1 Introduction

It is customary to make use of simple arithmetic to introduce lambda calculus. Here are two functions:

$$f(x) = x + 2 \tag{1}$$
$$g(x) = x^2 \tag{2}$$

The composition of these functions is expressed as:

$$f \circ g = f(g(x)) \tag{3}$$

(3) is of no use without looking up the definitions of the names f and g in (1) and (2). Furthermore, it is simply impossible to define a function that takes two functions as arguments and gives a third function which is the composition of the first two. Actually, it is impossible to define any higher order function within the language of our expressions – i.e. numbers, arithmetic operators and parentheses.

Given a formal language – the language of arithmetic in our case, lambda calculus adds to the language the lambda operator 'λ' and the dot '.'. This additions allow one to write anonymous functions who do not bear names like f or g, by which you look up definitions. Instead, in lambda calculus, the name and the function definition are one and the same entities. Here is how functions in (1) and (2) are rendered in lambda calculus:

$$(\lambda x.x + 2) \tag{4}$$
$$(\lambda x.x^2) \tag{5}$$

Application of a function to its argument is depicted by concatenating the function and its argument and enclosing both in a parentheses. To compute the result of application, you substitute the argument to its place as in standard functions and get rid of the lambda, the variable and the dot.

$$\begin{aligned}&((\lambda x.x + 2)8) \\ &= 8 + 2 = 10\end{aligned} \tag{6}$$

$$\begin{aligned}&((\lambda x.x^2)8) \\ &= 8^2 = 64\end{aligned} \tag{7}$$

So far everything appears as a little more cryptic way of defining functions and computing the result of applying a function to its argument. The real power of lambda calculus becomes visible when you apply functions to other functions, in exactly the same way you apply them to non-function arguments. We will illustrate this over function composition. Things will git a little complicated at this point; either go very slowly, or have a first pass, read on and come back. In either case you need to use pencil and paper to understand the transitions between forms. Now let us have the following higher order function that composes two functions – the lambda calculus way of having $f_1 \circ f_2$:

$$(\lambda f_1.(\lambda f_2.(\lambda z.(f_1(f_2 z))))) \tag{8}$$

First apply this to $(\lambda x.x + 2)$:

$$\begin{aligned}&((\lambda f_1.(\lambda f_2.(\lambda z.(f_1(f_2 z))))(\lambda x.x + 2)) \\ =&(\lambda f_2.(\lambda z.((\lambda x.x + 2)(f_2 z))))\end{aligned} \tag{9}$$

Now apply this result to the second function $(\lambda y.y^2)$:

$$\begin{aligned}&((\lambda f_2.(\lambda z.((\lambda x.x + 2)(f_2 z))))(\lambda y.y^2)) \\ =&(\lambda z.((\lambda x.x + 2)((\lambda y.y^2)z)))\end{aligned} \tag{10}$$

We obtained the composition of the functions defined in (1) and (2). To see that this is indeed the case, apply this composite to an argument, say 6:

$$\begin{aligned}&((\lambda z.((\lambda x.x + 2)((\lambda y.y^2)z)))6) \\ &= ((\lambda x.x + 2)((\lambda y.y^2)6))\end{aligned} \tag{11}$$

If you look carefully, you will realize two more patterns in the form of (fa) – a function f applied to argument a – in the result we obtained in (11). The one that is relatively easier to see is

$$((\lambda y.y^2)6) \tag{12}$$

if you perform this application you will obtain 36 as result. Replacing the occurrence of (12) in (11) with 36 will give,

$$\begin{aligned}&((\lambda x.x+2)36) \\ &=36+2=38\end{aligned} \tag{13}$$

We now turn to a detailed and formal characterization of the concepts we encountered.

2 The set of lambda terms

From here on, we take a rather abstract stance towards lambda calculus. The discussion will apply to any domain of use, not just arithmetic. Try to perceive the lambda calculus as a system of manipulating expressions constructed from a set of names, the lambda, the dot, and left and right parentheses, which we call **lambda terms**. Also refrain from assuming that x,y,z stand for variables, f,g,h for functions, a,b,c for constants, etc. Names are just names; although our use may be suggestive at times, there is no necessary association between a name and the type of object it stands for.

Definition 2.1 (Lambda terms)
Let $A=\{a,b,\ldots,z\}$ be the set of **names**, we extend the set by subscripts as we need so that we have an infinite set of names.

i. all names are lambda terms;

ii. if ω_1 and ω_2 are lambda terms, so is $(\omega_1\omega_2)$; (concatenation/application)

iii. if ω is a lambda term and α is a name, then $(\lambda\alpha.\omega)$ is an expression; (abstraction)

iv. nothing else is an expression.

□

We usually shorten "lambda term" to "term". Given a name α, we call the expression $\lambda\alpha$ a "lambda binder".

Example 2.2
Some example terms:

x	(xy)	$(x(yz))$	$((xy)z)$
$(\lambda x.x)$	$(\lambda y.(\lambda x.x))$	$(\lambda z.(x(\lambda y.(yz))))$	$(x(\lambda z.(\lambda y.(yz))))$
$(x(\lambda x.x))$	$((\lambda y.(\lambda x.x))(\lambda x.x))$	$(((\lambda y.(\lambda x.x))(\lambda x.x))(xy))$	$((x(yz))((xy)z))$

□

Once again note that in lambda calculus we write (fx) (simplified to fx) rather than the usual $f(x)$ to represent the application of function f to the argument x.

3 Notational conventions

A. Omit outer parentheses.

B. Concatenation of terms associate to left:

$$\begin{aligned}\omega_1\omega_2\omega_3\ldots\omega_n &\equiv (\ldots((\omega_1\omega_2)\omega_3)\ldots\omega_n)\\ fxyz &\equiv (((fx)y)z)\\ fx(yz) &\equiv ((fx)(yz))\\ f(xyz) &\equiv (f((xy)z))\\ f(xy)z &\equiv ((f(xy))z)\end{aligned}$$

C. The scope of dot extends to right until the first right parenthesis whose matching pair falls to the right of the dot:

$$(\lambda x.xx)y \equiv ((\lambda x.xx)y) \not\equiv \lambda x.xxy$$
$$\lambda x.xyz \equiv (\lambda x.xyz) \not\equiv (\lambda x.x)yz$$

D. Stacked lambda binders associate to right, and dots between lambda binders are deleted:

$$\lambda\alpha_1\lambda\alpha_2\lambda\alpha_3\ldots\lambda\alpha_n.\omega \equiv (\lambda\alpha_1.(\lambda\alpha_2.(\lambda\alpha_3\ldots(\lambda\alpha_n.\omega)\ldots)))$$
$$\lambda f\lambda x.f(fx) \equiv (\lambda f.(\lambda x.(f(fx))))$$
$$\lambda f.(\lambda x\lambda y.xyy)zf \equiv (\lambda f.(((\lambda x.(\lambda y.((xy)y)))z)f))$$

Example 3.1
The terms in Example 2.2 in simplified form:

$$(xy) \equiv xy$$
$$(x(yz)) \equiv x(yz)$$
$$((xy)z) \equiv xyz$$
$$(\lambda x.x) \equiv \lambda x.x$$
$$(\lambda y.(\lambda x.x)) \equiv \lambda y.\lambda x.x$$
$$(\lambda z.(x(\lambda y.(yz)))) \equiv \lambda z.(x(\lambda y.yz))$$
$$(x(\lambda z.(\lambda y.(yz)))) \equiv x(\lambda z.\lambda y.yz)$$
$$(x(\lambda x.x)) \equiv x(\lambda x.x)$$
$$((\lambda y.(\lambda x.x))(\lambda x.x)) \equiv (\lambda y\lambda x.x)\lambda x.x$$
$$(((\lambda y.(\lambda x.x))(\lambda x.x))(xy)) \equiv (\lambda y\lambda x.x)(\lambda x.x)(xy)$$
$$((x(yz))((xy)z)) \equiv x(yz)(xyz)$$

□

Here are some exercises from [1] to train your hand on notational conventions:

Exercise 3.2
Simplify the following terms by removing parentheses and dots using the conventions above:

i. $(\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))$
ii. $(((ab)(cd))((ef)(gh)))$
iii. $(\lambda x.((\lambda y.(yx))(\lambda v.v)z)u)(\lambda w.w)$

□

Exercise 3.3
Restore the parentheses to the following terms:

i. $xxxx$
ii. $\lambda x.x\lambda y.y$
iii. $\lambda x.(x\lambda y.yxx)x$

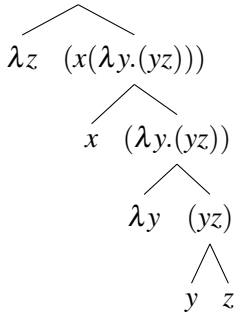
□

4 Bondage and freedom

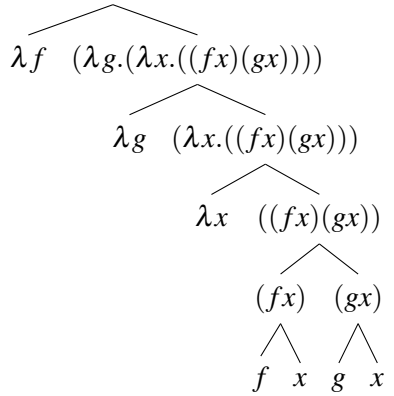
4.1 Expressions as binary trees

First observe that every compound term (=terms that are more complicated than a name) corresponds to a binary tree; this is obvious if you take the steps (ii) and (iii) in Definition (2.1) as an instruction to form a binary branching tree. Here are some examples:

(1) a. $(\lambda z.(x(\lambda y.(yz))))$

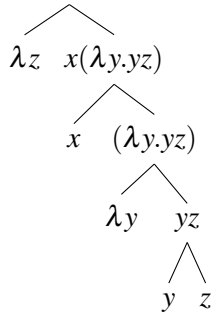


b. $(\lambda f.(\lambda g.(\lambda x.((fx)(gx)))))$

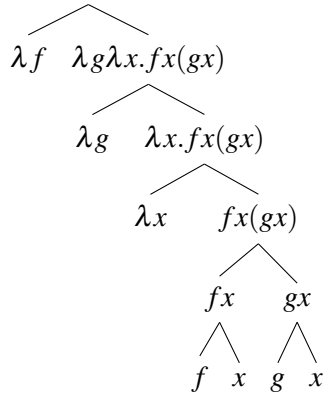


Now the same examples simplified by the conventions of Section 3:

(2) a. $\lambda z.x(\lambda y.yz)$



b. $\lambda f\lambda g\lambda x.fx(gx)$



We need some auxiliary definitions before we define bondage and freedom.

Occurrence
The notions of bondage and freedom are defined for **occurrences** of names and lambda binders. As every lambda term except names corresponds to a binary tree, each leaf[1] in the tree is either an occurrence of a name or a lambda binder. Note that we do not count a name that comes right after the lambda operator λ an occurrence of that name. For instance the name x has 1 occurrence in (2a) and 2 occurrences in (2b).

Step
In traversing a tree, going from one node to an adjacent[2] node is one **step**.

Bondage
Given an expression Γ and a name α, an occurrence of α is **bound** in Γ by an occurrence of a lambda binder $\lambda\alpha$ iff one can go from the name to the lambda term in n steps and there is no other occurrence of $\lambda\alpha$ such that it takes fewer than n steps to reach from the name to that occurrence.

Note that bondage is defined between occurrences of names and lambda binders and it is always relative to an expression, which goes unmentioned when clear from the context.

[1] Leaves are nodes without children.
[2] Two nodes are adjacent if there exists a line connecting them without any nodes in between.

Freedom
Given an expression Γ and a name α, an occurrence of α is **free** in Γ, if it is not bound by any occurrence of a lambda binder in Γ.

Example 4.1

- In (2a), there is a single occurrence of x and it is free in (2a)
- In (2b), there is no free occurrence of a name; all the occurrences of all the names are bound by some occurrence of a lambda term.
- Our definitions allow for expressions like $\lambda x\lambda k.cy$, where all the occurrences of all names are free and none of the lambda terms bind any name.
- In $\lambda x\lambda y.z(yx)y$ both occurrences of y are bound by λy, whereas in $(\lambda x\lambda y.z(yx))y$, the second y (counting from left to right) is free.

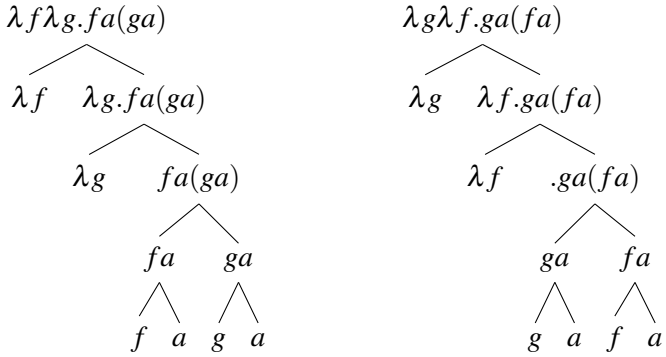
□

5 α-equivalence

The functions $\lambda x.x$ and $\lambda y.y$ are one and the same functions; they just do the same thing using different names. Similarly for $\lambda f\lambda g.fa(ga)$ and $\lambda g\lambda f.ga(fa)$. We express this relation of being alphabetical variants of the same term as **α-equivalence**. Symbolically,

$$\lambda f\lambda g.fa(ga) \equiv_\alpha \lambda g\lambda f.ga(fa)$$

Again think of α-equivalence over trees:



The terms have structurally identical trees with identical binding relations. The only difference between the two concerns the bound names. Here is a formal definition:

Definition 5.1 (α-equivalence)
Let Λ_1 and Λ_2 are lambda terms with construction trees τ_1 and τ_2;
and let ν_1 and ν_2 be sets of nodes of τ_1 and τ_2, respectively;
Λ_1 and Λ_2 are α-equivalent iff there exists a one to one correspondence c between ν_1 and ν_2 such that,

- for all nodes i, j in ν_1, i immediately dominates[3] j iff $c(i)$ immediately dominates $c(j)$;
- for all nodes i in ν_1, i is some name α free in τ_1 iff $c(i)$ is some name α free in τ_2;
- for all nodes i, j in ν_1, i is a lambda binder binding j iff $c(i)$ is a lambda binder binding $c(j)$.

□

If the above definition is hard to grasp in the first pass, one can also think of α-equivalence in procedural terms. One goes from a term to an α-equivalent term by renaming one or more bound variables of the original term. Care needs to be taken, however, in renaming bound variables. For instance take the following term,

$$\lambda x.z(\lambda y.yx) \tag{14}$$

You want to replace the name x with z. If you do the renaming without caring for accidental bondages you might introduce during renaming, you would obtain,

$$\lambda z.z(\lambda y.yz) \tag{15}$$

where you made the first z in (14) accidentally bound in (15). You have,

$$\lambda x.z(\lambda y.yx) \not\equiv_\alpha \lambda z.z(\lambda y.yz)$$

[3] A node ν_1 immediately dominates node ν_2 iff it takes only one step going down from ν_1 to ν_2

Likewise, renaming x to y would again yield a non-α-equivalent term:

$$\lambda x.z(\lambda y.yx) \not\equiv_\alpha \lambda y.z(\lambda y.yy)$$

At this point one way to avoid such accidental bondages, one should always rename to a name that is not in the term at all. But this move misses the general concept of α-equivalence. Observe that it is legitimate to rename y to z in (14):

$$\lambda x.z(\lambda y.yx) \equiv_\alpha \lambda x.z(\lambda z.zx)$$

We take another strategy for avoiding accidental bondage. We again make use of trees and define renaming as a single step procedure that targets a specific occurrence of a lambda binder. Here is our algorithm:

Let Λ be a lambda term, $\lambda\alpha$ be a lambda binder that occurs in Λ and ν a name; and let B be the set of all occurrences bound by $\lambda\alpha$.

Algorithm 5.2 (Stepwise rename)

function RENAME(*term*, *binder*, *name*) ▷ *term*: tree *binder*: node *name*: name
  nodes ← nodes bound by *binder*
  for *node* in *nodes* **do**
    replace the name on *node* with *name*
    if replacement creates new binder properly dominated by *binder* **then**
      ERROR!
  replace the name on binder with *name*
  if new bound nodes are created not in *nodes* **then**
    ERROR!

□

6 Substitution

7 β-reduction

- An expression of the form $(\lambda\alpha.\gamma)\omega$ is called a **β-redex**.
- It can be **β-reduced** to $\gamma[\omega/\alpha]$, called a **reduct**, if ω is free for α in γ.

- A lambda expression can be reduced by turning all the redexes to reducts, which results in a **β-normal** form.
- Some example reductions:

$$\begin{aligned}(\lambda f.fx)g &\rightarrow_\beta gx\\ (\lambda f.fx)ga &\rightarrow_\beta gxa\\ (\lambda f.fx)(ga) &\rightarrow_\beta gax\\ (\lambda f\lambda x.fx)ga &\rightarrow_\beta ga\end{aligned}$$

- There may be more than one redex in a expression:

$$\begin{aligned}(\lambda x.y)((\lambda z.zz)(\lambda w.w)) &\rightarrow_\beta (\lambda x.y)((\lambda w.w)(\lambda w.w))\\ &\rightarrow_\beta (\lambda x.y)(\lambda w.w) \rightarrow_\beta y\end{aligned}$$

- Another reduction of the same expression would be:

$$(\lambda x.y)((\lambda z.zz)(\lambda w.w)) \rightarrow_\beta y$$

- The first is called the **applicative order** reduction; the second is called the **normal order** reduction.
- Reduce the following expressions:

$$\begin{aligned}&(\lambda x.mx)j\\ &(\lambda y.yj)m\\ &(\lambda x.\lambda y.y(yx))jm\\ &(\lambda y.yj)(\lambda x.mx)\\ &(\lambda x.xx)(\lambda y.yyy)\end{aligned}$$

8 Lambda calculus in action: some examples

8.1 Logic

- Let's define the truth values:

$$\begin{aligned}\mathbf{T} &\equiv \lambda x\lambda y.x\\ \mathbf{F} &\equiv \lambda x\lambda y.y\end{aligned}$$

- Verify that $\lambda x\lambda y.yxy$ behaves like *and* in prefix notation (i.e. $\wedge pq$).
- Can you think of lambda expressions for $\vee$ and $-$?
- What about an if-then-else function that applies to the test, a function to execute if the test is true and a function to execute if the test is false.

8.2 Arithmetic

- Number can be represented as lambda expressions in the following way:

$$\begin{aligned}0 &\equiv \lambda f\lambda x.x\\ 1 &\equiv \lambda f\lambda x.fx\\ 2 &\equiv \lambda f\lambda x.f(fx)\\ 3 &\equiv \lambda f\lambda x.f(f(fx))\\ &\vdots\end{aligned}$$

- A successor function which returns $n+1$ given n is:

$$\mathbf{S} \equiv \lambda a\lambda f\lambda x.f(afx)$$

- Here are addition and multiplication (again in prefix notation); verify that they do what they are meant to do.

$$\begin{aligned}+ &\equiv \lambda a\lambda b\lambda f\lambda x.af(bfx)\\ \times &\equiv \lambda a\lambda b\lambda f.a(bf)\end{aligned}$$

[1]

References

[1] Peter Selinger. Lecture notes on the Lambda Calculus. 2013.