ANYTHING FROM ALMOST NOTHING
Haskell NOTE

SiKai Lu

May 2020

1 Introduction of Introduction

Haskell is sometimes being dismissed as academic because it is relatively up-to-data with the current state of mathematics and computer science research. In our view, that progress is good and helps us solve practical problems in modern computing and software design.

Scope is the area you can declare variables.i.e. global scope and local scope.If a variable is declared in the global scope, we can reference it anywhere within the program.Otherwise,it can only be referenced within the local scope it defined.

In simple words, lexcial scoping means the children scope has access to the variables defined in the parent scope. For instance, If I were to define a function and declare a variable inside it and inside the very same function, define another function, then I should be able to use that variable inside the inner function because of lexical scoping.

```
function outerFunction() {
    var variable1 = 'ney vatsa'
    const variable2 = 'shashank jha'
    let variable3 = 'huda'

    function innerFunction() {
        console.log(variable1)
        console.log(variable2)
        console.log(variable3)
    }
    innerFunction()
}
outerFunction()

// ney vatsa
//shashank jha
//huda
```

A closure is a special kind of object that combines two things: a function, and the environment in which that function was created. The environment consists of any local variables that were in-scope at the time that the closure was created.

```
1
2   function add(x) {
3       return function(y) {
4         return x + y;
5       };
6     }
7
8     var addFive = add(5);
9     var addTen = add(10);
10
11    console.log(addFive(10));  // 15
12    console.log(addTen(10)); // 20
```

Now we create new functions addFive() and addTen(), by passing arguments inside add() function, these addTen() and addFive() are actually closures, and although they have the same function body definition, they store different lexical environments. In addFive() lexical environment, x is five, while in the lexical environment for addTen(), x is ten.

---

2 All You Need is Lambda

Functional programming languages are all based on the lambda calculus.

The lambda calculus is one process for formalizing a method. Like Turing machines, the lambda calculus formalizes the concept of effective computability, thus determining which problems, or classes of problems, can be solved.

---

3 What is functional programming?

The essence of functional programming is that programs are a combination of expressions. Expressions include concrete values, variables and also functions.

Functions have a more specific definition: they are expressions that are applied to a n argument or input and, once applied, can be reduced or evaluated.

Functions are first-class: they can be used as values or passed as arguments,or inputs, to yet more functions.

Functional programming languages are all based on the lambda calculus. Some languages in this general category incorporate features that aren't translatable into lambda expressions. Haskell is a pure functional language, because it does not.

The word purity in functional programming is sometimes also used to mean what is more properly called referential transparency,given the same values to evaluate, will always return the same result in pure functional programming, as they do in math.

Haskell's pure functional basis also lends it a high degree of abstraction and composability. Abstraction allows you to write shorter, more concise programs by factoring common, repeated structures into more generic code that can be reused.

---

4 What is a function?

A function is a relation between a set of possible inputs and a set of possible outputs. The function itself defined and represents that relationship.When you apply a function such as addition to two inputs, it maps those two inputs to an output - the sum of those numbers.

The input set is known as the domain. The set of possible outputs for the function is called the codomain. Domains and codomains are sets of unique values. The mapping between the domain and the codomain need not be one-to-one. In some cases, multiple input values will map to the same value in the codomian. However, a given input should not map to multiple outputs.

The relationship of inputs and outputs is defined by the function and that the output is predictable when you know the input and the function definition

---

A function is a mapping of a set of inputs to a set of outputs.

---

5 The structure of lambda expressions

The lambda calculus has three basic components : expressions, variables, and abstractions.

The word expression refers to a superset of all those things : an expression can be a variable name, an abstraction, or a combination of those things.

An abstraction is a function. It is a lambda term that has a head(a lambda) and a body and is applied to an argument(an input value).The head of the function is a λ followed by a variable name. The body of the function is

another expression.

The variable named in the head is the parameter and binds all instances of that same variable in the body of the function. The act of applying a lambda function to an argument is called application, and application is the lynchpin of the lambda calculus.

A simple function might look like this :

$$\lambda x.x$$

When we apply this function to an argument, each x in the body of the function will have the value of that argument.

The lambda abstraction $\lambda x.x$ has no name. It is an anonymous function. A named function can be called by name by another function, an anonymous function cannot.

The dot (.) separates the parameters of the lambda from the function body.All thing precedes the dot is marked as the head of the lambda,Otherwise, the body of the lambda.

When we apply the abstraction to arguments, we replace the names with values, making it concrete.

---

6 Alpha Equivalence

$$\lambda x.x$$

The variable x is not semantically meaningful except in its role in that single expression. Therefore, the following abstractions are equivalent:

$$\lambda x.x$$
$$\lambda a.a$$
$$\lambda b.b$$

This property is often referred as alpha equivalence.

---

7 Beta reduction

When we apply a function to an argument, we substitute the input expression for all instances of bound variables within the body of the abstraction. This process is called beta reduction.Beta reduction 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.

$$\lambda x.x$$

This function is the identity function. All it does is accept a single argument x and return that same argument.

The input of an abstraction can be another abstraction.

$$(\lambda x.x)(\lambda y.y)$$
Introducing $(\lambda y.y)$ into $(\lambda x.x)$ as x
$$[x := (\lambda(y, y)]$$
$$\lambda(y, y)$$

Applications in the lambda calculus are left associative,unless specific parentheses suggest.

More example :

$$\begin{aligned}(\lambda x.x)(\lambda y.y)z\\ [x := (\lambda(y, y)]\\ (\lambda y.y)z\\ [y := z]\\ z\end{aligned}$$

A computation consists of an initial lambda expression plus a finite sequence of lambda terms, each deduced from the preceding term by one application of beta reduction. We keep following the rules of application, substituting arguments for bound variables until there are no more heads left to evaluate or no more arguments to apply them to.

---

8 Free variables

Free variables are variables in the body that are not bound by the head.

In $\lambda x.xy$, x in the body is a bound variable,because it is named in the head of the function ,while the y is a free variable, because it is not. When we apply this function to an argument, nothing can be done with the y, it remains irreducible.

---

9 Multiple Arguments

Each lambda can only bind one parameter and can only accept one argument. Functions that require multiple arguments have multiple, nested heads.

$\lambda xy.xy$ is a convenient shorthand for two nested lambdas $\lambda x.(\lambda y.xy)$. When you apply the first argument, you're binding x, eliminating the outer lambda, and have $\lambda y.xy$ with x being whatever the outer lambda was bound to.

$$\begin{aligned}
&\lambda xy.xy \\
&(\lambda xy.xy)12 \\
&\lambda x.(\lambda y.xy)12 \\
&[x := 1] \\
&(\lambda y.1y)2 \\
&[y := 2] \\
&12
\end{aligned}$$

Another Example:

$$\begin{aligned}
&\lambda xy.xy \\
&(\lambda xy.xy)(\lambda z, a)1 \\
&[x := (\lambda z, a)] \\
&(lambday.(\lambda z.a)y)1 \\
&[y := 1] \\
&(\lambda z.a)1 \\
&[z := 1] \\
&a
\end{aligned}$$

It's more common in academic lambda calculus to use abstract variables

instead of concrete values.

$$
\begin{aligned}
&(\lambda xyz.xz(yz))(\lambda mn.m)(\lambda p.p) \\
&[x := \lambda mn.m] \\
&\lambda yz.(\lambda mn.m)z(yz)(\lambda p.p) \\
&[y := \lambda p.p] \\
&\lambda z.(\lambda mn.m)z(\lambda p.p) \\
&[m := z \text{ and } n = (\lambda p.p)] \\
&\lambda z.z
\end{aligned}
$$

---

10 Evaluation is simplification

Beta normal form is a state when you cannot beta reduce the terms any further. This corresponds to a fully evaluated expression.

The normal form of $(10 + 2) * 100/2$ is 600. We cannot reduce the number 600 any further. There are no more functions that we can beta reduce. Normal form means there is nothing left that can be reduced.

It's valuable to have an appreciation(understanding) for evaluation as a form of simplification when you get to Haskell code.

---

11 Combinations

A combinator is a lambda term with no free variables. Combinators serve only to combine the arguments they are givens.

The following are combinators :

$$\lambda x.x$$
$$\lambda xy.x$$
$$\lambda xyz.xz(yz)$$

The following are not :

$$\lambda y.x$$
$$\lambda x.xz$$

12 Divergence

Not all reducible lambda terms reduce to a normal form. This isn't because they're already fully reduced, but because they diverge. Divergence means that the reduction process never terminates or ends. Divergence is the opposite of convergence(normal form).

An example of Divergence :

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

Substituting $(\lambda x.xx)$ for each occurence of x. We're back to where we started, and this reduction process never ends.

This matters in programming, because terms that diverge are terms that dont't produce an answer or meaningful result.

13 Summary

These things all apply to Haskell, as they do to any purely functional language. Haskell is a typed lambda calculus-more on types later-with a lot of surface-level decoration sprinkled on top, to make it easier for humans to

write, but the semantics of the core language are the same as the lambda calculus. That is, the meaning of Haskell is centered around evaluating expressions rather than executing instructions.

14 Noteworthy Questions

Try to reduce the following expression in normal form:

$$(\lambda y.y)(\lambda x.xx)(\lambda z.zq)$$
$$(\lambda z.z)(\lambda z.zz)(\lambda z.zy)$$
$$(\lambda xyz.xz(yz))(\lambda x.z)(\lambda x.a)$$

$(\lambda y.y)(\lambda x.xx)(\lambda z.zq)$ is equivalent to $(\lambda z.z)(\lambda z.zz)(\lambda z.zy)$ due to alpha equivalence.
Proof :

$$\begin{array}{rl} (\lambda y.y)(\lambda x.xx)(\lambda z.zq) & (\lambda z.z)(\lambda z.zz)(\lambda z.zy) \\ (\lambda x.(\lambda x.x)(\lambda z.zq) & (\lambda z.(\lambda z.z)(\lambda z.zy) \\ (\lambda z.zq)(\lambda z.zq) & (\lambda z.zy)(\lambda z.zy) \\ (\lambda z.zq)(q) & (\lambda z.zy)(y) \\ zz & yy \end{array}$$

$$(\lambda xyz.xz(yz))(\lambda x.z)(\lambda x.a)$$
$$[x := (\lambda x.z)]$$
$$(\lambda yz1.(\lambda x.z)z1(yz1)(\lambda x.a)$$
$$[y := (\lambda x.a)]$$
$$\lambda z1.(\lambda x.z)z1(\lambda x.a)z1)$$
$$[x := z1] \text{ and } [x := z1]$$
$$\lambda z1.za$$

The z_1 notation allows us to distinguish two variables named z that came from different places. One is bound by the first head; the second is a free

variable in the second lambda expression.

We cannot reduce z further-it's free, and we know nothing, so we go inside yet another nesting and reduce $((\lambda x.a)(z1))$. $\lambda x.a$ gets to applied to $z1$ but tosses it away and returns the free variable a. Al of our terms are in normal order now.

15 Definition

The lambda in lambda calculus is the greek letter λ used to introduce, or abstract, arguments for binding in an expression.

A lambda abstraction is an anonymous function or lambda term.

$$\lambda x.x + 1$$

The head of the expression, $\lambda x.$, abstracts out the term $x + 1$. We can apply it to any x and recompute different results for each x to which we apply the lambda.

Application is how one evaluates or reduces lambdas binding the parameter to the concrete arguments. The argument is what specific term the lambda is applied to. Computations are performed in lambda calculus by applying lambdas to arguments until you run out of applications to perform.

Lambda calculus is a formal system for expressing programs in terms of abstraction and application.

Normal order is a common evaluation strategy in lambda calculus. Normal order means evaluating the leftmost, outermost lambdas first, evaluating nested terms after you have run out of arguments to apply. Normal order isn't how Haskell code is evaluated. Haskell's evaluation strategy is call-by-need.

16 Follow-up reading

There references are optional and intended only to offer suggestions for how you might deepen your understanding of this topic.

16.1 Raul Rojas. A Tutorial Introduction to the Lambda Calculus

The Lambda calculus can be called the smallest universal programming language of the world. The word 'universal' stands for that any computable function can be expressed and evaluated using this formalism. Thus it is equivalent to Turing machines. However, the lambda calculus emphasizes the use of transformation rules and does not care about the actual machine implementing them.

16.1.1 Definition

The central concept in lambda calculus is the "expression". It is defined recursively as follows:

$$< expression > := < variable > \mid < function > \mid < application >$$
$$< function > := \lambda < variable > . < expression >$$
$$< application > := < expression >< expression >$$

16.1.1.1 Free and bound variables

In lambda calculus all names are local to definitions. In the expression

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

the x in the body of the first expression from the left is bound to the first λ. The y in the body of the second expression is bound to the second λ and the x is free. It is very important to notice that the x in the second expression

is totally independent of the x in the first expression.

Formally we say that a variable $< name >$ is free in an expression if one of the following three cases holds:

1. $< name >$ is free in $< name >$
2. $< name >$ is free in $\lambda < name_1 > . < exp >$ if the identifier $< name > \neq < name_1 >$ and $< name >$ is free in $< exp >$.
3. $< name >$ is free $E_1 E_2$ if $< name >$ is free in E_1 or if it is free in E_2.

A variable $< name >$ is bound if one of two cases holds:

1. $< name >$ is bound in $\lambda < name_1 > . < exp >$ if the identifier $< name > == < name_1 >$ and $< name >$ is free in $< exp >$.
2. $< name >$ is bound $E_1 E_2$ if $< name >$ is bound in E_1 or if it is bound in E_2.

It should be emphasized that the same identifier can be occur free and bound in the same expression. In the expression

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

the first y is the parenthesized subexpression to the left. It is bound in the subexpression to the right. It occurs therefore to the right. It occurs therefore free as well as bound in the whole expression.

16.1.1.2 Substitutions

We should be careful when performing substitutions to avoid mixing up free occurrences of an identifier with bound ones. In the expression

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

the function to the left contains a bound y, whereas the y at the right is free. An incorrect substitution would mix the two identifiers in the erroneous result $(\lambda y.yy)$. Simply by renaming the bound y to t we obtain $(\lambda x.(\lambda t.xt))y = (\lambda t.yt)$ which is a completely different result but nevertheless the correct one.
Another example:

$$\begin{aligned}(\lambda x.(\lambda y.(x(\lambda x.xy))))y\\(\lambda x.(\lambda b.(x(\lambda a.ab))))y\\(\lambda b.(y(\lambda a.ab)))\end{aligned}$$

16.1.2 Arithmetic

Numbers can be represented in lambda calculus starting from zero and writing "suc(zero)" to represent 1, "suc(suc(zero))" to represent 2, and so on.
In the lambda calculus, we can only define new functions. Numbers will be defined as functions using the following approach:

$$\begin{aligned}0 &\equiv \lambda sz.z\\1 &\equiv \lambda sz.s(z)\\2 &\equiv \lambda sz.s(s(z))\\3 &\equiv \lambda sz.s(s(s(z)))\end{aligned}$$

and so on.

The successor function can be defined as

$$S \equiv \lambda wyx.y(wyx)$$

The successor function applied to zero yields

$$\begin{aligned}S0 &\equiv (\lambda wyx.y(wyx)(\lambda sz.z)\\&\quad [w := (\lambda sz.z)]\\S0 &\equiv (\lambda yx.y((\lambda sz.z)yx)\\&\quad [s := y \text{ and } z := x]\\S0 &\equiv (\lambda yx.y(x))\end{aligned}$$

$(\lambda yx.y(x))$ is equivalent to $\lambda sz.s(z)$ due to alpha equivalence.

Successor applied to 1 yield:

$$\begin{aligned} S0 &\equiv (\lambda wyx.y(wyx)(\lambda sz.s(z))) \\ &[w := (\lambda sz.s(z)] \\ S0 &\equiv (\lambda yx.y((\lambda sz.s(z))yx) \\ &[s := y \text{ and } z := x] \\ S0 &\equiv (\lambda yx.y(y(x))) \end{aligned}$$

16.1.2.1 Addition

Addition can be obtained immediately by adding layer(s) of the successor function on a number.

$$\begin{aligned} & 2 + 3 \\ & (\lambda sz.s(s(z))(\lambda wyx.y(wyx))(\lambda sz.s(s(s(z)))) \\ & [s := (\lambda wyx.y(wyx)] \text{ and } [z := (\lambda sz.s(s(s(z))))] \\ & (\lambda wyx.y(wyx))(\lambda wyx.y(wyx))(\lambda sz.s(s(s(z))))) \\ & \lambda sz.s(s(s(s(z))))) \end{aligned}$$

16.1.2.2 Multiplication

The multiplication of two numbers x and y can be computed using the following function:

$$M \equiv (\lambda xyz.x(yz))$$

An example :

$$
\begin{aligned}
&2 * 2 \\
&(\lambda xyz.x(yz))(\lambda sz.s(s(z)))(\lambda sz.s(s(z))) \\
&(\lambda z1.(\lambda sz.s(s(z)))((\lambda sz.s(s(z)))z1)) \\
&[x := (\lambda sz.s(s(z)))] \text{ and } [y := (\lambda sz.s(s(z)))] \\
&(\lambda z1.((\lambda sz.s(s(z)))((\lambda sz.s(s(z)))(z1))) \\
&[s := z1] \\
&(\lambda z1.(\lambda sz.s(s(z)))) \\
&[s := (\lambda z.z1(z1(z))] \\
&(\lambda z1.(\lambda z2.\lambda z.z1(z1(z))(\lambda z.z1(z1(z))(z2)))) \\
&[z := z2] \\
&(\lambda z1.(\lambda z2.\lambda z.z1(z1(z))(z1(z1(z2))))) \\
&[z := (z1(z1(z2)))] \\
&(\lambda z1.(\lambda z2.z1(z1(z1(z1(z2)))))) \\
&\lambda z1z2.z1(z1(z1(z1(z2))))
\end{aligned}
$$

16.1.3 Logic Gates

True in lambda calculus is defined as

$$T \equiv \lambda xy.x$$

False in lambda calculus is defined as

$$F \equiv \lambda xy.y$$

The Not gate is defined as

$$\neg \equiv \lambda a.b(\lambda xy.y)(\lambda xy.x) \equiv \lambda x.xFT$$

An example :

$$
\begin{aligned}
\neg T \equiv &(\lambda a.a(\lambda xy.y)(\lambda xy.x))(\lambda xy.x) \\
\equiv &(\lambda xy.x)(\lambda xy.y)(\lambda xy.x) \\
\equiv &(\lambda xy.y) \\
\equiv &F
\end{aligned}
$$

The AND gate is defined as

$$\wedge \equiv \lambda xy.xy(\lambda xy.y) \equiv \lambda xy.xyF$$

An example :

$$\begin{aligned} T \wedge F \equiv & (\lambda xy.xy(\lambda xy.y))(\lambda xy.x)(\lambda xy.y) \\ \equiv & (\lambda xy.x)(\lambda xy.y)(\lambda xy.y) \\ \equiv & \lambda xy.y \\ \equiv & F \end{aligned}$$

The OR gate is defined as

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

An example :

$$\begin{aligned} F \vee T \equiv & (\lambda xy.x(\lambda xy.y)y)(\lambda xy.y)(\lambda xy.x) \\ \equiv & (\lambda xy.y)(\lambda xy.y)(\lambda xy.x) \\ \equiv & \lambda xy.x \\ \equiv & T \end{aligned}$$

The NAND gate is defined as

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

An example:

$$\begin{aligned} \text{T NAND T} \equiv & (\lambda xy.xy(\lambda xy.y)(\lambda xy.x)(\lambda xy.x))(\lambda xy.x)(\lambda xy.x) \\ \equiv & (\lambda xy.x)(\lambda xy.x)(\lambda xy.y)(\lambda xy.x)(\lambda xy.x) \\ \equiv & (\lambda xy.x)(\lambda xy.y)(\lambda xy.x) \\ \equiv & \lambda xy.y \end{aligned}$$

Surprising:
The xor gate can be defined as if a then not b,otherwise b.

The XOR gate is defined as

$$\begin{aligned} \text{XOR} \equiv & \lambda xy.x(y(\lambda xy.y)(\lambda xy.x))y \equiv \lambda xy.x(yFT)y \end{aligned}$$

An example:

$$\begin{aligned}\text{F XOR F} \equiv&(\lambda xy.x(y(\lambda xy.y)(\lambda xy.x))y)(\lambda xy.y)(\lambda xy.y)\\ \equiv&(\lambda xy.y)((\lambda xy.y)(\lambda xy.y)(\lambda xy.x))(\lambda xy.y)\\ \equiv&(\lambda xy.y)(\lambda xy.x)(\lambda xy.y)\\ \equiv&(\lambda xy.y)\end{aligned}$$

16.1.4 Conditional

A premise:

$$0fa \equiv (\lambda sz.z)fa = a$$

For any functions f applied zero times to an argument a yields a.

Another premise:

$$Fa \equiv (\lambda xy.y)a = \lambda y.y \equiv \text{Identity Function}$$

It indicates that F applied to any argument yields the identity function.

It is very convenient in a programming language to have a function which is true if a number is zero and false otherwise. The following function Z complies with this requirement

$$Z \equiv \lambda x.xF\neg F$$

Test:
The function applied to zero yields

$$\begin{aligned}Z0 \equiv&(\lambda x.xF\neg F)0\\ \equiv&0F\neg F\\ \equiv&\neg F\\ \equiv&T\end{aligned}$$

The function applied to N yields

$$\begin{aligned} ZN \equiv & (\lambda x.xF\neg F)N \\ \equiv & NF\neg F \\ \equiv & I\neg F \\ \equiv & F \end{aligned}$$

16.1.5 The Predecessor function

We can define the predecessor function combining some of the function introduced above. When looking for the predecessor of n, the general strategy will be to create a pair (n,n-1) and then pick the second element of the pair as the result.

A pair (a,b) can be represented in lambda calculus using the function

$$(\lambda z.zab)$$

We can extract the first element of the pair from the expression applying this function to T

$$(\lambda z.zab)T \equiv Tab \equiv a$$

and the second applying the function to F

$$(\lambda z.zab)F \equiv Fab \equiv b$$

The following function returns $(n+1, n)$ from $(n.n-1)$

$$\Phi \equiv (\lambda pz.z(S(pT))(pT))$$

The subexpression pT extracts the first element from the pair p. A new pair is formed using this element. The first element of the new pair is the incremented version of the element and the element is placed as the second element of the new pair.

An example :

$\Phi(\lambda a.a(\lambda sz.s(s(z)))(\lambda sz.s(z)))$
$\equiv (\lambda pz.z(S(pT))(pT))(\lambda a.a(\lambda sz.s(s(z)))(\lambda sz.s(z)))$
$\equiv (\lambda pz.z((\lambda wyx.y(wyx))((\lambda a.a(\lambda sz.s(s(z)))(\lambda sz.s(z)))T))((\lambda a.a(\lambda sz.s(s(z)))(\lambda sz.s(z)))T))$
$\equiv (\lambda pz.z((\lambda wyx.y(wyx))((T(\lambda sz.s(s(z)))(\lambda sz.s(z)))))((T(\lambda sz.s(s(z)))(\lambda sz.s(z)))))$
$\equiv (\lambda pz.z((\lambda wyx.y(wyx))(\lambda sz.s(s(z))))((\lambda sz.s(s(z)))))$
$\equiv (\lambda pz.z(\lambda sz.s(s(s(z))))(\lambda sz.s(s(z))))$
$\equiv (Pair32)$

The predecessor of a number n is obtained by applying n times the function Φ to the pair ($\lambda z.z00$ and then selecting the second member of the pair.

$$Second \equiv \lambda p.pFalse$$
$$P \equiv (\lambda n.\text{ second } (n\Phi(\lambda z.z00)))$$

An example:

$$\begin{aligned} P3 \equiv & (\lambda n.\text{ second } (n\Phi(\lambda z.z00)))3 \\ \equiv & \text{ second } (3\Phi(\lambda z.z00)) \\ \equiv & \text{ second } (3\Phi pair00) \\ \equiv & \text{ second } (2\Phi pair10) \\ \equiv & \text{ second } (1\Phi pair21) \\ \equiv & \text{ second } (0\Phi pair32) \\ \equiv & \text{ second } (pair32) \\ \equiv & 2 \end{aligned}$$

16.1.6 Equality and Inequality

With the predecessor function as the building block, we can now define a function which tests if a number x is greater than or equal to a number y

$$G \equiv (\lambda xy.Z(xPy))$$

In which if the predecessor function applied x times to y yields zero, then it is true that $x \geq y$.

An example:

$$\begin{aligned} G23 \equiv & (\lambda xy.Z(xPy))23 \\ \equiv & Z(2P3) \\ \equiv & Z(1(P2)) \\ \equiv & Z(0(1)) \\ \equiv & Z(1) \\ \equiv & Flase \end{aligned}$$

The following definition of the function E which tests if two numbers are equal

$$E \equiv (\lambda xy. \wedge (Z(xPy))(Z(yPx)))$$

An example:

$$\begin{aligned} E23 \equiv & (\lambda xy. \wedge (Z(xPy))(Z(yPx)))23 \\ \equiv & (\wedge(Z(2P3))(Z(3P2))) \\ \equiv & (\wedge(Z(1(P2)))(Z(2P1))) \\ \equiv & (\wedge(Z(1))(Z(1P0))) \\ \equiv & (\wedge(Z(1))(Z(0))) \\ \equiv & (\wedge(False)(True)) \\ \equiv & False \end{aligned}$$

The following definition of the function LT which tests if x is less than y

$$LE \equiv (\lambda xy. \wedge (Z(yPx))(\neg Exy)$$

The philosophy of this equation is that if y $\geq$ x and also x $\neq$ y, x is less than y.
An example:

$$\begin{aligned}
LT23 \equiv & (\lambda xy. \wedge (Z(yPx))(\neg Exy)23 \\
\equiv & (\wedge(Z(3P2))(\neg E23) \\
\equiv & (\wedge(Z(2P1))(\neg E23) \\
\equiv & (\wedge(Z(1P0))(\neg E23) \\
\equiv & (\wedge(Z0)(\neg E23) \\
\equiv & (\wedge(True)(\neg False) \\
\equiv & (\wedge(True)(True) \\
\equiv & True
\end{aligned}$$

The following definition of the function LT which tests if x is greater than y

$$GT \equiv (\lambda xy. \wedge (Z(xPy))(\neg Exy)$$

The philosophy of this equation is that if x $\geq$ y and also x $\neq$ y, x is greater than y.
An example:

$$\begin{aligned}
GT23 \equiv & (\lambda xy. \wedge (Z(xPy))(\neg Exy)23 \\
\equiv & (\wedge(Z(2P3))(\neg E23) \\
\equiv & (\wedge(Z(1P2))(\neg E23) \\
\equiv & (\wedge(Z(1))(\neg E23) \\
\equiv & (\wedge(False)(\neg False) \\
\equiv & (\wedge(False)(True) \\
\equiv & False
\end{aligned}$$

16.1.7 Recursion

Recursive functions can be defined using a function Y which calls a function y and then regenerates itself.

$$Y \equiv (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx)))$$

This function applied to a function R yields:

$$\begin{aligned} YR \equiv & (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx))R \\ \equiv & (\lambda x.R(xx))(\lambda x.R(xx)) \\ \equiv & (\lambda x.R(xx))(\lambda x.R(xx)) \\ \equiv & R((\lambda x.R(xx))(\lambda x.R(xx))) \end{aligned}$$

YR can be further reduced to R(YR).

An example:
A function adds up the first n natural numbers using recursion:

$$\begin{aligned} R \equiv & \sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} \\ \equiv & (\lambda rn.Zn0(nS(r(Pn)))) \end{aligned}$$

An example of combination of both:

$$\begin{aligned} YR3 \equiv & R(YR)3 \\ \equiv & Z30(3S(YR(P3))) \\ \equiv & F0(3S(YR(P3))) \\ \equiv & (3S(YR(P3))) \\ \equiv & (3S(YR(2))) \\ \equiv & (3S(R(YR)2)) \\ \equiv & (3S(Z20(2S(YR(P2))))) \\ \equiv & (3S(F0(2S(YR(P2))))) \\ \equiv & (3S(2S(R(YR)1))) \\ \equiv & (3S(2S(1S(YR(P0))))) \\ \equiv & (3S(2S(1S(R(YR)0)))) \\ \equiv & (3S(2S(1S(Z00(0S(YR(P0))))))) \\ \equiv & (3S(2S(1S(T0(0S(YR(P0))))))) \\ \equiv & (3S(2S(1S0))) \\ \equiv & 6 \end{aligned}$$

16.1.8 Extra Exercise

Define the factorial function recursively.

$$fac \equiv (\lambda rn.Zn1(Mn(r(Pn))))$$

An example:

$$\begin{aligned}
YR3 \equiv & R(YR)3 \\
\equiv & Z31(M3(YR(P3))) \\
\equiv & F1(M3(YR(P3))) \\
\equiv & M3(YR(P3)) \\
\equiv & M3(YR2) \\
\equiv & M3(R(YR)2) \\
\equiv & M3(z21(M2(YR(P2)))) \\
\equiv & M3(F1(M2(YR(P2)))) \\
\equiv & M3(M2(YR1)) \\
\equiv & M3(M2(R(YR)1)) \\
\equiv & M3(M2(M1(R(YR)0))) \\
\equiv & M3(M2(M1(Z01(M0(YR(P0)))))) \\
\equiv & M3(M2(M1(Z01(M0(YR(P0)))))) \\
\equiv & M3(M2(M1(1))) \\
\equiv & 6
\end{aligned}$$

Define a data structure to represent a list of numbers.
A list required following operations:

Function	Description
nil	Construct an empty list.
isnil	Test if list is empty.
cons	Prepend an element to a (possibly empty) list
head	Get the first element of the list
Tail	Get the rest of the list

$$
\begin{aligned}
cons \equiv & pair \equiv \lambda z.zab \\
head \equiv & first \equiv \lambda xy.x \\
tail \equiv & second \equiv \lambda xy.y \\
nil \equiv & false \equiv \lambda xy.y \\
isnil \equiv & \lambda l.l(\lambda htd.false)true
\end{aligned}
$$

An example :

$$[1,2,3] \equiv cons(1, cons(2, cons(3, nil)))$$

isnil a non-empty list:

$$
\begin{aligned}
\text{isnil } [1,2,3] \equiv & isnil(cons(1, cons(2, cons(3, nil)))) \\
\equiv & (\lambda l.l(\lambda htd.false)true)(\lambda z.z1(\lambda z.z2(\lambda z.z3(nil)))) \\
\equiv & (\lambda z.z1(\lambda z.z2(\lambda z.z3(nil))))(\lambda htd.false)true \\
\equiv & (\lambda htd.false)1(\lambda z.z2(\lambda z.z3(nil)))true \\
\equiv & false
\end{aligned}
$$

isnil an empty list:

$$
\begin{aligned}
isnilnil \equiv & (\lambda l.l(\lambda htd.false)true)(\lambda xy.y) \\
\equiv & (\lambda xy.y)(\lambda htd.false)true \\
\equiv & true
\end{aligned}
$$
