(n):
    return 1 if n == 0 else n * fact(n-1)
```

  becomes

```
fact = lambda n: 1 if n == 0 else n * fact(n-1)
```

  but this is an assignment.

- The trick is to contrive to "feed `fact` into itself" by means of lambda.
- Start with

```
lambda fact: lambda n: 1 if n == 0 else n * fact(n-1)
```

- But `fact` is a parameter here (no value yet given). How can we feed it into itself?

Recursion (II)

- We have the function value

```
lambda fact: lambda n: 1 if n == 0 else n * fact(n-1)
```

  and want the result of substituting this same function to `fact` in the body. How?

Recursion (II)

- We have the function value

```
lambda fact: lambda n: 1 if n == 0 else n * fact(n-1)
```

  and want the result of substituting this same function to `fact` in the body. How?

- Another lambda!

```
>>> (lambda f: f(f))(lambda fact: lambda n: 1 if n == 0 else n * fact(fact)(n-1)
24
```

Loops

- We've seen examples in class of getting rid of loops.

```
def fib(n):
    if n == 0: return 0

    f0, f1 = 0, 1
    while n > 1:
        f0, f1, n = f1, f0 + f1, n-1
    return f1
```

- Rewrite as a recursive function with no loop:

Loops

- We've seen examples in class of getting rid of loops.

```
def fib(n):
    if n == 0: return 0

    f0, f1 = 0, 1
    while n > 1:
        f0, f1, n = f1, f0 + f1, n-1
    return f1
```

- Rewrite as a recursive function with no loop:

```
def fib(n):
    def loop(f0, f1, n):
        if n > 1:
            return loop(f1, f0+f1, n-1)
        else:
            return f1
    if n == 0: return 0
    return loop(0, 1, n)
```

- And we've already seen how to turn all of this into a lambda!

Multiple Arguments

- In lab 2, you saw that only one argument to a function is all you need.

```
(lambda x, y: something(y, x))(a, b)
    # Can be written
(lambda x: lambda y: something(y, x))(a)(b)
```

- This rewrite is called *currying*, after Haskell Curry (who didn't invent it).

Getting Rid of Integers?! Church Numerals

- Alonzo Church was a famous logician and mathematician, responsible for, among other things, the lambda calculus (coming) and the Church-Turing Thesis.
- The Church-Turing thesis is that any function on the natural numbers that is computable by some algorithm is computable using a Turing Machine.
- The lambda calculus is a pure calculus of functions, without numbers, strings, booleans, conditionals, etc.
- Yet it can represent the non-negative integers, using an encoding known as *Church numerals*.

The Representation

- We *define* (or if you prefer, *encode*) zero and the successor (+1) operator as follows:

```
zero =      lambda f: lambda x: x
successor = lambda n: lambda f: lambda x: f(n(f)(x))
```

- So 1 is successor(zero) and 2 is successor(successor(zero)).
- What is 1 (as a function that does not use successor)?
- What is 2 (as a function that does not use successor)?

Operations

- How does one turn a Church numeral into the integer it represents?
- How does one implement addition?

```
def add_church(a, b):

    return _____
```

- Multiplication?

```
def mul_church(a, b):

    return _____
```

Answers

- The Church numeral representing the natural number n is a function that takes two parameters, say f and x, and returns the result of applying f to x n times.
- So:

```
def add_church(a, b):
    return a(successor)(b)
    # or
    return a(lambda n: lambda f: lambda x: f(n(f)(x)))(b)
```

- Currying this and using lambda notation gives

```
add_church = lambda a: lambda b: a(lambda n: lambda f: lambda x: f(n(f)(x)))(b)
```

- Which allows us to define multiplication:

```
def mul_church(a, b):
    return a(add_church(b))(zero)

# or in curried form:

def mul_church = lambda a: lambda b: a(add_church(b))(zero)
```

A Radical Purification: The Pure (Untyped) Lambda Calculus

- The discussion so far suggests that we can get rid of a lot of Python, rewriting it into a small number of constructs.
- Taken to the extreme, we get something called the *lambda calculus,* a model of computation developed by Church to answer various questions about the foundations of computation.
- Consider a language in which there are *only* the following terms:
  - **Symbols:** `x`, `y`, etc. These are *not* necessarily identifiers (of parameters, etc.), athough they often act as such.
  - **Applications:** $(E_1\ E_2)$ where E_1 and E_2 are terms.
  - **Abstractions (lambda terms):** $(\lambda x.\ E)$, where x is a symbol and E is a term.
- Because all our lambda terms have single arguments, we generally abbreviate $(((A\ B)\ C)\ D)$, for example, as $A\ B\ C\ D$.
- Likewise, we leave off the parenteses around lambda terms if they are followed by another parenthesis or the end of an expression: $f\ (\lambda x.\ E)$ becomes $f\ \lambda x.\ E$.

The Substitution Model

- Before we had environments, there was the *substitution model* for calling functions (see 61A Lecture 2).
- For example, to evaluate

```
(lambda x: lambda y: x*y)(2+3)(3+4)
==> (lambda x: lambda y: x*y)(5)(3+4)
==> (lambda x: lambda y: x*y)(5)(7)
==> (lambda y: 5*y)(7)
==> 5*7
==> 35
```

- Since we've gotten rid of mutable state, the substitution model now works fine as it is.

Applicative Order Vs. Normal Order

- However, there is an essential difference between evaluation in the pure lambda calculus and in Python.
- We've said in lecture that to evaluate $E_1(E_2)$, we
  - Evaluate E_1 and E_2 to get values v_1 and v_2.
  - Then substitute v_2 for the parameter of f (which must be a function) in the body of f.
  - And then evaluate the resulting body.
- This is called *applicative order* evaluation.

Normal Order

- In the pure lambda calculus, however, we deal with *terms,* not values, and instead of applicative-order calls, we instead do *beta reductions* (substitutions) in what is called *normal order*. For a term T:
  - If T is a symbol, do nothing.
  - If T is a lambda term, evaluate its body.
  - If T is an application $(A\ B)$,
    - If A is a lambda term, $\lambda x.\ C$, replace T with the result of substituting the term B for all bound instances of x in C.
    - Otherwise, evaluate A.
    - Otherwise (if the above changed nothing), evaluate B.
  - If none of these change anything, no further evaluation is possible; we have a *normal form.*
- By repeating this process until we have a normal form, we have a model of computation.

Church Numerals Revisited

- Converting to our new notation:
  - ZERO is $\lambda f.\ \lambda x.\ x$
  - SUCC (+1) is $\lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$.

Getting Rid of Conditionals

- Because of the way normal-order evaluation works, we can do away with conditional statements.
  - TRUE is $\lambda x.\ \lambda y.\ x$.
  - FALSE is $\lambda x.\ \lambda y.\ y$.
  - IFTHENELSE is $\lambda p.\ \lambda a.\ \lambda b.\ p\ a\ b$.
  - AND is $\lambda p.\ \lambda q.\ p\ q\ p$.
  - OR is $\lambda p.\ \lambda q.\ p\ p\ q$.
  - NOT is $\lambda p.\ p\ FALSE\ TRUE$.
  - ISZERO is $\lambda n.\ n\ (\lambda x.\ FALSE)\ TRUE$.
- Because of normal-order evaluation, computations that blow up, such as LOOP—$(\lambda x.\ x\ x)(\lambda x.\ x\ x)$—need not cause IFTHENELSE to fail. What happens with

```
IFTHENELSE FALSE TRUE LOOP ?
```

Getting Rid of Parameters??!

- In fact, if we define a few primitive terms, we can get rid of all other uses of parameters (all other lambda terms):
  - I is $\lambda x.\ x$.
  - K is $\lambda x.\ \lambda y.\ x$.
  - S is $\lambda x.\ \lambda y.\ \lambda z.\ xz(yz)$.

  I, K, and S are called *combinators*.
- For example, in place of $\lambda x.\ \lambda y.\ yx$, we can write

```
S (K (S I)) (S (K K) I)
```

(Try it out, for example: converting to Python notation, try

```
I = lambda x: x
K = lambda x: lambda y: x
S = lambda x: lambda y: lambda z: x(z)(y(z))

(lambda x: lambda y: y(x))(-2)(abs)
# and
S(K(S(I)))(S(K(K))(I))(-2)(abs)
```