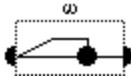


To Dissect a Mockingbird: A Graphical Notation for the Lambda Calculus with Animated Reduction



David C Keenan, 27-Aug-1996

last updated 1-Apr-2014

116 Bowman Parade, Bardon QLD 4065, Australia

d.keenan7@gmail.com

<http://dkeenan.com>

Abstract

The lambda calculus, and the closely related theory of combinators, are important in the foundations of mathematics, logic and computer science. This paper provides an informal and entertaining introduction by means of an animated graphical notation.

Introduction

In the 1930s and 40s, around the birth of the "automatic computer", mathematicians wanted to formalise what we mean when we say some result or some function is "effectively computable", whether by machine or human. A "computer", originally, was a *person* who performed arithmetic calculations. The "effectively" part is included to indicate that we are not concerned with the time any particular computer might take to produce the result, so long as it would get there eventually. They wanted to find the simplest possible system that could be said to compute.

Several such systems were invented and for the most part looked entirely unlike each other. Remarkably, they were all eventually shown to be equivalent in the sense that any one could be made to behave like the others. Today, the best known of these are the Turing Machine, of the British mathematician Alan Turing (not to be confused with his touring machine, the bicycle of which he was so fond), and the Lambda Calculus, of the American logician [Alonzo Church \(1941\)](#). The Turing machine is reflected in the Von Neumann machine which describes the general form of most computing hardware today. The Lambda calculus is reflected in the programming language Lisp and its relatives which are called "functional" or "applicative" languages.

The Turing machine is based on the idea of a tape of unbounded length, with a head which can move left or right along the tape reading and writing symbols. These symbols can be interpreted as instructions (to move left or right or read or write symbols) but may also be interpreted as data, particularly when it comes time to read the result.

The lambda calculus is based on the more abstract notion of "applying a function". For example we may write $y := 2x+3$ to describe how to obtain y given x , but suppose we want to describe, in the abstract, what it is we are doing to x , so that we can do it to other things. We might make up a name for it, say "DoubleAndAddThree" or just " f ", and write $f(a) = 2a+3$ to describe what we mean. But this is all rather indirect and it would be a nuisance to have to keep making up new names and remembering what they meant. Church used the greek letter lambda and a dot and showed that we could simply write $\lambda a.2a+3$ as a name for the function. This operation is called *function abstraction*. We could, of course, have used any symbol we like in place of a . We can then *apply* this function to anything we like, for example z^2 , and we find that $(\lambda a.2a+3)(z^2)$ simplifies to $2z^2+3$. We see that

function abstraction and function application are inverses. Note that we must now be careful to use an explicit multiplication sign where juxtaposition might be wrongly interpreted as function application.

We might want to express functions of more than one argument, such as addition $\lambda ab.a+b$, but we can always express these as functions of one argument which return another function, e.g. $\lambda a.(\lambda b.a+b)$. This transformation is called *Currying*, after Haskell Curry. If we apply this to the number 5 for example, we obtain $\lambda b.5+b$ which may then be applied to a second number. We can think of $\lambda ab.a$ as shorthand for $\lambda a.(\lambda b.a)$.

You may say, "So what?" All this seems to be making things more complicated not less. For example, why is " $\lambda ab.a+b$ " any better than plain old "+"?

What is remarkable about the lambda calculus is that one doesn't need anything as crude, gross or substantial as the digits 0 and 1 or operators like + and × in order to do arithmetic and logic, or indeed any kind of computation.

All we need is function abstraction and application. "Abstracted from what and applied to what?", you may ask. Other such functions! Actually when they are not applied to anything except each other we refer to them as *combinators* rather than functions. This is called the *pure lambda calculus*.

Since there is no confusion with multiplication in the pure lambda calculus, we can omit some parentheses and $f(x)$ may be written simply as fx . Application is taken to be left associative so that $(f(x))(y)$ or $(fx)y$ may be written as just fy .

We can write lambda expressions like $\lambda a.a$ the identity combinator, or $\lambda a.aa$ the combinator that applies any combinator to itself, or $\lambda ab.ba$ the combinator that reverses the order of application of two combinators.

When I first understood the lambda calculus I felt that those arbitrary bound variables, like a,b,f and x above, just served to obscure what was really going on. They are only necessitated by the constraint of writing everything as a one-dimensional string of symbols. So I am going to introduce you to some of the strange characters that inhabit the world of combinators by using a two-dimensional notation that I developed. This notation started from the idea that, instead of the bound variables we could draw arrows connecting each lambda symbol to blank spaces in the expression where its variable would have appeared.

As pointed out by Raymond Smullyan the eminent logician and author of several puzzle books, the theory of combinators is an abstract science dealing with objects whose only important property is how they act upon each other. We are free to choose other properties of these objects in any way we like. In his delightful book *To mock a mockingbird*, [Smullyan \(1985\)](#) chooses *birds* for his combinators, in memory of [Haskell Curry](#), an early pioneer in the theory of combinators ([1958](#)) and an avid bird-watcher.

The story begins,

"A certain enchanted forest is inhabited by talking birds. Given any birds A and B, if you call out the name of B to A, then A will respond by calling out the name of some bird to you; this bird we designate AB. Thus AB is the bird named by A upon hearing the name of B."

Smullyan also notes in his preface,

"This remarkable subject is currently playing an important role in computer science and artificial intelligence. Despite the profundity of the subject, it is no more difficult to learn than high school algebra or geometry."

In the hope of making it even easier I introduce the following graphical notation by extending Smullyan's bird metaphor. We will go inside the birds' heads and see how to draw diagrams of the internal plumbing which connects their ear to their throat in order to produce the correct song in response to each song that they hear. In other words we will draw maps of their brains.

This notation was developed as part of an attempt to place the lambda calculus on an even deeper foundation.

Both the attempt (so far unsuccessful) and the notation were inspired by [George Spencer-Brown's \(1969\)](#) controversial book *Laws of Form*.

"The theme of this book is that a universe comes into being when a space is severed or taken apart. The act is itself already remembered, even if unconsciously, as our first attempt to distinguish different things in a world where, in the first place, the boundaries can be drawn anywhere we please. At this stage the universe cannot be distinguished from how we act upon it, and the world may seem like shifting sand beneath our feet."

It is a wonderfully bizarre fact that each song of a combinatory bird is not merely the *name* of another bird but is actually a complete description of the internal plumbing of that other bird. That is, each song is actually a brain map of some bird. Since a song is a complete description of how some bird will respond when it hears another bird, and the *only* important thing about a combinatory bird is how it responds when it hears another bird, we see that songs and singers are interchangeable. So we can say that the birds sing *birds* to each other, or we can equally say that what we have is a bunch of *songs* that sing songs to each other! Combinatory birds exist at an almost mystical level. Their language has no distinction between verbs and nouns. A description of action can equally well be a name. To emphasise this, in future we will call our diagrams *song maps*.

Unfortunately, although we have these diagrams, and the purely textual notation of the lambda calculus, we no longer know how to turn them into music. However I am confident that some day soon some brilliant mathematical musician or musical mathematician (or mathical musimatician?) will rediscover this ability. In the meantime we must be content to appreciate the rhyme and rhythm of their graphical forms. It is something of a mystery as to why the textual notation is called the lambda calculus, since surely any other letter of the Greek alphabet would have done just as well. One suggestion is that these songs are actually quite suitable for dancing and that "lambda" is in fact a corruption of "lambada".

[Carol Hindley \(1986\)](#) has given some marvellous drawings of the *outsides* of several well known combinatorics in her hilarious note "Care of Your Pet Combinator". Here we find that they bear somewhat more resemblance to insects and reptiles than to conventional birds.

One layer birds

The simplest of these birds is called the Identity bird since its response to hearing the name of any bird is that same (identical) bird. Smullyan remarks that superficially, the Identity bird appears to have no intelligence at all, and has been referred to as the Idiot bird. However the real reason for its apparently unimaginative behaviour is that it has a big heart and is *fond* of every bird. So when you call x to the Identity bird, the reason it responds by calling back x is not that it can't think of anything else; it's just that it wants you to know that it is fond of x .

It turns out to be an important fact that every combinatory bird is fond of at least one bird (speaking technically, we say every bird has a *fixed point*). That is, for every bird there is some song which you can call to it and receive the same song in response.

Figure 1 shows the song map of the Identity bird. We won't be as kind as Smullyan. We will refer to it as the Idiot bird from now on since that's more fun and easier to say.

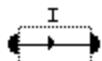


Figure 1. Idiot Bird

We make the convention that the filled half-circle on the left of the box represents the bird's ear (combinatory birds only have one ear) and the one on the right represents its throat. The dotted box is to enclose the plumbing which connects the two. We can also label the box. The label is not a necessary part of the graphical notation but merely to remind us of what we call this bird in English; "I" for "Identity" or "Idiot". In this case the plumbing is

the simplest possible, a single pipe represented by the solid line. I have put an arrowhead on the line to indicate the direction of flow of information from ear to throat, but in future since flow will nearly always be from left to right we will not clutter our diagrams with arrow heads except to indicate reverse flow.

Another simple bird is the Mockingbird. It is called a Mockingbird because its response to any bird is the same as that bird's response to itself. This means that if you call out x to a Mockingbird you will get the same response as if you had called out x to x .

Figure 2 shows the song map of the Mockingbird. We will see later the reason for using a lower-case omega (ω) as its abbreviation. For now you can think of it as an upside-down 'm'.



Figure 2. Mockingbird

I have introduced a new symbol, the filled circle, which you might think of as the combinatory birds' brain cell. I call it an *applicator*. We can see that an applicator receives information from above and from its left and responds to its right. It is called an applicator because of what it does with the information it receives; it *applies* the bird whose description it receives from above, to the song which it receives from its left, in order to determine its response. The song arriving at the top is called the *operator* and the one from the left, the *operand*. You might think of the circle as representing an ear and throat with nothing between them, but waiting to have a song map introduced between them from above.

You might also think of the applicator as a sort of universal bird since all it requires is the description of a bird in order to *become* that bird.

It may seem a little inside-out, or like pulling yourself up by your own bootstraps, to define ordinary combinators as arrangements of universal ones, but such are the foundations of mathematics.

The strips of movie film in figures 3 and 4 show the sequence of what happens when a Mockingbird hears the Idiot song and when it hears the Mockingbird song (i.e. the song describing itself).

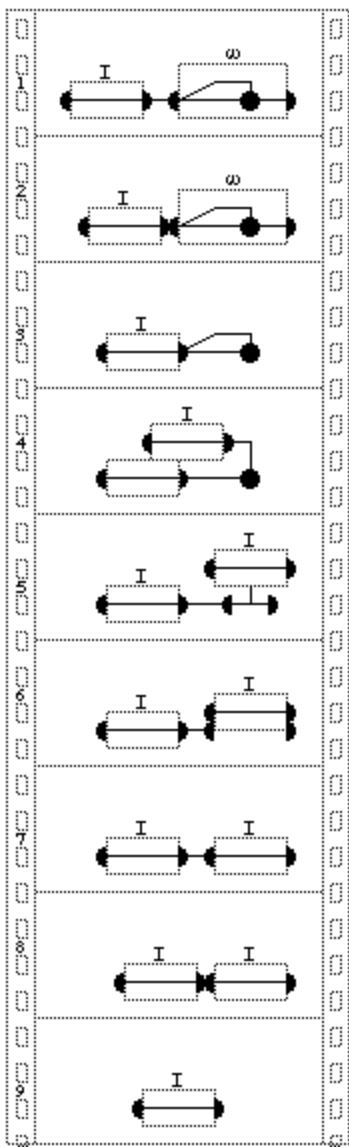


Figure 3. A Mockingbird hears the Idiot song

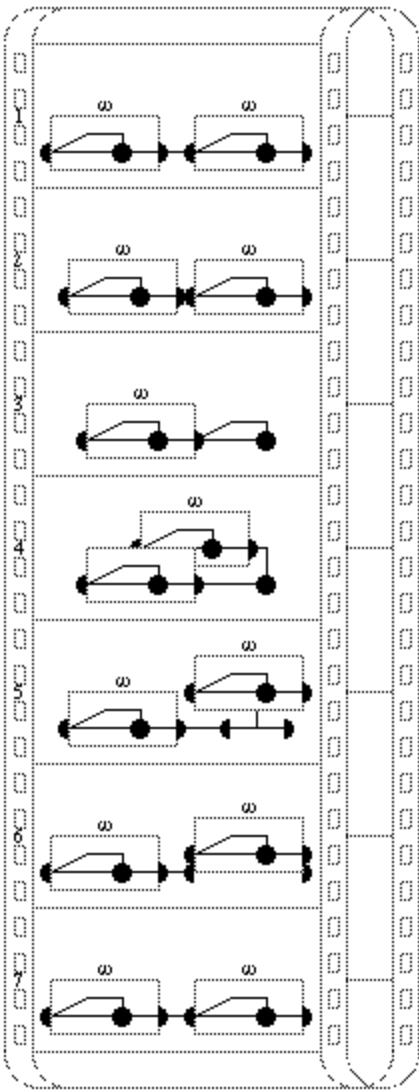


Figure 4. A Mockingbird hears the Mockingbird song

Ideally these movies would be shown in real time on a computer screen with many more in-between frames to give the appearance of smooth motion. You may be able to get some sense of motion if you can cause your sight to snap suddenly from each frame to the next. Alternatively you could copy the movie, paste it onto light card, cut out the individual frames and make a 'flick-picture'. The later movies in this document leave out more and more in-between frames to save space, so the 'flick-picture' approach will not work and we will be forced to *imagine* the motion based on the more detailed movies we will have already seen.

In both movies we can see the song on the left approaching the Mockingbird on the right. As the song passes through the Mockingbird's ear, the box around the Mockingbird disappears. This annihilation as song meets ear might have been called box reduction but for historical reasons it is called *beta (β) reduction*. The song is then *replicated* to follow two pipes which branch out from what was the inside of the ear. Both pipes lead to the same applicator. The high pipe causes the song (or the bird it describes) to be *substituted* for the applicator which has been acting as a stand-in or variable bird. The lower pipe then leads the other copy of the same song to be heard by this newly installed bird.

At this stage (in the seventh frame of each movie) we see that the definition of a Mockingbird has indeed been satisfied because the result corresponds in both cases to a song being heard by that bird which it represents.

In the case of the Idiot song being heard by an Idiot bird we can perform another beta reduction and obtain a result which is just the Idiot bird/song. This cannot be reduced any further. When no further beta-reductions can

be performed we say that a bird (or song) is in *normal form*.

Notice that the Mockingbird has disappeared in the process of producing its response, or rather it has been transformed into its response. This is another aspect of combinatory birds which is very different from ordinary birds.

In the case of the Mockingbird song being heard by a Mockingbird, the seventh frame is just what we started with. No matter how many times the same steps are repeated the song will never be complete. For this reason it is generally considered rather cruel to go around the forest calling out the Mockingbird song to Mockingbirds. If you call out any other song they know just what to do but when they hear their own song their poor little brains get lost in an endless loop. A Mockingbird in this magical or paradoxical state (i.e. responding to the Mockingbird song) is called an Omega (Ω) bird, and this is why the Mockingbird alone is often called the little omega (ω) bird (Omega is the last letter of the Greek alphabet and is often associated with infinity). Since the Omega bird does not stop reducing, we say it has no *normal form*, and so we are free to represent it by any frame of its endless movie. Figure 5 shows the most popular representation.



Figure 5. Omega bird

Note that the outer box in this song map is not a proper box since it has no ear. It is merely to allow a name to be given to the arrangement inside it and doesn't alter the meaning. Not only doesn't the Omega bird have a normal form, it doesn't even have a *head normal form* (we also say it is *unsolvable*). Having a head normal form requires at least having a single proper outer box (i.e. with an ear), as well as some other requirements which are beyond the scope of this paper. The Omega bird can never respond to anything it hears. The Omega bird is usually identified with *all* unsolvable birds and is interpreted as *undefined* when the lambda calculus is interpreted for logic and arithmetic.

The direction of flow in song maps will usually be understood to be from left to right, but if we allow flow in the reverse direction and indicate it with an arrow we can give a simpler *cyclic* song map for the Omega bird as shown in figure 6.

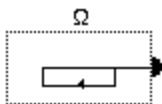


Figure 6. Omega bird (cyclic form)

It is unfortunate that the primary form of reduction was named after the *second* letter of the Greek alphabet, beta (β). The form of conversion which was given the first letter, alpha (α), turns out to be merely an artifact of the particular method of representing these songs as one-dimensional strings of symbols in the textual lambda calculus. In the textual lambda calculus it is not possible for *replication* or *substitution* to occur separately from *beta reduction* and so they are not recognised as separate steps.

Figure 7 describes the allowable transformations by using a *meta*-notation. The cloud with an italic character stands for any expression (arrangement). The fanned-out lines with the ellipsis '...' between them stands for any number of pipes originating from the same point. The box with the italic character stands for a box containing any expression.

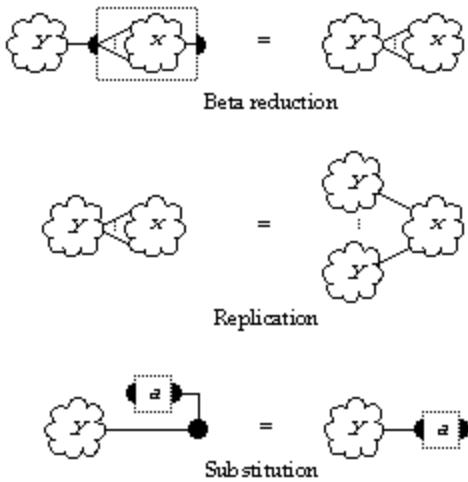


Figure 7. The transformation rules expressed in a meta-notation

Two layer birds

It would be a very boring forest if the only birds were the Idiot bird and Mockingbird and the birds derivable from them. I now introduce the simplest of the *two layer birds*, the Kestrel and the Kite. Like the Idiot bird, these birds contain no applicator and yet they turn out to be amazingly useful. See figure 8.

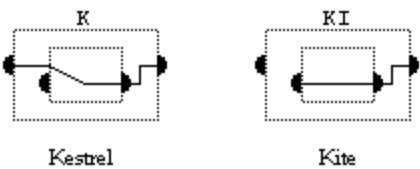


Figure 8. The two layer birds having no applicator

The Kestrel's response to hearing a song x is (a song which describes) a bird which responds with x no matter what it hears. We say the Kestrel's response to x is the constant- x or Kx bird (In German, the word for "constant" starts with a 'k'). The Kite ignores what it hears and always responds with the Idiot song, so it is the KI bird.

Note that these birds have an inner box with an inner ear and throat. Pipes may pass freely into inner boxes from outer ears without having to pass through an ear. Combinatory birds can have ears which aren't connected, so long as something connects to every throat.

Figures 9 and 10 show movies of the Kestrel hearing the Idiot bird and the Kite hearing the Mockingbird.

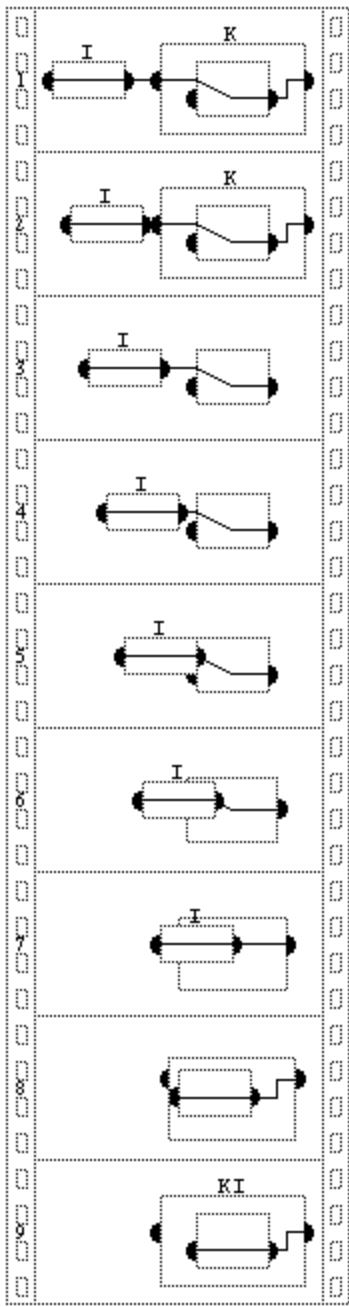


Figure 9. A Kestrel hears the Idiot bird

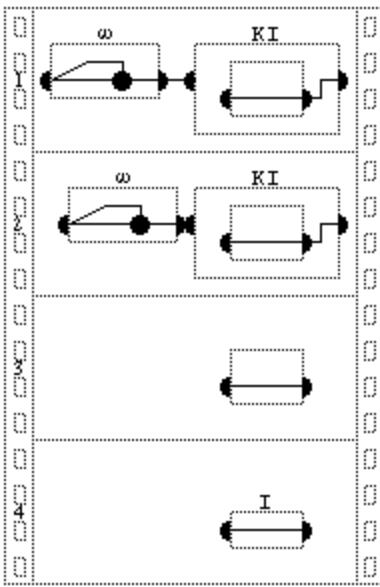


Figure 10. A Kite hears the Mockingbird (or any bird)

So the Kestrel's response to the Idiot song is in fact the Kite. In the Kite movie, notice that it wouldn't have mattered what song the Kite heard, it would still have responded with the Idiot song. The Kestrel and Kite are sometimes called the *true* and *false* birds because they are commonly interpreted as such when combinatory birds are made to do logic and arithmetic. See Appendix A.

Note that between frames 3 and 9 of the Kestrel movie we perform a simple relocation which must not be treated as a beta reduction. No box is annihilated because no song passes through an ear. Figure 11 describes the general situation in the meta-notation.

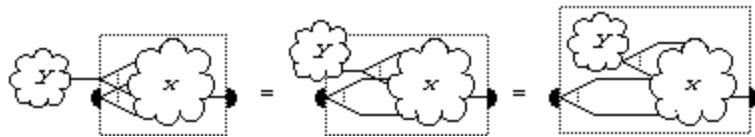


Figure 11. Simple relocation (no beta reduction is involved)

Next we introduce the two-layer birds having one applicator. There are four of these as shown in figure 12, the Idiot bird once removed, the Thrush, the Konstant Mocker and the Crossed Konstant Mocker. Notice that we've included the equivalent textual lambda notation, " $\lambda ab.ab$ " etc., underneath each song map.

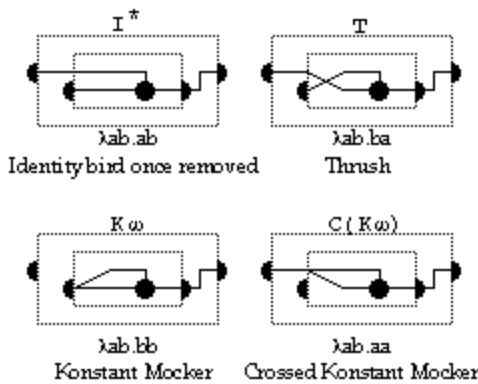


Figure 12. The two layer birds having one applicator

The meaning of the term "once removed" will become clearer when we see some more once removed relatives. You should see what is meant by "crossed" if I tell you that the Thrush could be called "the Crossed Idiot-bird-once-removed" (CI*). We can think of the 'T' in 'Thrush' as also standing for 'transposed'. We see that the Konstant Mocker will respond with the Mockingbird no matter what it hears. The Crossed Konstant Mocker is of no particular interest but is shown to complete the family.

Note that the crossed pipes in the Thrush are not to be considered as connecting together but merely passing each other. This is one of the limitations of only having two-dimensions in which to draw our song maps. However, we can always tell the difference between a joint and a crossover because joints can not occur as fan-ins $\Rightarrow \times$,

only fan-outs $\leftarrow \checkmark$. For a crossing to be interpreted as a joint, it would have to involve an illegal fan-in followed by a fan-out.

There are sixteen two-layer birds that have two applicators and not all of them have English names. I show a few of the most popular in figure 13, the Lark, Owl, Warbler and Crossed Warbler.

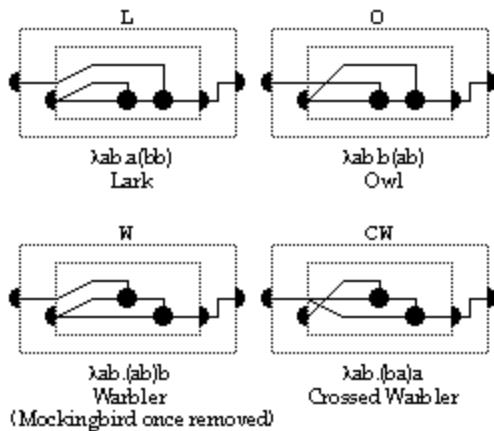


Figure 13. Some common two layer birds having two applicators

Although the Warbler could have been abbreviated as ω^* since it is the Mockingbird-once-removed, historically the use of W goes back further than that of ω and in fact the Mockingbird was referred to only as WI for a long time. This is another reason for the choice of ω rather than M for the mockingbird. Note that LI = OI = WI = ω .

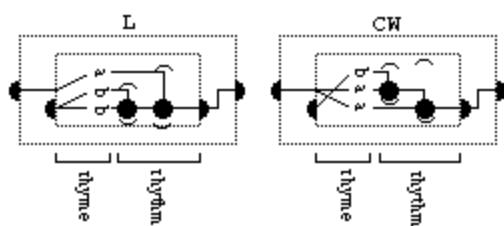


Figure 14. Lark and Crossed Warbler annotated to show rhyme and rhythm

Now that we have birds with more than one layer and more than one applicator we can distinguish two major sections of any song map, the rhyme and the rhythm. The rhyme is the section at the leftmost end of a box, where all the fan-outs and crossings occur. For example the Lark and the Warbler have the same rhyme. If we assign the letter 'a' to the outermost ear and 'b' to the inner one and follow the pipes to the right past the sloping sections until they run parallel again, we can write the Lark and Warbler's rhyme as "abb" corresponding to the

downward direction on the song map. The Crossed Warbler then has "baa" and the Owl has "bab". For birds with two layers and two applicators there are eight possible rhyme schemes. The general rule is for birds with l layers and a applicators there are l^{a+1} rhyme schemes.

The rhythm section is the section immediately to the right of the rhyme, containing the applicators. Notice that there are no fan-outs or crossings in the rhythm section, only pipes being reduced in pairs, via applicators, until only one pipe remains. This pipe then connects to the throat. You can see that the Lark and the Owl have the same rhythm but this is different from the rhythm shared by the Warbler and the Crossed Warbler. The first we write as "(- - -)" and the second "((- -) -)", although in future we can omit the outer brackets without ambiguity. These are the only possible rhythms for a bird with two applicators. It could be that the duration of the notes is halved each time we enter parentheses and doubled again when we leave. Figure 14 shows a Lark and a Crossed Warbler annotated to show the correspondence between textual and graphical notations.

You may have noticed that a rhythm section corresponds to a binary tree. The number of rhythms using a applicators is the a -th Catalan number which can be found by taking the number of rhythms using $a-1$ applicators and multiplying by $(4a-2)/(a+1)$. Of course there is only one rhythm with zero applicators. The series starts, 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786,

So we see that the ultimate question to the answer to life the universe and everything, from Douglas Adams' *The hitchhikers guide to the galaxy*, could in fact have been "How many combinatory rhythms can you make with five applicators?".

[My thanks to Walt "BMeph" Rorie-Baety for encouraging a long overdue mention of Catalan numbers. 27-Sep-2009]

We can determine the number of possible birds with a given number of layers and applicators by multiplying these two numbers together (rhymes x rhythms). We can combine the textual notations used above for rhyme and rhythm to give song schemes like "(ab)b" for the Warbler and "b(ab)" for the owl. However, this is not enough to uniquely describe a bird textually since for example the Idiot bird and the Kestrel would both be "a". We must also indicate how many layers the bird has. We do this by prefixing the rhyme and rhythm with a complete list of the layers involved from outside to inside, preceded traditionally by a lambda (λ) and followed by a dot. So the Idiot bird is " $\lambda a.a$ ", the Mockingbird is " $\lambda a.aa$ ", the Kestrel is " $\lambda ab.a$ " and the Kite is " $\lambda ab.b$ ".

There is no limit to the number of applicators a bird can have. It does not depend on the number of layers. For example, the multiple-mockingbird family consists of all birds with only one layer. There are two different double-mockingbirds (having two applicators), five triple-mockingbirds (three applicators), fourteen quadruple-mockingbirds and so on. Under this scheme, the Idiot bird may be considered as the zeruple-mockingbird. [Note: This is different to the sense in which Smullyan uses "double mockingbird"]

Of the 80 two-layer birds which have three applicators, I show only the Turing bird, named after its discoverer the logician and computer scientist Alan Turing. See figure 15. I will tell you more later about the importance of the Turing bird in connection with *fixed-point* birds.

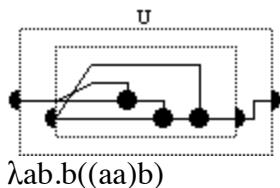


Figure 15. Turing bird

Three layer birds

We now move on to three-layer birds. We start with the most popular three-layer two-applicator birds, the Cardinal and the Bluebird in figure 16. These birds, along with the Idiot bird and the Kestrel, were among the first five combinatorists described in 1920 ([published in 1924](#)) by [Moses Schönfinkel](#). Incidentally "schöne Finken" is German for "beautiful finches".

[My thanks to Henning Kopp for correcting my earlier poor translation. 30-Aug-2009]

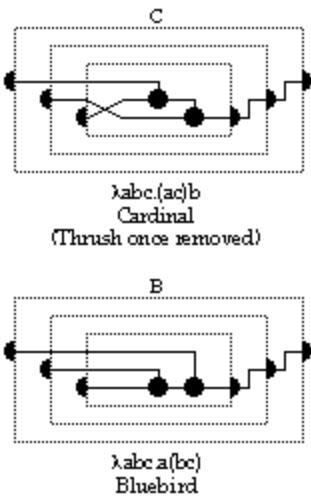


Figure 16. The best known three layer two applicator birds

The Cardinal might have been called the crossing bird because it always responds with the crossed cousin of whatever bird it hears. For example if it hears the Warbler song it will respond with the Crossed Warbler and vice versa. Its response to the Thrush song is the Idiot bird once removed, and vice versa. The word "converse" is often used instead of "crossed" in this context. If it hears a song which has more than two layers the Cardinal will cross the connections from the two outermost layers only. The Cardinal has the effect of altering the rhyme of whatever bird it is applied to, but it will never alter the rhythm.

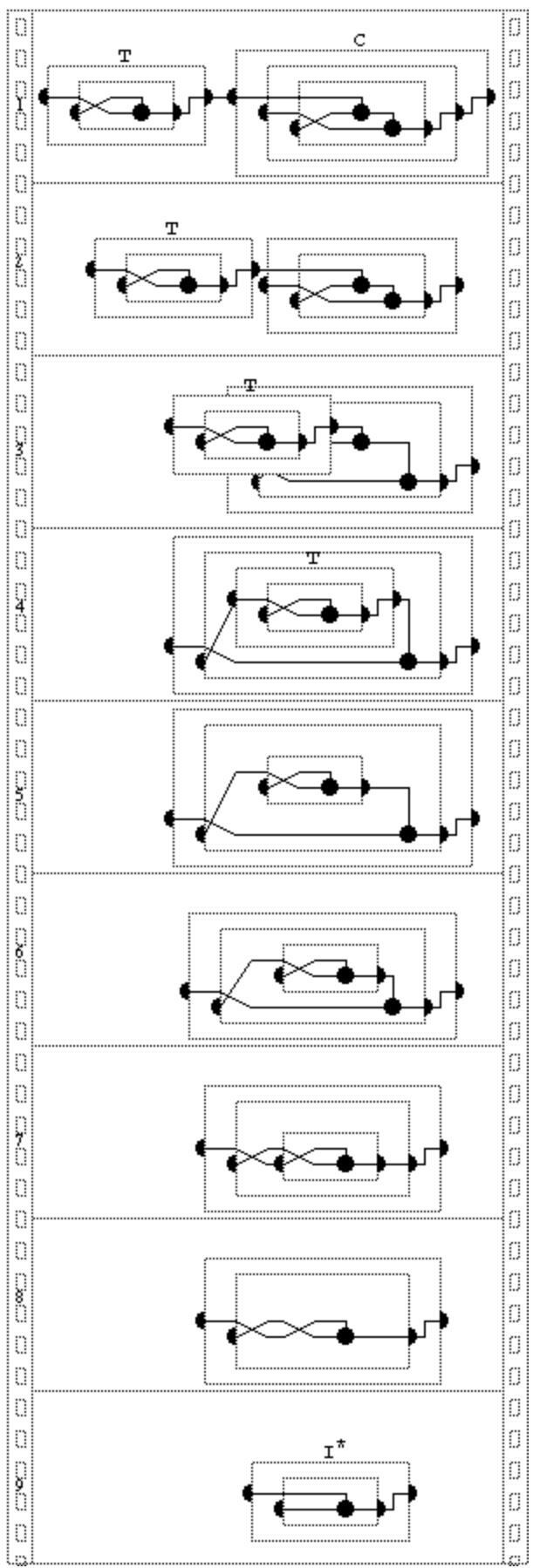


Figure 17. A Cardinal hears the Thrush song

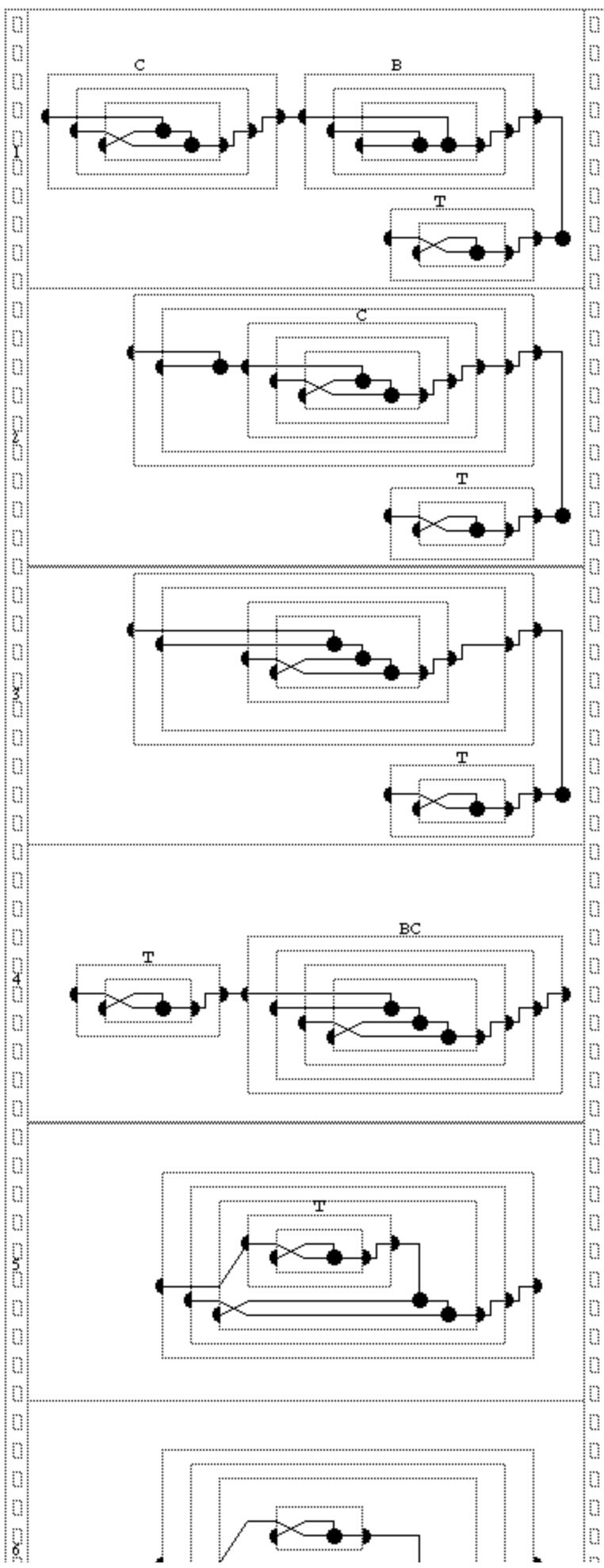
Figure 17 shows a movie of a Cardinal doing its thing with a Thrush. It shows between successive frames, (1) a beta reduction, (2) a simple relocation, (3) a substitution, (4) a second beta reduction, (5) relocation, (6) substitution, (7) a third beta reduction and (8) relocation (9).

Notice that the second and third beta reductions are slightly different to those we have seen before. The connection which allows the box to be removed comes not from the throat of another bird but from *the inside of the ear* of another bird. Our meta-notation (cloud picture) for beta reduction should be understood to allow for either.

While the Cardinal is a rhyme altering bird which will never alter the rhythm, the Bluebird is a rhythm altering bird which will never alter the rhyme. The Bluebird is often called the composition bird (another musical reference?) because when applied to two birds in succession it produces a single bird which is the *composition* of those two birds. By "applied to two birds in succession" we mean that we apply B to the first bird *a* then we apply the resulting bird *Ba* to the second bird *b* to obtain the result *(Ba)b*. By the *composition* of two birds *a* and *b* we mean a single bird which when applied to some bird *c* has the same effect as first applying *b* and then applying *a* to the result, i.e. *a(bc)*. Unfortunately "composition" doesn't start with a 'B' but we can think of the 'B' as standing for bracketing if we think in terms of the textual rhythm notation "- (- -)" we saw earlier.

The Bluebird also functions as a *once-removal* bird for three layer birds only. You should have guessed by now that once-removal involves inserting a new applicator into the topmost pipe and supplying its operator from a new outer layer. Calling out the Idiot song to any once-removed bird will have the effect of undoing the once-removal.

Figure 18 shows how a Bluebird composes a Cardinal and a Thrush. I have substantially reduced the amount of in-between detail to the point where I no longer show relocations, replications or substitutions separately where they immediately follow a beta reduction. This coarse level of detail is the *best* one can do in the textual lambda calculus!



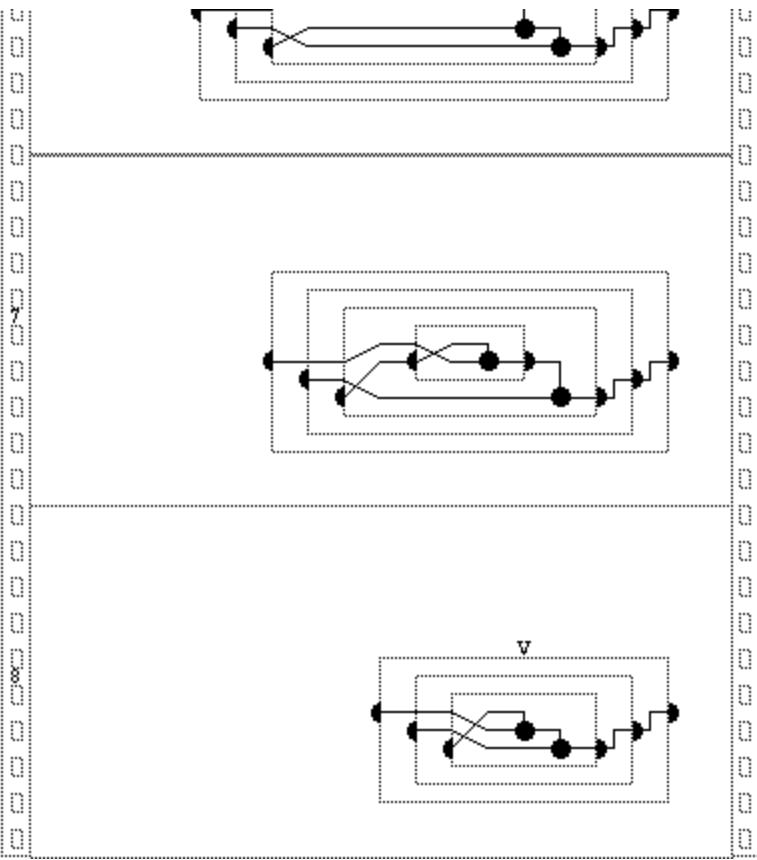


Figure 18. A Bluebird composes a Cardinal and a Thrush

The resulting bird is called the Vireo, also known as the pairing bird because of its ability to take two birds and form a single bird from which either part may be recovered by applying it to either the Kestrel (*true*) or the Kite (*false*) depending on whether the first or second part is required. This pairing property can be used repeatedly to make lists or trees. This property also comes in handy for doing logic and arithmetic as shown in appendix A.

Note that the first frame of the movie in figure 18 has the form

$$\langle \langle C \rangle \rangle \rightarrow \langle \langle B \rangle \rangle = \langle \langle T \rangle \rangle \rightarrow \bullet = \langle \langle B \rangle \rangle \rightarrow \langle \langle C \rangle \rangle = (B \circ T).$$

This is different from

$$\langle \langle T \rangle \rangle \rightarrow \langle \langle C \rangle \rangle \rightarrow \langle \langle B \rangle \rangle = \langle \langle T \rangle \rangle \rightarrow \bullet = B(C \circ T).$$

The above equivalences follow from the substitution rule given in figure 7. Figure 19 expresses this non-equivalence as a general non-rule in the meta-notation.

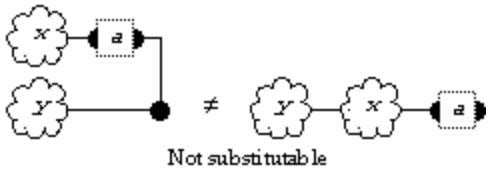


Figure 19. Only an abstraction (a box with a free ear) can be substituted for an applicator

It is traditional in the textual form that operator and operand are simply placed side by side with the operator on the left and the operand on the right with parentheses to remove any ambiguity. This is opposite to the order for direct application in the graphical form. It is also traditional in the textual lambda calculus that we avoid having to write so many brackets by using a *left*-associative convention. That is, BCT is the same as (BC)T. This proves to be rather unfortunate when compared with the graphical form. The graphical expression which looks most like "BCT" (apart from the left-right reversal) corresponds in fact to B(CT). If we were to make a *right*-associative convention for the textual form, the correspondence with the graphical form would be more obvious. [György Révész \(1988\)](#) also remarks on how unfortunate is the traditional choice and, in fact, chooses to go against tradition in his book (p16-17). We will stay with tradition in the remainder of this paper.

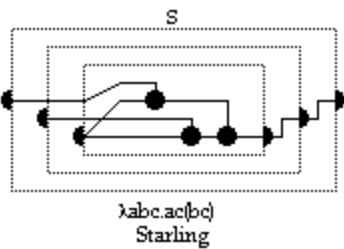


Figure 20. (Owl once removed)

The Starling, shown in figure 20, is the last of Schönfinkel's original five combinators to be introduced. These are the Idiot bird, Kestrel, Cardinal, Bluebird and Starling, I, K, C, B, S (although Schönfinkel called some of them by different letters at the time). These five birds alone can easily generate all other birds. The Starling is the only one of these birds to perform replication. Note that SII = ω.

Primitive birds

While it is remarkable enough that the five birds I, K, C, B, S can be used to derive all others, it is an even more remarkable fact, shown by Schönfinkel, that all other birds can be derived from the Starling and Kestrel alone, although the expressions involved can be enormous. For example:

$$\begin{aligned} I &= SKK \\ B &= S(KS)K \\ C &= S(BBS)(KK) = S(S(KS)K(S(KS)K)S)(KK) \end{aligned}$$

K and S have been likened to Adam and Eve or Yin and Yang. Note that crudely speaking, K destroys (eliminates) and S creates and changes (replicates and changes both rhyme and rhythm).

There are deterministic rules which allow us to take any expression and work backwards to obtain an equivalent in terms of K and S alone (see [Smullyan \(1985\)](#); [Peyton Jones, 1987](#)). This fact has been used to build computers called combinator reduction machines, whose basic instructions correspond to K and S, although for efficiency reasons I, C, B and Y (which we will meet later) are usually included as well. With regard to the number 42 and

The hitchhikers guide to the galaxy; could it be that Deep Thought was a combinator reduction machine?

Schönfinkel dreamed of another bird he called the Jay, from which the Starling and Kestrel (and so all birds) could be derived. The quite complicated bird shown in figure 21 is not quite what Schönfinkel had in mind, but it has been given the name Jay.

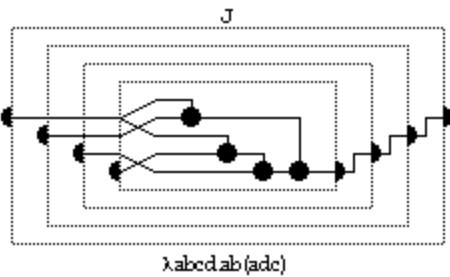


Figure 21. Jay

The Jay was discovered in 1935 by J. Barkley Rosser and has the property that it can be used in association with the Idiot bird to derive all birds except those which ignore or eliminate one or more of their inputs (such as the Kestrel and Kite). I will show how we obtain C, B and S. To do this it is convenient to show certain other birds along the way. We have met the Thrush, Mockingbird and Warbler before, but the Robin (R) is new.

$$T = JII$$

$$R = JT = J(JII)$$

$$C = RRR = J(JII)(J(JII))(J(JII))$$

Here the expressions become too big to continue expanding fully.

$$B = C(JIC)(JI)$$

$$\omega = C(C(C(BJT)T)T)T$$

$$W = C(B\omega R)$$

$$S = B(BW)(BBC)$$

[My thanks to Dr. Johannes Waldmann for correcting an error in an earlier version of the expression for the Bluebird.]

To test your understanding of reduction you might like to work out the normal form of the Robin now by reducing the graphical version of the expression JT. The Robin is shown in Appendix A.

Since Rosser's Jay, Schönfinkel's dream has been realised by the discovery of several bird species of the genus Xenops (strange operators?), each of which alone is a basis for the entire combinatory forest. One such was even discovered by Rosser. A typical Xenops will generate a Kestrel when applied to itself (possibly multiple times) and will generate a Starling when applied to the resulting Kestrel. See [Fokker \(1992\)](#). The simplest known Xenops (*Xenops minutus*), also known as the iota (ι) bird, was discovered by [Chris Barker](#) around [2001](#). It can be written as $\iota = \lambda a.aSK$. In [2012](#), [Jeff Tamer](#) was so impressed by single-point bases for computation, and this graphical notation, that he had the song map for the equally simple $X = \lambda a.a(CS)K$, plus a few embellishments, permanently tattooed on his back! Tamer considers the Crossed Starling a more natural choice than the Starling given the usual arrangement of operator and operand in song maps.

Many other sets of primitive combinators are possible and we could argue forever over which is more fundamental. In fact there is a direct relationship between bases for the theory of combinators and axiom schemes for implication logic, via a mapping called "Formulae as Types". See [Hindley & Seldin \(1986\)](#).

One basis that proves particularly convenient for this graphical notation is K, ω , T, B, in which elimination, replication, permuting (rhyme-changing) and regrouping (rhythm-changing) are represented in their simplest

form, each by a separate combinator. Given that $R = BBT$ we can use the derivations above to show that S is derivable from ω , T , and B .

Fixed-point birds

I remind you that the bird/song analogy is only an analogy. There is really no distinction between birds and songs. This corresponds merely to the operator/operand distinction which is relative to a particular applicator.

If we cause a Turing bird (U) to hear its own song we end up with a bird called the Theta (Θ) bird which is a fixed-point finding bird or simply a *fixed-point* bird. The Turing and Theta birds were discovered by Alan Turing in 1937. The Theta bird does not have a normal form. Figure 22 shows one possible representation obtained by performing a single beta reduction on UU and relocating the first U inside the remains of the second. This version of the Theta bird is at least in *weak head normal form* which means only that it has an outer box with an ear.

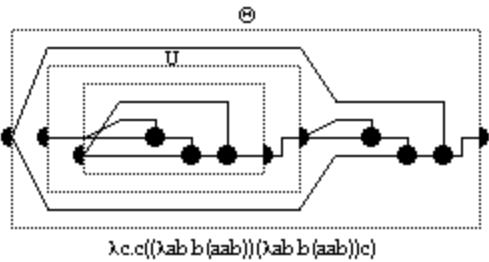


Figure 22. Theta bird

A fixed-point bird has the amazing property that on hearing any bird b , it responds with a bird f of which b is fond. That is, it responds with a fixed-point of b . So if we want a solution for f in the recursive equation $f = bf$ we need only write Θb (or Yb as we shall see later). The proof that fixed-point birds exist, is a very powerful result since it says that it is meaningful to *define* functions by means of recursive equations such as that for f above. While a combinatory bird may have more than one fixed point it turns out that the fixed points can always be ordered according to their *definedness*. It is beyond the scope of this paper to describe the precise meaning of definedness in this context, but recall that the Omega bird is considered to be completely undefined. The thing defined by a recursive equation is considered to be that fixed point which is least defined, called simply "the least fixed point". See any of the introductory texts mentioned in the references at the end of this paper.

The simplest fixed-point bird, the Why bird, is generally attributed to Haskell Curry around 1942 (see the notes on p185 of [Curry \(1958\)](#)). Perhaps the name refers to the incredulity of its discoverer, "Why does it exist?" or "Why does it work the way it does?". See figure 23.

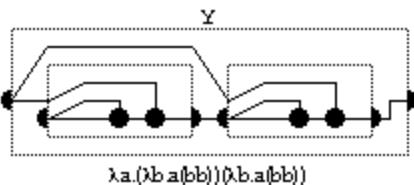


Figure 23. Why bird

As with the Omega bird, the Why bird turns out to have a simpler cyclic description ([Simon Peyton Jones \(1987\)](#)) which better supports our intuition about how it works. See figure 24.

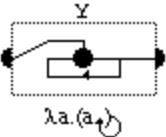


Figure 24. Why bird (cyclic description)

Note for example that $YI = \Omega$ for both cyclic and acyclic forms of Y and Ω . Every combinator is a fixed point of I but Ω is the *least* fixed-point of I although the meaning of "fixed" must be stretched somewhat to accommodate this. Some other interesting derivations involving Y are $Y = SLL$, (where $L = CB\omega$, the Lark) and $\Theta = YO$, (where $O = SI$, the Owl).

The cyclic *textual* form shown for the Why bird in figure 24 can similarly be used to describe the Omega bird as " (\rightarrow) ".

In such a fundamental discipline as this, we should not be surprised to find an object which is apparently disappearing up its own fundament.

The surprising thing is that Y , Θ and Ω can be written *non-cyclically*.

It is a fact of mathematical life (as shown by Kurt Gödel) that if we are to have something as powerful as a Why bird (a fixed-point bird) we must accept the risk of producing an Omega bird (a non-terminating bird). Our old friend the Mockingbird, with its ability to apply a bird to itself, is implicated in both.

Conclusion

I hope that this paper has shown the potential of a carefully designed animated graphical notation to bring human spatial and temporal intuitions to bear on the study of such an abstract discipline as the lambda calculus. I hope it will be as useful as an intuition amplifier for the lambda calculus and combinators as are [Warren Robinett's \(1982\)](#) game *Rocky's Boots* (and its successor [Robot Odyssey](#)) for Boolean algebra and John Conway's game of *Life* for cellular automata (see [Poundstone, 1985](#)).

Some important aspects of the notation are:

- its potential to be automatically generated and manipulated,
- its potential for the smooth animation of reduction steps for teaching purposes,
- its complete independence of text, and hence elimination of alpha conversion (although textual cues may still be used, such as the naming of abstractions),
- its use of *containment* as well as directed-connectivity as a visual cue,
- its separation of rearrangement/replication of variables from their order of application (rhyme vs. rhythm),
- its aesthetic layout rules (as yet unstated) which should determine a canonical graphical form for every normal form expression, for ease of recognition.

Future directions

While this paper has been an attempt at an *informal* introduction to lambda calculus, the graphical notation should be formally described and the correspondences between it and the well-understood textual lambda calculus should be elucidated if it is to be widely used. If the drawings are to be made by machine or with machine assistance, the aesthetic layout rules should also be made explicit.

I might be convinced that the diagrams should be flipped horizontally so that the order of *applications* maps

more directly onto the textual form, however the order of *abstractions* would no longer map directly, unless (for example) $\lambda abc.\text{def}$ were to be rewritten as $\text{def}.cba\lambda$.

If this exposition were to be extended, some explanation should be given of *normal order* reduction versus unsafe reduction orders, and *head normal form* and *weak head normal form* should be explained fully. The graphical version of *eta* (η) reduction should be introduced to allow for the extensional equivalence of combinators such as I and I^* . Note that these are the only extensionally equivalent pair of combinators in this paper.

I hope that someone, with more resources than I, will implement it as a computer 'game'. The user would set up an initial expression and watch as it evolved according to the combinatory 'laws of physics' (normal order reduction). It might be more interesting if the metaphor was changed to one of mythical beasts devouring one another (could give a whole new meaning to "eta" reduction). [I thank Chris Wong for launching such a project in 2012 with his [lambda calculus visualiser, Sylvia \("of the forest"\)](#)]. It should also be possible to find a mapping which will give a unique musical tune for most combinators. The tune might correspond to the *type* of the combinator when it has one. See [Hindley & Seldin \(1986\)](#).

The notation might be extended to become a full programming language, or at least a program *reading* language. For this purpose I would allow the internal details of lower-level named abstractions to be suppressed while providing pan, zoom-in and zoom-out facilities for program browsing and editing. A translator from an existing textual functional language would be a worthwhile project, although matching some of their more recent syntactic conveniences could be difficult.

Appendix A. Logic and Arithmetic

There are many ways of *interpreting* lambda calculus for logic and arithmetic. By far the most popular choices for *true* and *false* are K and KI respectively, although I and KI have some advantages, and of course any such pair could also be swapped. The former is a particularly useful choice because the '*if then else*' function is then simply the Identity combinator (or no combinator at all). Remember that K (*true*) returns the first of two arguments and KI (*false*) returns the second. Figure 25 is to remind you what these birds look like, along with the Thrush and two of its descendants, the Vireo and the Robin.

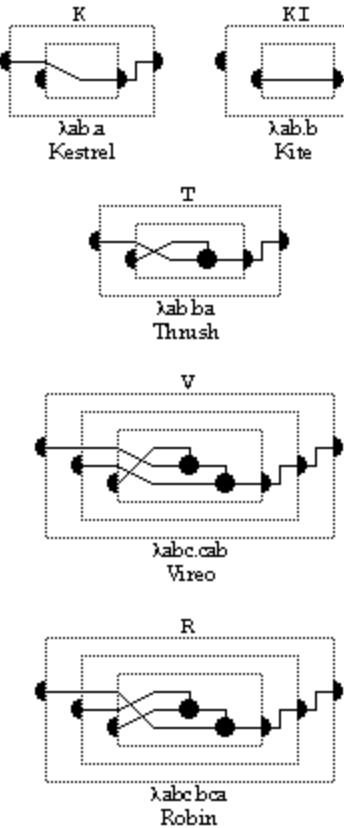


Figure 25. Some birds which can be used to perform logic and arithmetic

We have seen the Vireo before as the result of BCT. The Robin may also be derived from the Thrush as BBT. I now give the usual interpretations of the logical constants and connectives from [Henk Barendregt \(1984\)](#), which I leave for you to verify for yourself. Note that all functions are prefix not infix, that is, we must write "*implies a b*" rather than "*a implies b*". You may like to consider the expression "*Y not*" (the least fixed point of the logical negation function) to determine why Curry referred to the Y combinator as the paradoxical combinator.

```

true = K
false = KI
not = V false true = V(KI)K
implies = R true = RK
and = R false = R(KI)
or = T true = TK
equiv = CS not = CS(V(KI)K)

```

Here are Barendregt's interpretations for Peano arithmetic (many others are possible).

```

zero = I
succ = V false = V(KI)
pred = T false = T(KI)
isZero = T true = TK

```

succ is the successor function such that $succ\ n = n+1$, *pred* is the predecessor function such that $pred\ n+1 = n$, and *isZero* returns *true* if its argument is *zero* and *false* if it is positive. We don't care what *pred zero* is and we don't care what happens when any of the functions are applied to expressions which do not represent numbers. Expressions which represent numbers are called numerals. Here are some example numerals. You should test the operation of *pred* and *isZero* on them.

one = *succ zero* = $V(KI)I$
two = *succ one* = $V(KI)(V(KI)I)$

Note that these numerals work by tallying (sometimes called unary although it is not base one). The numeral n consists of a list of n *falses* terminated by an I (as the empty list). It is possible to produce more efficient numeral systems, such as binary, in the lambda calculus but the definitions of their *succ*, *pred* and *isZero* functions are more complicated. For example den Hoed's binary numerals are $\lambda abc.a$, $\lambda abc.ac$, $\lambda abc.abc$, $\lambda abc.acc$, $\lambda abc.abbc$, $\lambda abc.acbc$, , where b and c act as 0 and 1 and the most significant bit is the rightmost ([van der Poel et al., 1980](#)).

Given any numeral system with *succ*, *pred* and *isZero*, we can define addition recursively as follows.

add b c
 $= if(c = 0) then b else ((add b c-1)+1)$
 $= if(isZero c) then b$
 $else (succ (add b (pred c)))$
 $= ifThenElse (isZero c) b$
 $(succ (add b (pred c)))$
 $= (isZero c) b (succ (add b (pred c)))$
 $= isZero c b (succ (add b (pred c)))$

Then by abstracting b and c (abstraction is the inverse of beta-reduction) we can write

add b c
 $= (\lambda bc.isZero c b$
 $(succ (add b (pred c)))) b c$

so

add
 $= \lambda bc.isZero c b (succ (add b (pred c)))$

Now by abstracting *add* itself we can write

add
 $= (\lambda abc.isZero c b$
 $(succ (a b (pred c)))) add$

This is now in the form of $f = bf$ so we can use a fixed-point combinator to solve it, and we write

add
 $= Y (\lambda abc.isZero c b$
 $(succ (a b (pred c))))$

This procedure can be continued to define multiplication and exponentiation and indeed any computable function on the integers, as proved by Stephen Kleene.

Here's another way of interpreting the pure lambda calculus for arithmetic, called Church numerals. Church numerals have a particularly simple form when expressed in the textual notation.

zero = $\lambda fx.x$ = KI
succ = $\lambda afx.f(afx)$ = SB
So
one = $\lambda fx.fx$
two = $\lambda fx.f(fx)$
three = $\lambda fx.f(f(fx))$

etc.

$\text{add} = \lambda abfx.\text{af}(bfx)$

$\text{multiply} = \lambda abf.a(bf)$

$\text{power} = \lambda ab.ab$

[Jeff James \(1993\)](#) has developed a system, inspired by Spencer-Brown's work, which does arithmetic using a topological notation with only two kinds of boundary. These may be represented textually as brackets () and [], with the interpretation of the empty expression as zero, () as one, ()() as two etc., addition is simple juxtaposition ab and multiplication is $([a][b])$ and so on. If () is understood as an exponential function and [] as a logarithmic function, it all suddenly makes sense, however the arithmetic is actually performed using only four simple axioms: $([a]) = a$, $([a]) = a$, $[[]a = []]$, $([a][b])x = ([ax][bx])$. I mention this because I can't help feeling that it is somehow related to the lambda calculus and combinators but I don't see how. Maybe you do.

Acknowledgement

This work was carried out while the author was supported by an Australian Postgraduate Coursework Award. Thanks go to Lou Kauffman and Dick Shoup for their discussions regarding earlier drafts of this paper.

References

- Barendregt, H.P. <http://www.cs.ru.nl/~henk/> (1984) *The lambda calculus Its syntax and semantics*. North-Holland Publishing Company, Amsterdam.
- Barker, Chris (2001) *Iota and Jot: The simplest languages?*. <http://www.nyu.edu/projects/barker/Iota/>
- Church, A. (1941) *The calculi of lambda-conversion*. Princeton University Press.
- Curry, H.B. & Feys R. (1958) *Combinatory logic*. North-Holland Publishing Company, Amsterdam.
- Fokker, Jeroen (1992) *The systematic construction of a one-combinator basis for lambda-terms*, Formal Aspects of Computing 4:776-780. <http://www.cs.uu.nl/research/techreps/repo/CS-1989/1989-14.pdf>
- Grim, Leslie and Wallace, Mike (1984) *Robot Odyssey*, (computer software).
https://en.wikipedia.org/wiki/Robot_Odyssey The Learning Company.
- Hindley, C.J. (1986) Care of Your Pet Combinator, in Hindley, J.R. and Seldin, J.P. (1986) *Introduction to Combinators and Lambda-Calculus*, Appendix 3. Cambridge University Press.
- Hindley, J.R. and Seldin, J.P. (1986) *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press.
- James, J. M. (1993). *A calculus of number based on spatial forms*. Thesis, Master of Science in Engineering, University of Washington. Summarised here:
<http://www.wbricken.com/pdfs/01bm/06number/numbers/03boundary-alg%20copy.pdf>
- Peyton Jones, S.L. (1987) *The implementation of functional programming languages*. Prentice-Hall International.
- van der Poel et al. (1980) *New arithmetical operators in the theory of combinators*, Indag. Math. 42.
- Poundstone, William (1985) *The Recursive Universe*. Oxford University Press, Oxford.
- Révész, G.E. (1988) *Lambda-calculus, combinators and functional programming*. Cambridge tracts in

theoretical computer science v.4. Cambridge University Press, UK.

Robinett, Warren, et al. (1982) *Rocky's Boots*, (computer software).

https://en.wikipedia.org/wiki/Rocky%27s_Boots The Learning Company.

Schönfinkel, M. (1924) On the building blocks of mathematical logic, in van Heijenoort, J. (1967) *From Frege to Gödel: A source book in mathematical logic, 1879-1931*. Harvard University Press.

Smullyan, R.M. (1985) *To mock a mockingbird*. Alfred A. Knopf, New York.

Spencer-Brown, G. (1969) *Laws of Form*. George Allen and Unwin, London. 2nd edition, 1972, Julian Press, New York. 3rd edition, 1979, E.P.Dutton Paperback, New York. https://en.wikipedia.org/wiki/Laws_of_Form

Tamer, Jeff (2012) *A single-combinator-basis tattoo*. Designed using OmniGraffle. Inked by Megan Wilson of Picture Machine Tattoo. [Song-map](#), [Tattoo](#), [Tattoo closeup](#)

Other Links

Thanks to John Atwood for some of the following visual language links.

For similar notations for the Lambda calculus see Wayne Citrin et al's VEX/VIPR:

<http://users.encs.concordia.ca/~haarslev/vl95www/html-papers/citrin/citrin.html>

and see John Tromp's animated Lambda Diagrams (partly inspired by this paper), with some awesome videos:

<https://tromp.github.io/cl/diagrams.html>

For more on visual programming languages:

<http://www.cs.orst.edu/~burnett/vpl.html>

Another visual language: ToonTalk:

<http://www.toontalk.com>

Here is a 2005 paper by Henk Barendregt which deals in more depth with some of the more philosophical themes hinted at in this paper.

[Reflection and its Use, from science to meditation](#)