Chapter 1

All You Need is Lambda

> Even the greatest mathematicians, the ones that we would put into our mythology of great mathematicians, had to do a great deal of leg work in order to get to the solution in the end.
>
> Daniel Tammett

1.1 All You Need is Lambda

This chapter provides a very brief introduction to the lambda calculus, a model of computation devised in the 1930s by Alonzo Church. A calculus is a method of calculation or reasoning; 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.

You may be wondering where the Haskell is. You may be contemplating skipping this chapter. You may feel tempted to skip ahead to the fun stuff when we build a project.

DON'T.

We're starting from first principles here so that when we get around to building projects you know what you're doing. You don't start building a house from the attic down; you start from the foundation. Lambda calculus is your foundation, because Haskell is a lambda calculus.

1.2 What is functional programming?

Functional programming is a computer programming paradigm that relies on functions modeled on mathematical functions. 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 an argument or input, and once applied, can be *reduced* or *evaluated*. In Haskell, and in functional programming more generally, functions are *first-class*: they can be used as values or passed as arguments, or inputs, to yet more functions. We'll define these terms more carefully as we progress through the chapter.

Functional programming languages are all based on the lambda calculus. Some languages in this general category incorporate features into the language that aren't translatable into lambda expressions. Haskell is a *pure* functional language, because it does not. We'll address this notion of purity more later in the book, but it isn't a judgment of the moral worth of other languages.

The word *purity* in functional programming is sometimes also used to mean what is more properly called *referential transparency*. Referential transparency means that the same function, 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. Haskell programs are built from separate, independent functions, kind of like LEGO®: the functions are bricks that can be

assembled and reassembled.

These features also make Haskell's syntax rather minimalist, as you'll soon see.

1.3 What is a function?

If we step back from using the word "lambda," you most likely already know what a function is. A function is a relation between a set of possible inputs and a set of possible outputs. The function itself defines and represents the 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.

For example, let's imagine a function named f that defines the following relations where the first value is the input and the second is the output:

$$f(1) = A$$
$$f(2) = B$$
$$f(3) = C$$

The input set is $\{1, 2, 3\}$ and the output set is $\{A, B, C\}$.[1] A crucial point about how these relations are defined: our hypothetical function will *always* return the value A given the input 1 — no exceptions!

[1] For those who would like precise terminology, the input set is known as the domain. The set of possible outputs for the function is called the codomain. All domains and

In contrast, the following is *not* a valid function:

$$f(1) = X$$
$$f(1) = Y$$
$$f(2) = Z$$

This gets back to the referential transparency we mentioned earlier: given the same input, the output should be predictable.

Is the following function valid?

$$f(1) = A$$
$$f(2) = A$$
$$f(3) = A$$

Yes, having the same output for more than one input is valid. Imagine, for example, that you need a function that tests a positive integer for being less than 10. You'd want it to return `True` when the input was less than 10 and `False` for all other cases. In that case, several different inputs will result in the output `True`; many more will give a result of `False`. Different inputs can lead to the same output.

codomains are sets of unique values. The subset of the codomain that contains possible outputs related to different inputs is known as the image. The mapping between the domain and the image or codomain need not be one-to-one; in some cases, multiple input values will map to the same value in the image, as when a function returns either 'true' or 'false' so that many different inputs map to each of those output values. However, a given input should not map to multiple outputs.

What matters here is that 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.

In the above examples, we didn't demonstrate a relationship between the inputs and outputs. Let's look at an example that does define the relationship. This function is again named f:

$$f(x) = x + 1$$

This function takes one argument, which we have named x. The relationship between the input, x, and the output is described in the function body. It will add 1 to whatever value x is and return that result. When we apply this function to a value, such as 1, we substitute the value in for x:

$$f(1) = 1 + 1$$

f applied to 1 equals 1 + 1. That tells us how to map the input to an output: 1 added to 1 becomes 2:

$$f(1) = 2$$

Understanding functions in this way — as a mapping of a set of inputs to a set of outputs — is crucial to understanding functional programming.

1.4 The structure of lambda terms

The lambda calculus has three basic components, or *lambda terms*: 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. The simplest expression is a single variable. Variables here have no meaning or value; they are only names for potential inputs to functions.

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 *argument* is an input value.

Abstractions consist of two parts: the *head* and the *body*. The head of the function is a λ (lambda) followed by a variable name. The body of the function is another expression. So, a simple function might look like this:

$$\lambda x.x$$

The variable named in the head is the *parameter* and *binds* all instances of that same variable in the body of the function. That means, when we apply this function to an argument, each x in the body of the function will have the value of that argument. We'll demonstrate this in the next section.

In the previous section, we were talking about functions called f, but 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.

Let's break down the basic structure:

```
λ x . x
^─┬─^
  └──── extent of the head of the lambda.

λ x . x
  ^──── the single parameter of the
        function. This binds any
        variables with the same name
        in the body of the function.


λ x . x
      ^── body, the expression the lambda
          returns when applied. This is a
          bound variable.
```

The dot (.) separates the parameters of the lambda from the function body.

The abstraction as a whole has no name, but the reason we call it an *abstraction* is that it is a generalization, or abstraction, from a concrete instance of a problem, and it abstracts through the introduction of names. The names stand for concrete values, but by using named variables, we allow for the

possibility of applying the general function to different values (or, perhaps even values of different *types*, as we'll see later). When we apply the abstraction to arguments, we replace the names with values, making it concrete.

Alpha equivalence

Often when people express this function in lambda calculus you'll see something like

$$\lambda x.x$$

The variable x here is not semantically meaningful except in its role in that single expression. Because of this, there's a form of equivalence between lambda terms called *alpha equivalence*. This is a way of saying that:

$$\lambda x.x$$

$$\lambda d.d$$

$$\lambda z.z$$

all mean the same thing. They're all the same function.

Let's look next at what happens when we apply this abstraction to a value.

1.5 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. You also eliminate the head of the abstraction, since its only purpose was to bind a variable. This process is called *beta reduction*.

Let's use the function we had above:

$$\lambda x.x$$

We'll do our first beta reduction using a number.[2] We apply the function above to 2, substitute 2 for each bound variable in the body of the function, and eliminate the head:

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

The only bound variable is the single x, so applying this function to 2 returns 2. This function is the *identity* function.[3] All it does is accept a single argument x and return that same argument. The x has no inherent meaning, but, because it is bound in the head of this function, when the function is

[2]The lambda calculus can derive numbers from lambda abstractions, rather than using the numerals we are familiar with, but the applications can become quite cumbersome and difficult to read.

applied to an argument, all instances of x within the function body must have the same value.

Let's use an example that mixes some arithmetic into our lambda calculus. We use the parentheses here to clarify that the body expression is $x+1$. In other words, we are not applying the function to the 1:

$$(\lambda x.x + 1)$$

What is the result if we apply this abstraction to 2? How about to 10?

Beta reduction is this 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 you the function has been applied.

We can also apply our identity function to another lambda abstraction:

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

In this case, we'd substitute the entire abstraction in for x. We'll use a new syntax here, $[x := z]$, to indicate that z will be substituted for all occurrences of x (here z is the function $\lambda y.y$). We reduce this application like this:

[3] Note that this is the same as the identity function in mathematical notation: $f(x) = x$. One difference is that $f(x) = x$ is a declaration involving a function named f while the above lambda abstraction *is* a function.

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

Our final result is another identity function. There is no argument to apply it to, so we have nothing to reduce.

Once more, but this time we'll add another argument:

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

Applications in the lambda calculus are *left associative*. That is, unless specific parentheses suggest otherwise, they associate, or group, to the left. So, this:

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

can be rewritten as:

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

Onward with the reduction:

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

We can't reduce this any further as there is nothing left to apply, and we know nothing about z.

We'll look at functions below that have multiple heads and also *free variables* (that is, variables in the body that are not bound by the head), but the basic process will remain the same. The process of beta reduction stops when there are either no more heads, or lambdas, left to apply or no more arguments to apply functions to. A computation therefore consists of an initial lambda expression (or two, if you want to separate the function and its input) 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 in for bound variables until there are no more heads left to evaluate or no more arguments to apply them to.

Free variables

The purpose of the head of the function is to tell us which variables to replace when we apply our function, that is, to bind the variables. A bound variable must have the same value throughout the expression.

But sometimes the body expression has variables that are not named in the head. We call those variables *free variables*. In the following expression:

$$\lambda x.xy$$

The 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.

That whole abstraction can be applied to an argument, z, like this: $(\lambda x.xy)z$. We'll show an intermediate step, using the := syntax we introduced above, that most lambda calculus literature does not show:

1. $(\lambda x.xy)z$
   We apply the lambda to the argument z.

2. $(\lambda[x := z].xy)$
   Since x is the bound variable, all instances of x in the body of the function will be replaced with z. The head will be eliminated, and we replace any x in the body with a z.

3. zy
   The head has been applied away, and there are no more heads or bound variables. Since we know nothing about z or y, we can reduce this no further.

Note that alpha equivalence does not apply to free variables. That is, $\lambda x.xz$ and $\lambda x.xy$ are not equivalent because z and y might be different things. However, $\lambda xy.yx$ and $\lambda ab.ba$

are equivalent due to alpha equivalence, as are $\lambda x.xz$ and $\lambda y.yz$ because the free variable is left alone.

1.6 Multiple arguments

Each lambda can only bind one parameter and can only accept one argument. Functions that require multiple arguments have multiple, nested heads. When you apply it once and eliminate the first (leftmost) head, the next one is applied and so on. This formulation was originally discovered by Moses Schönfinkel in the 1920s but was later rediscovered and named after Haskell Curry and is commonly called *currying*.

What we mean by this description is that the following:

$$\lambda xy.xy$$

is a convenient shorthand for two nested lambdas (one for each argument, x and y):

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

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.

To try to make this a little more concrete, let's suppose that we apply these lambdas to specific values. First, a simple example with the identity function:

1. $\lambda x.x$
2. $(\lambda x.x)\ 1$
3. $[x := 1]$
4. 1

Now let's look at a "multiple" argument lambda:

1. $\lambda xy.xy$
2. $(\lambda xy.xy)\ 1\ 2$
3. $(\lambda x.(\lambda y.xy))\ 1\ 2$
4. $[x := 1]$
5. $(\lambda y.1y)\ 2$
6. $[y := 2]$
7. $1\ 2$

That wasn't too interesting because it's like nested identity functions! We can't meaningfully apply a 1 to a 2. Let's try something different:

1. $\lambda xy.xy$
2. $(\lambda xy.xy)(\lambda z.a)\ 1$

3. $(\lambda x.(\lambda y.xy))(\lambda z.a)\ 1$

4. $[x := (\lambda z.a)]$

5. $(\lambda y.(\lambda z.a)y)\ 1$

6. $[y := 1]$

7. $(\lambda z.a)\ 1$ We still can apply this one more time.

8. $[z := 1]$ But there is no z in the body of the function, so there is nowhere to put a 1. We eliminate the head, and the final result is

9. a

It's more common in academic lambda calculus materials to refer to abstract variables rather than concrete values. The process of beta reduction is the same, regardless. The lambda calculus is a process or method, like a game with a few simple rules for transforming lambdas, but no specific meaning. We've introduced concrete values to make the reduction somewhat easier to see.

The next example uses only abstract variables. Due to alpha equivalence, you sometimes see expressions in lambda calculus literature such as:

$$(\lambda xy.xxy)(\lambda x.xy)(\lambda x.xz)$$

The substitution process can become a tangle of xs that are not the same x because each was bound by a different head. To help make the reduction easier to read we're going to use different variables in each abstraction, but it's worth emphasizing that the name of the variable (the letter) has no meaning or significance:

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

2. $(\lambda x.\lambda y.\lambda z.xz(yz))(\lambda m.\lambda n.m)(\lambda p.p)$
   We've not reduced or applied anything here, but made the currying explicit.

3. $(\lambda y.\lambda z.(\lambda m.\lambda n.m)z(yz))(\lambda p.p)$
   Our first reduction step was to apply the outermost lambda, which was binding the x, to the first argument, $(\lambda m.\lambda n.m)$.

4. $\lambda z.(\lambda m.\lambda n.m)(z)((\lambda p.p)z)$
   We applied the y and replaced the single occurrence of y with the next argument, the term $\lambda p.p$. The outermost lambda binding z is, at this point, irreducible because it has no argument to apply to. What remains is to go inside the terms one layer at a time until we find something reducible.

5. $\lambda z.(\lambda n.z)((\lambda p.p)z)$
   We can apply the lambda binding m to the argument z. We keep searching for terms we can apply. The next thing

we can apply is the lambda binding n to the lambda term $((\lambda p.p)z)$.

6. $\lambda z.z$
   In the final step, the reduction takes a turn that might look slightly odd. Here the outermost, leftmost *reducible* term is $\lambda n.z$ applied to the entirety of $((\lambda p.p)z)$. As we saw in an example above, it doesn't matter what n got bound to, $\lambda n.z$ unconditionally tosses the argument and returns z. So, we are left with an irreducible lambda expression.

Intermission: Equivalence Exercises

We'll give you a lambda expression. Keeping in mind both alpha equivalence and how multiple heads are nested, choose an answer that is equivalent to the listed lambda term.

1. $\lambda xy.xz$

   a) $\lambda xz.xz$

   b) $\lambda mn.mz$

   c) $\lambda z.(\lambda x.xz)$

2. $\lambda xy.xxy$

   a) $\lambda mn.mnp$

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

c) $\lambda a.(\lambda b.aab)$

3. $\lambda xyz.zx$

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

b) $\lambda tos.st$

c) $\lambda mnp.mn$

1.7 Evaluation is simplification

There are multiple normal forms in lambda calculus, but here when we refer to normal form we mean *beta normal form*. Beta normal form is when you cannot beta reduce (apply lambdas to arguments) the terms any further. This corresponds to a fully evaluated expression, or, in programming, a fully executed program. This is important to know so that you know when you're done evaluating an expression. It's also valuable to have an appreciation for evaluation as a form of simplification when you get to the Haskell code as well.

Don't be intimidated by calling the reduced form of an expression its normal form. When you want to say "2," do you say $2000/1000$ each time or do you say 2? The expression $2000/1000$ is not fully evaluated because the division function has been fully applied (two arguments), so it could be reduced, or evaluated. In other words, there's a simpler form it can be

reduced to — the number two. The normal form, therefore, is 2.

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

Similarly, the normal form of the following is 600:

$$(10 + 2) * 100/2$$

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.

The identity function, $\lambda x.x$, is fully reduced (that is, in normal form) because it hasn't yet been applied to anything. However, $(\lambda x.x)z$ is ***not*** in beta normal form because the identity function hasn't been applied to a free variable z and hasn't been reduced. If we did reduce it, the final result, in beta normal form, would be z.

1.8 Combinators

A combinator is a lambda term with no free variables. Combinators, as the name suggests, serve only to *combine* the arguments they are given.

So the following are combinators because every term in the body occurs in the head:

1. $\lambda x.x$
   x is the only variable and is bound because it is bound by the enclosing lambda.

2. $\lambda xy.x$

3. $\lambda xyz.xz(yz)$

And the following are not because there's one or more free variables:

1. $\lambda y.x$
   Here y is bound (it occurs in the head of the lambda) but x is free.

2. $\lambda x.xz$
   x is bound and is used in the body, but z is free.

We won't have a lot to say about combinators per se. The point is to call out a special class of lambda expressions that can *only* combine the arguments it is given, without injecting any new values or random data.

1.9 Divergence

Not all reducible lambda terms reduce neatly to a beta normal form. This isn't because they're already fully reduced, but rather because they *diverge*. Divergence here means that the reduction process never terminates or ends. Reducing terms should ordinarily *converge* to beta normal form, and divergence is the opposite of convergence, or normal form. Here's an example of a lambda term called *omega* that diverges:

1. $(\lambda x.xx)(\lambda x.xx)$
   x in the first lambda's head becomes the second lambda

2. $([x := (\lambda x.xx)]xx)$
   Using $[var := expr]$ to denote what x has been bound to.

3. $(\lambda x.xx)(\lambda x.xx)$
   Substituting $(\lambda x.xx)$ for each occurence of x. We're back to where we started and this reduction process never ends — we can say omega diverges.

This matters in programming because terms that diverge are terms that don't produce an *answer* or meaningful result. Understanding what will terminate means understanding what programs will do useful work and return the answer we want. We'll cover this idea more later.

1.10 Summary

The main points you should take away from this chapter are:

- Functional programming is based on expressions that include variables or constant values, expressions combined with other expressions, and functions.
- Functions have a head and a body and are those expressions that can be applied to arguments and reduced, or evaluated, to a result.
- Variables may be bound in the function declaration, and every time a bound variable shows up in a function, it has the same value.
- All functions take one argument and return one result.
- Functions are a mapping of a set of inputs to a set of outputs. Given the same input, they always return the same result.

These things all apply to Haskell, as they do to any pure functional languages, because semantically Haskell is a lambda calculus. 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 programs is centered around evaluating expressions rather than executing instructions, although Haskell has a way to execute instructions, too. We will still be making reference to the lambda calculus when we write about all the later, apparently very complex topics: function composition, monads, parser combinators. Don't worry if you don't know those words yet. If you understood this chapter, you have the foundation you need to understand them all.

1.11 Chapter Exercises

We're going to do the following exercises a bit differently than what you'll see in the rest of the book, as we will be providing some answers and explanations for the questions below.

Combinators Determine if each of the following are combinators or not.

1. $\lambda x.xxx$
2. $\lambda xy.zx$
3. $\lambda xyz.xy(zx)$
4. $\lambda xyz.xy(zxy)$
5. $\lambda xy.xy(zxy)$

Normal form or diverge? Determine if each of the following can be reduced to a normal form or if they diverge.

1. $\lambda x.xxx$
2. $(\lambda z.zz)(\lambda y.yy)$
3. $(\lambda x.xxx)z$

Beta reduce Evaluate (that is, beta reduce) each of the following expressions to normal form. We *strongly* recommend writing out the steps on paper with a pencil or pen.

1. $(\lambda abc.cba)zz(\lambda wv.w)$
2. $(\lambda x.\lambda y.xyy)(\lambda a.a)b$
3. $(\lambda y.y)(\lambda x.xx)(\lambda z.zq)$
4. $(\lambda z.z)(\lambda z.zz)(\lambda z.zy)$
   Hint: alpha equivalence.
5. $(\lambda x.\lambda y.xyy)(\lambda y.y)y$
6. $(\lambda a.aa)(\lambda b.ba)c$
7. $(\lambda xyz.xz(yz))(\lambda x.z)(\lambda x.a)$

1.12 Answers

Please note: At this time, this is the only chapter in the book for which we have provided answers. We provide them here due to the importance of being able to check your understanding of this material and the relative difficulty of checking answers that you probably wrote by hand in a notebook.

Equivalence Exercises

1. b
2. c
3. b

Combinators

1. $\lambda x.xxx$ is indeed a combinator, it refers only to the variable x which is introduced as an argument.
2. $\lambda xy.zx$ is not a combinator, the variable z was not introduced as an argument and is thus a free variable.
3. $\lambda xyz.xy(zx)$ is a combinator, all terms are bound. The head is $\lambda xyz.$ and the body is $xy(zx)$. None of the arguments in the head have been applied so it's irreducible. The variables x, y, and z are all bound in the head and are not free.

This makes the lambda a combinator - no occurrences of free variables.

4. $\lambda xyz.xy(zxy)$ is a combinator. The lambda has the head $\lambda xyz.$ and the body: $xy(zxy)$. Again, none of the arguments have been applied so it's irreducible. All that is different is that the bound variable y is referenced twice rather than once. There are still no free variables so this is also a combinator.

5. $\lambda xy.xy(zxy)$ is not a combinator, z is free. Note that z isn't bound in the head.

Normal form or diverge?

1. $\lambda x.xxx$ doesn't diverge, has no further reduction steps. If it had been applied to itself, it *would* diverge, but by itself does not as it is already in normal form.

2. $(\lambda z.zz)(\lambda y.yy)$ diverges, it never reaches a point where the reduction is done. This is the *omega* term we showed you earlier, with different names for the bindings. It's *alpha equivalent* to $(\lambda x.xx)(\lambda x.xx)$.

3. $(\lambda x.xxx)z$ doesn't diverge, it reduces to zzz.

Beta reduce The following are evaluated in *normal order*, which is where terms in the outer-most and left-most positions get

evaluated (applied) first. This means that if all terms are in the outermost position (none are nested), then it's left-to-right application order.

1. $(\lambda abc.cba)zz(\lambda wv.w)$
   $(\lambda a.\lambda b.\lambda c.cba)(z)z(\lambda w.\lambda v.w)$
   $(\lambda b.\lambda c.cbz)(z)(\lambda w.\lambda v.w)$
   $(\lambda c.czz)(\lambda w.\lambda v.w)$
   $(\lambda w.\lambda v.w)(z)z$
   $(\lambda v.z)(z)$
   z

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

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

4. $(\lambda z.z)(\lambda z.zz)(\lambda z.zy)$
   $(\lambda z.zz)(\lambda z.zy)$
   $(\lambda z.zy)(\lambda z.zy)$
   $(\lambda z.zy)(y)$
   yy

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

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

7. Steps we took

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

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

   c) $(\lambda y.\lambda z1(\lambda x.z)z1(yz1))(\lambda x.a)$

   d) $(\lambda z1.(\lambda x.z)(z1)((\lambda x.a)z1))$

   e) $(\lambda z1.z((\lambda x.a)(z1)))$

   f) $(\lambda z1.za)$ The $z1$ 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.

   How we got there, step by step

   a) Our expression we'll reduce.

   b) Add the implied lambdas to introduce each argument.

c) Apply the leftmost x and bind it to $(\lambda x.z)$, rename leftmost z to $z1$ for clarity to avoid confusion with the other z. Hereafter, "z" is exclusively the z in $(\lambda x.z)$.

d) Apply y, it gets bound to $(\lambda x.a)$.

e) Can't apply z1 to anything, evaluation strategy is normal order so leftmost outermost is the order of the day. Our leftmost, outermost lambda has no remaining arguments to be applied so we now examine the terms nested within to see if they are in normal form. $(\lambda x.z)$ gets applied to $z1$, tosses the $z1$ away and returns z. z is now being applied to $((\lambda x.a)(z1))$.

f) 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 applied to $z1$, but tosses it away and returns the free variable a. The a is now part of the body of that expression. All of our terms are in normal order now.

1.13 Definitions

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

2. 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 we applied the lambda to.

3. *Application* is how one evaluates or reduces lambdas, this binds the argument to whatever the lambda was applied to. Computations are performed in lambda calculus by applying lambdas to arguments until you run out of arguments to apply lambdas to.

   $(\lambda x.x)1$

   This example reduces to 1, the identity $\lambda x.x$ was applied to the value 1, x was bound to 1, and the lambda's body is x, so it just kicks the 1 out. In a sense, applying the $\lambda x.x$ *consumed* it. We *reduced* the amount of structure we had.

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

5. *Normal order* is a common evaluation strategy in lambda calculi. Normal order means evaluating (ie, applying or beta reducing) the leftmost outermost lambdas first, evaluating terms nested within after you've run out of arguments to apply. Normal order isn't how Haskell code is evaluated - it's *call-by-need* instead. We'll explain this more

later. Answers to the evaluation exercises were written in normal order.

1.14 Follow-up resources

These are *optional* and intended only to offer suggestions on how you might deepen your understanding of the preceding topic. Ordered approximately from most approachable to most thorough.

1. Raul Rojas. A Tutorial Introduction to the Lambda Calculus
   `http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf`
2. Henk Barendregt; Erik Barendsen. Introduction to Lambda Calculus
   `http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf`
3. Jean-Yves Girard; P. Taylor; Yves Lafon. Proofs and Types
   `http://www.paultaylor.eu/stable/prot.pdf`