INF 212
ANALYSIS OF PROG. LANGS
LAMBDA CALCULUS
IN PYTHON

Instructors: Crista Lopes

Credits: http://matt.might.net/articles/python-church-y-combinator/

Syntax

$M ::= x$	(variable)
$\mid \lambda x.M$	(abstraction)
$\mid MM$	(application)

Nothing else!

- No numbers
- No arithmetic operations
- No loops
- No etc.

Symbolic computation

Syntax refresher

anonymous functions in Python

$\lambda x.M$ ➔ lambda x : M

LN, e.g. $\underbrace{\lambda x.M}_{L}\ \underbrace{y}_{N}$ ➔ (lambda x : M) (y)

A PL from Lambda Calculus

- Bare minimum for a ***modern*** PL:
  - void value
  - multi-argument functions
  - booleans and conditionals
  - numbers and arithmetic
  - pairs and lists
  - recursive functions

Void

- void: a value to be ignored
  - remember we have only functions of 1 argument, so a void value must be a function of 1 argument
- two viable options:
  - identity function

```
VOID = lambda x : x
```

  - error function

```
def VOID(_):
      raise Exception('VOID cannot be called')
```

A PL from Lambda Calculus

- Bare minimum for a ***modern*** PL:
  - void values
  - multi-argument functions
  - booleans and conditionals
  - numbers and arithmetic
  - pairs and lists
  - recursive functions

Multi-argument Functions

- Currying

```
add = lambda x, y: x+y
add(2, 3)
```

```
add = lambda x : lambda y: x+y
add(2)(3)
```

A PL from Lambda Calculus

- Bare minimum for a ***modern*** PL:
  - void values
  - multi-argument functions
  - booleans and conditionals
  - numbers and arithmetic
  - pairs and lists
  - recursive functions

Booleans and Conditionals

- Simple conditionals:
  ```
  if <condition> <t_exp> else <f_exp>
  ```
- How to do this in λ-calculus?

```
IF = lambda c, t, f: c(t)(f)
   = lambda c: lambda t: lambda f: c(t)(f)
```

- TRUE and FALSE

```
TRUE  = lambda t, f: t
      = lambda t: lambda f: t

FALSE = lambda t, f: f
      = lambda t: lambda f: f
```

A PL from Lambda Calculus

- Bare minimum for a ***modern*** PL:
  - void values
  - multi-argument functions
  - booleans and conditionals
  - numbers and arithmetic
  - pairs and lists
  - recursive functions

Numbers as functions (Church)

- Natural numbers as iterated application
  - e.g. 3 = lambda f, x: f(f(f(x))))
- We need ZERO
  - Given a function of 1 argument, return the identity function (it "refuses" to apply the given function :-)

```
ZERO = lambda f: lambda x: x
```

- We need the rest of them

```
ONE   = lambda f: lambda x: f(x)
TWO   = lambda f: lambda x: f(f(x))
THREE = lambda f: lambda x: f(f(f(x)))
etc.
```

Some helpers

- Python number => Church numeral

```
def numeral(n):
    return lambda f: lambda x: x if n==0 else f(numeral(n-1)(f)(x))
```

- Church numeral => Python number

```
npy = lambda c: c(lambda x: x+1)(0)
```

Arithmetic: successor

- Given a Church numeral, n, n+1 should apply f one more time, e.g.

```
SUCC(TWO)(f)(x) = lambda f: lambda x: f(f(f(x)))
```

```
SUCC = lambda n: lambda f: lambda x: f(n(f)(x))
```

Arithmetic: addition, multiplication

- Addition: Given two Church numerals, n and m, apply a function n times, then apply it m times

```
ADD = lambda n: lambda m: lambda f: lambda x: n(f)(m(f)(x))
```

- Multiplication: Given two Church numerals, n and m, apply n exactly m times

```
MUL = lambda n: lambda m: lambda f: lambda x: m(n(f))(x)
```

A PL from Lambda Calculus

- Bare minimum for a ***modern*** PL:
  - ✅ void values
  - ✅ multi-argument functions
  - ✅ booleans and conditionals
  - ✅ numbers and arithmetic
  - pairs and lists
  - recursive functions

Pairs

- Given a and b, we want to create (a, b), a pair.

```
PAIR = lambda a: lambda b: lambda f: f(a)(b)

LEFT  = lambda p: p(lambda a: lambda b: a)
RIGHT = lambda p: p(lambda a: lambda b: b)
```

Lists

- The empty list

```
NIL   = lambda onnil: lambda onlist: onnil(VOID)
```

- Given a head and a tail, we want to construct a list from their concatenation

```
CONS  = (lambda hd: lambda tl:
           lambda onnil: lambda onlist: onlist(hd)(tl))
```

A PL from Lambda Calculus

- Bare minimum for a ***modern*** PL:
  - ✓ void values
  - ✓ multi-argument functions
  - ✓ booleans and conditionals
  - ✓ numbers and arithmetic
  - ✓ pairs and lists
  - recursive functions

Combinators

- Functions that combine functions in interesting ways
  - The call can be made later, and multiple times

Non-termination: U Combinator

- Applies its argument to its argument

```
U = lambda f: f(f)
```

Recursion via the U Combinator

- Example:

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

- We can try passing fact to itself:

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

  - doesn't work. Why? (hint: check the type of n)

- Extra parameter:

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

  - works, but it still has explicit recursion (call to fact)

Recursion via the U Combinator

- Example:

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

- U Combinator to the rescue:

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

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

- Clean version:

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

More elegant: the Y Combinator

- A bit cumbersome to call U(f)
- Another idea: find "fixed point" of the function
  - x such that x = f(x) ← fixed point of f
  - g such that g = f(g) for functionals (functions that take a function as argument)
- Functional for which `fact` is a fixed point

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

  - iff f is `fact`
- Wanted: Y that finds fixed point of our functional:

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

Deriving the Y Combinator

- Main property:
  - Given functional F and function f,

```
Y(F) == f    and   f == F(f)

➔Y(F) == F(f)
➔Y(F) == F(Y(F))

➔Y = lambda F: F(Y(F))
```

  - but this doesn't work. Why? (hint: recursion)

Deriving the Y Combinator

- Observation: for any expression e,

```
e == lambda x: e(x)
```

- Let's expand the call to Y:

```
Y = lambda F: F(Y(F))

Y = lambda F: F(lambda x: Y(F)(x))
```

  - this works! why? (hint: extra abstraction)

```
Y(lambda f: lambda n: 1 if n <= 0 else n*f(n-1))(5)
prints 120
```

Deriving the Y Combinator

- We have:

```
Y = lambda F: F(lambda x: Y(F)(x))
```

- Let's eliminate explicit recursion with U combinator:

```
Y = U(lambda h: lambda F: F(lambda x:U(h)(F)(x)))
```

- Or, inlining U:

```
Y = ((lambda h: lambda F: F(lambda x:h(h)(F)(x)))
     (lambda h: lambda F: F(lambda x:h(h)(F)(x))))
```

Recursion via the Y Combinator

```
Y = ((lambda h: lambda F: F(lambda x:h(h)(F)(x)))
      (lambda h: lambda F: F(lambda x:h(h)(F)(x))))
```

```
Y(lambda f: lambda n: 1 if n <= 0 else n*f(n-1))(5)
prints 120
```

fixed point of this factorial functional

A PL from Lambda Calculus

- Bare minimum for a ***modern*** PL:
  - ✅ void values
  - ✅ multi-argument functions
  - ✅ booleans and conditionals
  - ✅ numbers and arithmetic
  - ✅ pairs and lists
  - ✅ recursive functions

Conclusion

If we have anonymous functions,
we have a Programming Language.