61A Extra Lecture 3
Announcements

cs61a.org/extra.html
Church-Turing Thesis
The Church-Turing Thesis

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.
Representation
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?

\[
t = \lambda a: \lambda b: a \\
f = \lambda a: \lambda b: b
\]

```python
def py_pred(p):
    return p(True)(False)

def f_not(p):
    """Define Not.\n    >>> py_pred(f_not(t))
    False
    >>> py_pred(f_not(f))
    True
    """
    return \lambda a: \lambda b: p(b)(a)
```

Exercise:

```python
def f_and(p, q):
    """Define And.\n    >>> py_pred(f_and(t, t))
    True
    >>> py_pred(f_and(t, f))
    False
    >>> py_pred(f_and(f, t))
    False
    >>> py_pred(f_and(f, f))
    False
    """
    return \_\_\_\_p(q)(f)\_\_\_\_

def f_or(p, q):
    """Define Or.\n    >>> py_pred(f_or(t, t))
    True
    >>> py_pred(f_or(t, f))
    True
    >>> py_pred(f_or(f, t))
    True
    >>> py_pred(f_or(f, f))
    False
    """
    return \_\_\_\_p(t)(q)\_\_\_\_
```
If all we have to work with are functions and call expressions, is there any way to represent other primitive values?

```python
def zero(s):
    return lambda z: z

def one(s):
    return lambda z: s(z)

def two(s):
    return lambda z: s(s(z))

def successor(n):
    return lambda s: lambda z: s(n(s)(z))

three = successor(two)
```

```python
def add_church(m, n):
    return lambda s: lambda x: m(s)(n(s)(x))

def mul_church(m, n):
    return lambda s: m(n(s))

def pow_church(m, n):
    return n(m)
```

Note: `lambda x: f(x)` is the same as `f`
Lambda Calculus Notation
Lambda Calculus

Variables: single letters, such as $x$

Functions: Instead of 
\[ \text{lambda } x: x \]
write $\lambda x . x$ ; Instead of 
\[ \text{lambda } x, y: x \]
write $\lambda xy . x$

Assignment: Write 
\[ \text{var } f = \ldots \]

Application: Instead of 
\[ f(x) \]
write 
\[ (f x) \]; 
\[ f(x)(y) \] and 
\[ f(x, y) \] are both written 
\[ (f x y) \]

Follow along! [http://chenyang.co/lambda/](http://chenyang.co/lambda/)

To type $\lambda$, just type \\n
\[ \text{var } I = \lambda x . x \]
\[ \text{var } K = \lambda r . \lambda s . r \]

Are (I I) and I the same? Are (K I I) and (K I K) the same?

Are (K I) and I the same? What's ((K K) (K K)) the same as?

Are (K K I) and K the same? Can you construct a 4-argument function by just calling K & I?
**Boolean Values**

**Variables:** single letters, such as `x`

**Functions:** Instead of `lambda x: x`, write `λx.x`; Instead of `lambda x, y: x`, write `λxy.x`

**Assignment:** Write `var f = ...`

**Application:** Instead of `f(x)`, write `(f x); f(x)(y)` and `f(x, y)` are both written `(f x y)`

Follow along! [http://chenyang.co/lambda/](http://chenyang.co/lambda/)

To type `λ`, just type `\`

```plaintext
var T = λab.a
var F = λab.b
```

Define **and**, **or**, and **not**!

Define exclusive or:

- `xor(False, False) -> False`
- `xor(False, True)   -> True`
- `xor(True,  False)  -> True`
- `xor(True,  True)   -> False`