All you need is lambda

↓

introduction to lambda calculus : model of computation

↗ Haskell is lambda calculus

→ functional programming is a computer programming paradigm that relies on functions modelled on mathematical functions

↓

hence the essence of this paradigm is that programs are a combination of expressions.

↓

expressions include : a) concrete values
b) variables
c) functions

↓

expressions that are applied to an argument or input, and once applied, can be reduced or evaluated.

← first class elements of the paradigm

↓

can be used as values, passed as arguments, or inputs to other functions.

↓

Referential transparency : same fn. given the same values to evaluate, will always return the same result. This behaviour is similar to that in maths.

(or a mapping)

→ A function is a relation between a set of possible inputs and outputs. many to one mapping. This can be attributed

to first the general principle around functions in maths, and also stands testament to the principle of referential transparency.

↓

Keeping this definition of functions in mind is key to understanding functional programming.

→ The structure of lambda terms : Lambda calculus has three basic components : expressions, variables and abstractions.

·) expression refers to a superset of all those things. an expression can be a variable name, an abstraction or a combination.

↓

The simplest expression is a single variable.

↓

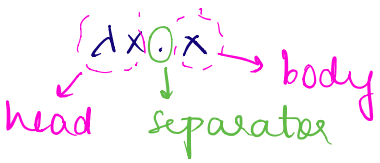
potential inputs to functions

·) abstraction is a function. Lambda term that has a head (a lambda) and a body, and is applied to an argument.

↓

input value

A simple function :



abstraction

↓ ↓

head (λ followed by a name)

body (another expression)

Variable named in head is the parameter and binds all instances of the same variable in the body of the function.

↓

lambda abstraction has no name → anonymous function

↙

generalization or abstraction from a concrete instance of a problem (concrete vs abstraction)

↓

named function can be called by name by another fn; an anonymous fn. cannot be.

→ alpha equivalence : In $\lambda x.\ x$, the x is not semantically meaningful except in its role in that single expression. Hence $\lambda x.\ x$, $\lambda d.\ d$, $\lambda z.\ z$, all mean the same thing. They are all the same function. This kind of equivalence is called alpha equivalence.

→ beta redn : The process of elimination of the head of the abstraction, since its only fn. was to bind the variables at the time of applying the fn. to an argument is called beta redn. for example.

$$\lambda x.\ x$$

↓ beta redn. when applying this to 2

$$(\lambda x.\ x)\ 2 \Rightarrow 2$$

∴ beta redn. is the process of applying a lambda term to an argument, replacing the bound variables with the value of the argument, and eliminating the head. Eliminating the head tells us that the fn. has been applied.

→ Applying lambda abstraction to another lambda abstraction.

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

↓

Here we replace x with the argument abstraction.

↓

Redn. process : $(\lambda x.x)(\lambda y.y)$

$[x := (\lambda y.y)]$

$\lambda y.y$ → final result is another abstraction.

↓

No other argument to apply to this, so we have nothing to reduce.

↓

∴ the process of redn. stops when there are either no more heads, or lambdas left to apply or no more arguments to apply functions to.

Now $(\lambda x.x)(\lambda y.y)\,z$

Redn 1 ⇒ $(\lambda y.y)\,z$

Redn 2 ⇒ z

→ Free variables : purpose of the head is to bind variables. A bound variable has the same value throughout the expression. But sometimes there are variables that are not named in the head. These variables are called free variables. for example

$\lambda x.xy$ — x: bound variable; y: free variable → irreducible

$(\lambda x.xy)z$

$\Rightarrow (\lambda[x:=z].xy)$ (variables replaced and head eliminated)

$\Rightarrow zy$ → irreducible

Note: alpha equivalence does not apply to free variables.

→ Multiple Arguments : each lambda can only bind one parameter and can accept one argument. functions that require multiple arguments have multiple, nested heads. They are applied one after the other. This process is called currying.

$\lambda xy.xy$ → convenient shorthand for two nested lambdas. (one for each x and y) i.e.

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

① $(\lambda xy.xy)\,1\,2$

$\Rightarrow (\lambda x.(\lambda y.xy))\,1\,2$

$\Rightarrow (\lambda y.y)\,2$ (binding and redn. for x)

$\Rightarrow 2$ (binding and redn. for y)

② $(\lambda xy.xy)(\lambda z.a)\,1$

$\Rightarrow (\lambda x.(\lambda y.xy))(\lambda z.a)\,1$ (binding and redn. for x)

$\Rightarrow (\lambda y.(\lambda z.a)y)\,1$ (binding and redn. for y)

$\Rightarrow (\lambda z.a)\,1$ (binding and redn. for z)

$\Rightarrow a$

③ $(\lambda xyz.xz(yz))(\lambda mn.m)(\lambda p.p)$

$\Rightarrow (\lambda x.(\lambda y.(\lambda z.xz(yz))))(\lambda mn.m)(\lambda p.p)$

$\Rightarrow$ binding with x and redn. λ

$\Rightarrow \lambda y(\lambda z.(\lambda mn.m)z(yz))(\lambda p.p)$

$\Rightarrow$ binding with y and redn λ.

$\Rightarrow \lambda z.((\lambda mn.m)\,z((\lambda p.p)z))$

$\Rightarrow$ nothing to bind z with, so we move inside

$\Rightarrow \lambda z((\lambda m.(\lambda n.m))\,z((\lambda p.p)z))$

$\Rightarrow$ binding z to m and redn. λ

$\Rightarrow \lambda z((\lambda n.z)((\lambda p.p)z))$

$\Rightarrow$ binding $((\lambda p.p)z)$ to n and redn n.

$\Rightarrow$ $\lambda z\ (z)$

·) we move inside the inner layer if we find something irreducible in the current layer

→ Beta normal form: when we cannot beta reduce the terms any further. This corresponds to a fully evaluated expression or in programming, a fully executed program

↓

normal form → beta normal form

imp. to know so that we know when to stop evaluating an expression

↓

Also valuable to have an appreciation for evaluation as simplification.

↓

Point is that if we have a function, such as I, saturated (all arguments applied) but we haven't simplified it to the final result, then it is not fully evaluated, but only applied. Application is what makes evaluation / simplification possible.

Normal form : $(10+2) * 100/2 \Rightarrow 12 * 100/2$

Arguments applied

$\Rightarrow 12 * 50$

$\Rightarrow 600$ (Reduced or evaluated)

Normal form

→ Combinators : lambda term with no free variables.

→ Divergence : Not all reducible lambda terms reduce neatly to a beta normal form. This isn't because they are already fully reduced, but rather because they diverge. Divergence here means that the redn. process never ends or terminates. for example, a lambda term called omega.

1. $(\lambda x.\, xx)(\lambda x.\, xx)$

$\Rightarrow$ $(\lambda x.\, xx)(\lambda x.\, xx)$ { binding for x and redn. λ)

↓

same term → hence this process never ends

↓

Divergence

↓

This is important because understanding what will terminate means what programs will do useful work and return the answer we want.

•) Haskell is typed lambda calculus i.e. the meaning of Haskell programs is centered around evaluating expressions rather than executing instructions

→ Solutions

Combinators : 1. , 3, 4

Normal form or diverge : 1 , 3 → Normal form
2 → Divergent

Beta reduce : 1. $(\lambda abc.cba)\ zz\ (\lambda wv.w)$

$\Rightarrow (\lambda a.(\lambda b.(\lambda c.cba)))\ zz\ (\lambda wv.w)$

$\Rightarrow$ binding z to a and redn. λ

$\Rightarrow (\lambda b.\lambda c.cbz)\ z\ (\lambda wv.w)$

$\Rightarrow$ binding z to b and redn. λ

$\Rightarrow (\lambda c.czz)\ (\lambda wv.w)$

$\Rightarrow$ binding to c and redn. λ

$\Rightarrow (\lambda wv.w)\ zz$

$\Rightarrow (\lambda w(\lambda v.w))\ zz$

$\Rightarrow$ binding to w and redn. λ

$\Rightarrow (\lambda v.z)\ z$

$\Rightarrow z$

2. $(\lambda x.\lambda y.xyy)\ (\lambda a.a)\ b$

$\Rightarrow (\lambda y.(\lambda a.a)yy)\ b$

$\Rightarrow (\lambda a.a)(b)b$

$\Rightarrow$ bb

3. $(\lambda y.y)(\lambda x.xx)(\lambda z.zq)$

$\Rightarrow$ $(\lambda x.xx)(\lambda z.zq)$

$\Rightarrow$ $(\lambda z.zq)(\lambda z.zq) \Rightarrow (\lambda z.zq)q \Rightarrow qq$

4. $(\lambda z.z)(\lambda x.xx)(\lambda w.wy)$

$\Rightarrow$ $(\lambda x.xx)(\lambda w.wy)$

$\Rightarrow$ $(\lambda w.wy)(\lambda w.wy) \Rightarrow (\lambda w.wy)y \Rightarrow yy$

5. $(\lambda x.\lambda y.xyy)(\lambda y.y)y$

$\Rightarrow$ $(\lambda y.(\lambda y.y)yy)y$

$\Rightarrow$ $(\lambda y.y)yy$

$\Rightarrow$ yy

6. $(\lambda a.aa)(\lambda b.ba)c$

$\Rightarrow$ $(\lambda b.ba)(\lambda b.ba)c$

$\Rightarrow$ $(\lambda b.ba)ac$

$\Rightarrow$ aac

7. $(\lambda xyz.xz(yz))(\lambda x.z)(\lambda x.a)$

$\Rightarrow (\lambda x.\lambda y.\lambda z.xz(yz))(\lambda x.z)(\lambda x.a)$

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

$\Rightarrow$ $\lambda z.((\lambda x.z)z(\lambda x.a)z)$

$\Rightarrow \lambda z\,(z\,(\lambda x.a)\,z)$

$\Rightarrow \lambda z\,(z\,a)$

*) Normal order is a common evaluation strategy in lambda calculi. Normal Order means evaluating (i.e. applying or beta reducing) the leftmost outermost lambdas first, evaluating terms nested within after we're run out of arguments to apply.

↓

Haskell code is evaluated by call-by-need method instead.