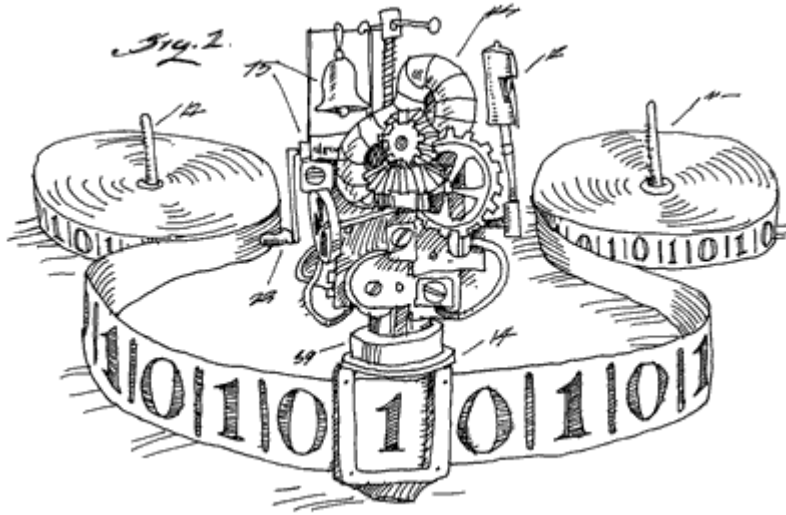




Church-Turing Thesis



Church-Turing Thesis



Motivation: Encoding values with functions

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

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

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

```
Exercise: Implement f_and, f_or
f_and = lambda p1: lambda
p2: p1(p2)(f)
f_or = lambda p1: lambda p2: p1(t)(p2)
p2:
```

A Cleaner Syntax

$\lambda[\text{argument}].[\text{return value}]$

$$f(x) \equiv fx$$

$$(\lambda x.M)N = M[x := N]$$

Basic Functions

$$I = \lambda t.t$$

$$C_r = \lambda s.r$$

$$C_I = \lambda s.(\lambda t.t)$$

$$K = \lambda r.(\lambda s.r)$$

Exercise:

Confirm that $K I$ is equivalent to C_I

Write down C_{C_I} , also write it in terms of K and I

Evaluate $(K K) (K K)$

<http://www.chenyang.co/lambda>

Currying

$$fgh \equiv ((fg)h)$$

$$\lambda x.(\lambda y.xy) \equiv \lambda xy.xy$$

$$\pi_1 = \lambda xy.x$$

$$\pi_2 = \lambda xy.y$$

What does `pi_1` and `pi_2` remind you of?

True and False

```
true  = lambda a: lambda b: a
false = lambda a: lambda b: b
```

$$T \equiv \lambda ab.a$$
$$F \equiv \lambda ab.b$$

```
f_not = lambda p: lambda a:
          lambda b: p(b)(a)
f_and = lambda p1: lambda p2: p1(p2)(false)
f_or  = lambda p1: lambda p2: p1(true)(p2)
```

Exercise:

Translate boolean operators into lambda notation

$$\text{Not} \equiv \lambda p a b.p b a$$
$$\text{And} \equiv \lambda p q.p q F$$
$$\text{Or} \equiv \lambda p q.p T q$$

Write an if-function using lambda notation

Zero, One, Two, ...

$$\begin{aligned}nf \rightarrow f^n &= f \circ f \circ \dots \circ f \\ &= \lambda x.f(f(\dots f(x)\dots))\end{aligned}$$

$$n = \lambda f x.f(f(\dots f(x)\dots))$$

$$\begin{aligned}Sn &= \lambda f x.f^{n+1}x \\ &= \lambda f x.f(f^n x) \\ &= \lambda f x.f(nfx) \\ S &= \lambda n f x.f(nfx)\end{aligned}$$

$$0 \equiv \lambda f x.x$$

$$1 \equiv \lambda f x.fx$$

$$2 \equiv \lambda f x.f(fx)$$

Exercise:

Implement add, mul, power

Suppose we have an operator pred that returns the predecessor of a number, implement subtraction using pred

Implement pred (Challenging)

Recursion!

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

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

```
F(fac) = fac <- fac is the fixed point of F
```

Y computes the fixed point of F: YF is the fixed point

$$F(YF) = YF$$

Fixed Points

$$F(Y F) = Y F$$

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

$$\begin{aligned}\omega' &= (\lambda x.F(xx))(\lambda x.F(xx)) \\ &= F((\lambda x.F(xx))(\lambda x.F(xx))) = F(\omega')\end{aligned}$$

$$Y = \lambda F.(\lambda x.F(xx))(\lambda x.F(xx))$$

Puzzle

What is X? (There are 26 Ls)

Hint: A fixed point combinator has the form $YF=F(YF)$

$$X = (LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL)$$

$L = \lambda abcdefghijklmnopqrstuvwxyzr.r(\text{this is a fixed point combinator})$

$$X = \lambda r.r(Xr)$$

$$XF = F(XF)$$

Conclusion + Acknowledgements

Everything + more:

<http://xuanji.appspot.com/isicp/lambda.html>

Lambda interpreter:

<https://github.com/tarao/LambdaJS>
