

C

The Lambda Calculus

λ calculus

This is an appendix to upcoming book "[Introduction to Theoretical Computer Science](#)", which is also available online as a Jupyter notebook in the [boazbk/handnotebooks](#) on Github. You can also try the [live binder version](#).

The λ calculus is discussed in [Chapter 7: "Equivalent Models of Computation"](#)

[Click here](#) for the live Binder version. (Service can sometimes be slow.)

This Python notebook provides a way to play with the lambda calculus and evaluate lambda expressions of the form $\lambda var_1(exp_1) \lambda var_2(exp_2) \dots$. If you don't know Python you can safely ignore the Python code and skip below to where we actually talk about the λ calculus itself.

To better fit with python there are two main differences:

- Instead of writing $\lambda var . exp$ we write $\lambda var(exp)$
- Instead of simply concatenating two expressions $exp_1 exp_2$ we use the $*$ operator and write $exp_1 * exp_2$. We can also use exp_1, exp_2 if they are inside a function call or a variable binding parenthesis.
- To reduce an expression exp , use $exp.reduce()$
- Since Python does not allow us to override the default 0 and 1 we use $_0$ for $\lambda x(y(y))$ and $_1$ for $\lambda x(y(x))$.

Python code (can skip at first read)

If you don't know Python feel free to skip ahead to the part where we play with the λ calculus itself.

```
In [1]: # We define an abstract base class Lambdaexp for Lambda expressions
# It has the following subclasses:
# Applicableexp: an expression of the form  $\lambda x.exp$ 
# Combinedexp: an expression of the form  $(exp,exp')$ 
# Boundvar: an expression corresponding to a bounded variable
# Unboundvar: an expression corresponding to a free variable
#
# The main operations in a Lambdaexp are:
# 1. Replace: given  $exp,x$  and  $exp'$ , obtain the expression  $exp[x \rightarrow exp']$ 
# 2. Reduce: continuously evaluate expressions to obtain a simpler form
# 3. Apply: given  $exp,exp'$ , if  $exp$  is applicable then apply it to  $exp'$ , otherwise combine the two
# (we also use the  $*$  operator for it)

import operator ,functools
```

```

class Lambdaexp:
    """Lambda expressions base class"""

    counter = 0
    call_by_name = True # if False then do normal form evaluation.

    def __init__(self):
        self.mykey = {}

    def apply(self, other):
        """Apply expression on an argument"""
        return self*other

    def _reduce(self, maxlevel=100):
        """Reduce expression"""
        return self

    def replace(self, old, new):
        """Replace all occurrences of old with new"""
        raise NotImplemented

    def bounded(self):
        """Set of bounded variables inside expression"""
        return set()

    def asstring(self, m, pretty=False):
        """Represent self as a string mapping bounded variables to particular numbers."""
        raise NotImplemented

    #-----#
    # Ignore this code in first read: Python specific details

    lambdanames = {}
    reducedstrings = {}

    def reduce(self, maxlevel=100):
        if not maxlevel: return self
        #m = {b:b for b in self.bounded()}
        #t = Lambdaexp.reducedstrings.get((self.asstring(m), maxlevel), None)

        #if t: return t
        return self._reduce(maxlevel)
        #k = t.asstring(m)
        #for i in range(maxlevel+1):
        #    Lambdaexp.reducedstrings[(k, i)] = t
        #return t

    def __mul__(self, other):
        """Use * for combining."""
        return Combinedexp(self, other) if other else self

```

```

def __call__(self, *args):
    """Use function call for application"""
    return functools.reduce(operator.mul, args, self)

def _key(self, maxlevel=100):
    #if maxlevel not in self.mykey:
    return self.reduce(maxlevel).__repr__()
    # for i in range(maxlevel+1): self.mykey[i] = s
    # return self.mykey[maxlevel]

def __eq__(self, other): return self._key()==other._key() if isinstance
(other, Lambdaexp) else False
def __hash__(self): return hash(self._key())

def __repr__(self, pretty=False):
    B = sorted(self.bounded())
    m = {}
    for v in B: m[v] = len(m)
    return self.asstring(m, pretty)

def _repr_pretty_(self, p, cycle):
    if cycle: p.text( self._repr())
    p.text( self.reduce().__repr__(True))

def addconst(self, srep):
    """Return either exp.string or replaced with a keyword if it's in
table."""
    if self in Lambdaexp.lambdanames: return blue(Lambdaexp.lambdanam
es[self])
    return srep

#-----#
-----#

```

```

In [2]: #-----#
# Utility functions: print color
def bold(s, justify=0):
    return "\x1b[1m"+s.ljust(justify)+"\x1b[21m"

def underline(s, justify=0):
    return "\x1b[4m"+s.ljust(justify)+"\x1b[24m"

def red(s, justify=0):
    return "\x1b[31m"+s.ljust(justify)+"\x1b[0m"

def green(s, justify=0):
    return "\x1b[32m"+s.ljust(justify)+"\x1b[0m"

```

```
def blue(s,justify=0):
    return "\x1b[34m"+s.ljust(justify)+"\x1b[0m"
#-----#
```

In [3]:

```
class Applicableexp(Lambdaexp):
    """Lambda expression that can be applied"""

    def __init__(self, exp, name):
        Lambdaexp.counter += 1
        self.arg = Lambdaexp.counter
        self.inner = exp.replace(name, Boundvar(self.arg))
        super().__init__()

    def apply(self, other):
        return self.inner.replace(self.arg, other)

    def replace(self, old, new):
        if self.arg==old: self.arg = new.myid
        return Applicableexp(self.inner.replace(old, new), self.arg)

    def bounded(self): return self.inner.bounded()|{self.arg}

    def _reduce(self, maxlevel=100):
        if Lambdaexp.call_by_name: return self
        # in call by name there are no reductions inside abstractions
        inner = self.inner.reduce(maxlevel-1)
        return Applicableexp(inner, self.arg)

    def asstring(self, m, pretty=False):
        if not pretty: return "\lambda"+Boundvar(self.arg).asstring(m, False)+".".
        ("+self.inner.asstring(m)+"")
        return self.addconst(green("\lambda")+Boundvar(self.arg).asstring(m, True)
        )+".( "+self.inner.asstring(m, True)+"")"
```

In [4]:

```
class Boundvar(Lambdaexp):
    """Bounded variable"""

    def __init__(self, arg):
        self.myid = arg
        super().__init__()

    def replace(self, argnum, exp): return exp if argnum==self.myid else self

    def bounded(self): return { self.myid }

    def asstring(self, m, pretty=False):
        arg = m.get(self.myid, self.myid)
```

```

        return cnr(ora('a')+arg)

class Unboundvar(Lambdaexp):
    """Unbounded (free) variable."""
    def __init__(self, name):
        self.name = name
        super().__init__()

    def replace(self, name, arg): return arg if name==self.name else self

    def asstring(self, m, pretty=False):
        return self.addconst(self.name) if pretty else self.name

class Combinedexp(Lambdaexp):
    """Combined expression of two expressions."""
    def __init__(self, exp1, exp2):
        self.exp1 = exp1
        self.exp2 = exp2
        super().__init__()

    def replace(self, arg, exp):
        return Combinedexp(self.exp1.replace(arg, exp), self.exp2.replace(arg, exp))

    def bounded(self): return self.exp1.bounded() | self.exp2.bounded()

    def _reduce(self, maxlevel=100):
        if not maxlevel: return self
        e1 = self.exp1.reduce(maxlevel-1)
        if isinstance(e1, Applicableexp):
            return e1.apply(self.exp2).reduce(maxlevel-1)
        return Combinedexp(e1, self.exp2)

    def asstring(self, m, pretty=False):
        s = f"({self.exp1.asstring(m, False)} {self.exp2.asstring(m, False)})"
        if not pretty: return s
        return f"({self.exp1.asstring(m, True)} {self.exp2.asstring(m, True)})"

```

```

In [5]: class λ:
        """Binds a variable name in a Lambda expression"""

        def __init__(self, *varlist):
            """
            Get list of unbounded variables (for example a,b,c) and returns an
            operator that binds an expression exp to
            λa(λb(λc(exp))) and so on. """
            if not varlist: raise Exception("Need to bind at least one variable")

            self.varlist = varlist[::-1]

```

```

def bindexp(self,exp):
    res = exp
    for v in self.varlist:
        res = Applicableexp(res,v.name)
    return res

#-----#
-----#
# Ignore this code in first read: Python specific details

def __call__(self,*args):
    exp = functools.reduce(operator.mul,args[1:],args[0])
    return self.bindexp(exp)

#-----#
-----#

```

Initialization

The above is all the code for implementing the λ calculus. We now add some convenient global variables: $\lambda a \dots \lambda z$ and $a \dots z$ for variables, and 0 and 1.

```

In [6]: Lambdaexp.lambdanames = {}
import string

def initids(g):
    """Set up parameters a...z and correponding Binder objects  $\lambda a.. \lambda z$ """
    lcase = list(string.ascii_lowercase)
    ids = lcase + [n+"_" for n in lcase]
    for name in ids:
        var = Unboundvar(name)
        g[name] = var
        g[" $\lambda$ "+name] =  $\lambda$ (var)
        Lambdaexp.lambdanames[var] = name

```

```
In [7]: initids(globals())
```

```
In [8]: # testing...
 $\lambda y(y)$ 
```

```
Out[8]:  $\lambda a.(a)$ 
```

```
In [9]:  $\lambda(a,b)(a)$ 
```

```
Out[9]:  $\lambda a.(\lambda \beta.(a))$ 
```

```

In [10]: def setconstants(g,consts):
    """Set up constants for easier typing and printing."""

    for name in consts:
        Lambdaexp.lambdanames[consts[name]] = name
        if name[0].isalpha():
            g[name]=consts[name]

```

```

        else: # Numeric constants such as 0 and 1 are replaced by _0 and _
1
            g["_"+name] = consts[name]

setconstants(globals(),{"1" :  $\lambda(x,y)(x)$  , "0" :  $\lambda(x,y)(y)$  })

def register(g,*args):
    for name in args:
        Lambdaexp.lambdanames[g[name]] = name

```

```
In [11]: # testing
         $\lambda a(\lambda z(a))$ 
```

```
Out[11]: 1
```

λ calculus playground

We can now start playing with the λ calculus

If you want to use the λ character you can copy paste it from here: [λ](#)

Let's start with the function $\lambda x,y.y$, also known as 0

```
In [12]:  $\lambda a(\lambda b(b))$ 
```

```
Out[12]: 0
```

Our string representation recognizes that this is the 0 function and so "pretty prints" it. To see the underlying λ expression you can use `__repr__()`

```
In [13]:  $\lambda a(\lambda b(b)).__repr__()$ 
```

```
Out[13]: ' $\lambda a. (\lambda \beta. (\beta))$ '
```

Let's check that `_0` and `_1` behave as expected

```
In [14]:  $_1(a,b)$ 
```

```
Out[14]: a
```

```
In [15]:  $_0(a,b)$ 
```

```
Out[15]: b
```

```
In [16]:  $_1$ 
```

```
Out[16]: 1
```

```
In [17]:  $_1(_0)$ 
```

```
Out[17]:  $\lambda a. (0)$ 
```



```
In [18]: _1.__repr__()
```

```
Out[18]: 'λa.(λβ.(α))'
```

Here is an exercise:

Question: Suppose that $F = \lambda f. (\lambda x. (fx)f)$, $1 = \lambda x. (\lambda y. x)$ and $0 = \lambda x. (\lambda y. y)$.
What is $F\ 1\ 0$?

a. 1

b. 0

c. $\lambda x.1$

d. $\lambda x.0$

Let's evaluate the answer

```
In [19]: F=λf(λx((f*x)*f))
F
```

```
Out[19]: λa.(λβ.(((α β) α)))
```

```
In [20]: F(_1)
```

```
Out[20]: λa.(((1 α) 1))
```

```
In [21]: F(_1,_0)
```

```
Out[21]: 0
```

```
In [22]: ID = λa(a)
register(globals(),"ID")
```

Some useful functions

Let us now add some of the basic functions in the λ calculus

```
In [23]: NIL= λf(_1)
PAIR =λx(λy(λf(f*x*y)))
ISEMPTY= λp(p *(λx(λy(_0))))
HEAD = λp(p(_1))
TAIL =λp(p *_0)
IF = λ(a,b,c)(a * b * c)

register(globals(),"NIL", "PAIR")
```

And test them out

```
In [24]: ISEMPTY(NIL)
```

Out[24]: 1

In [25]: IF(_0,a,b)

Out[25]: b

In [26]: IF(_1,a,b)

Out[26]: a

In [27]: P=PAIR(_0,_1)

In [28]: HEAD(P)

Out[28]: 0

In [29]: TAIL(P)

Out[29]: 1

We can make lists of bits as follows:

```
In [30]: def makelist(*L):  
         """Construct a λ List of _0's and _1's."""  
         if not L: return NIL  
         h = _1 if L[0] else _0  
         return PAIR(h,makelist(*L[1:]))
```

In [31]: L=makelist(1,0,1)
L

Out[31]: $\lambda\alpha.(((\alpha\ 1)\ ((\text{PAIR}\ 0)\ ((\text{PAIR}\ 1)\ \text{NIL}))))$

In [32]: HEAD(L)

Out[32]: 1

In [33]: TAIL(L)

Out[33]: $\lambda\alpha.(((\alpha\ 0)\ ((\text{PAIR}\ 1)\ \text{NIL})))$

In [34]: HEAD(TAIL(L))

Out[34]: 0

In [35]: HEAD(TAIL(TAIL(L)))

Out[35]: 1

Recursion

We now show how we can implement recursion in the λ calculus. We start by doing this in Python. Let's try to define XOR in a recursive way and then avoid recursion

```
In [36]: # XOR of 2 bits
def xor2(a,b): return 1-b if a else b

# XOR of a List - recursive definition
def xor(L): return xor2(L[0],xor(L[1:])) if L else 0

xor([1,0,0,1,1])
```

Out[36]: 1

Now let's try to make a *non recursive* definition, by replacing the recursive call with a call to *me* which is a function that is given as an extra argument:

```
In [37]: def myxor(me,L): return 0 if not L else xor2(L[0],me(L[1:]))
```

The first idea is to try to implement `xor(L)` as `myxor(myxor, L)` but this will not work:

```
In [38]: def xor(L): return myxor(myxor,L)

try:
    xor([0,1,1])
except Exception as e:
    print(e)
```

myxor() missing 1 required positional argument: 'L'

The issue is that `myxor` takes *two* arguments, while in `me` we only supply one. Thus, we will modify `myxor` to `tempxor` where we replace the call `me(x)` with `me(me,x)`:

```
In [39]: def tempxor(me,L): return myxor(lambda x: me(me,x),L)
```

Let's check this out:

```
In [40]: def xor(L): return tempxor(tempxor,L)

xor([1,0,1,1])
```

Out[40]: 1

This works!

Let's now generalize this to any function. The `RECURSE` operator will take a function `f` that takes two arguments `me` and `x` and return a function `g` where the calls to `me` are replaced with calls to `g`

```
In [41]: def RECURSE(f):
    def ftemp(me,x): return f(lambda x: me(me,x),x)
```

```
return lambda x: ttemp(ttemp,x)

xor = RECURSE(myxor)

xor([1,1,0])
```

Out[41]: 0

The λ version

We now repeat the same arguments with the λ calculus:

```
In [42]: # XOR of two bits
XOR2 =  $\lambda(a,b)(\text{IF}(a,\text{IF}(b,0,1),b))$ 

# Recursive XOR with recursive calls replaced by m parameter
myXOR =  $\lambda(m,l)(\text{IF}(\text{ISEMPTY}(l),0,\text{XOR2}(\text{HEAD}(l),m(\text{TAIL}(l)))))$ 

# Recurse operator (aka Y combinator)
RECURSE =  $\lambda f((\lambda m(f(m*m)))(\lambda m(f(m*m))))$ 

# XOR function
XOR = RECURSE(myXOR)
```

Let's test this out:

```
In [43]: XOR(PAIR(1,NIL)) # List [1]
```

Out[43]: 1

```
In [44]: XOR(PAIR(1,PAIR(0,PAIR(1,NIL)))) # List [1,0,1]
```

Out[44]: 0

```
In [45]: XOR(makelist(1,0,1))
```

Out[45]: 0

```
In [46]: XOR(makelist(1,0,0,1,1))
```

Out[46]: 1