A Tutorial Introduction to the Lambda Calculus

Raul Rojas*

Freie Universität Berlin
Version 2.0, 2015

Abstract

This paper is a concise and painless introduction to the λ-calculus. This formalism was developed by Alonzo Church as a tool for studying the mathematical properties of effectively computable functions. The formalism became popular and has provided a strong theoretical foundation for the family of functional programming languages. This tutorial shows how to perform arithmetical and logical computations using the λ-calculus and how to define recursive functions, even though λ-calculus functions are unnamed and thus cannot refer explicitly to themselves.

1 Definition

The λ-calculus can be called the *smallest universal programming language in the world*. The λ-calculus consists of a single transformation rule (variable substitution, also called β-conversion) and a single function definition

---

*Send corrections or suggestions to rojas@inf.fu-berlin.de

scheme. It was introduced in the 1930s by Alonzo Church as a way of formalizing the concept of effective computability. The λ-calculus is universal in the sense that any computable function can be expressed and evaluated using this formalism. It is thus equivalent to Turing machines. However, the λ-calculus emphasizes the use of symbolic transformation rules and does not care about the actual machine implementation. It is an approach more related to software than to hardware.

The central concept in λ-calculus is that of "expression". A "name" is an identifier which, for our purposes, can be any of the letters a, b, c, etc. An expression can be just a name or can be a function. Functions use the Greek letter λ to mark the name of the function's arguments. The "body" of the function specifies how the arguments are to be rearranged. The identity function, for example, is represented by the string $(\lambda x.x)$. The fragment "λx" tell us that the function's argument is x, which is returned unchanged as "x" by the function.

Functions can be applied to other functions. The function A, for example, applied to the function B, would be written as AB. In this tutorial, capital letters are used to represent functions. In fact, anything of interest in λ-calculus is a function. Even numbers or logical values will be represented by functions that can act on one another in order to transform a string of symbols into another string. There are no types in λ-calculus: any function can act on any other. The programmer is responsible for keeping the computations sensible.

An expression is defined recursively as follows:

$$< \text{expression} > \quad := \quad < \text{name} > | < \text{function} > | < \text{application} >$$
$$< \text{function} > \quad := \quad \lambda < \text{name} > . < \text{expression} >$$
$$< \text{application} > \quad := \quad < \text{expression} > < \text{expression} >$$

An expression can be surrounded by parenthesis for clarity, that is, if E is an expression, (E) is the same expression. Otherwise, the only keywords used in the language are λ and the dot. In order to avoid cluttering expressions with parenthesis, we adopt the convention that function application associates

from the left, that is, the composite expression

$$\mathsf{E_1 E_2 E_3 \ldots E_n}$$

is evaluated applying the successive expressions as follows

$$\left(\ldots \left(\left(\mathsf{E_1 E_2} \right) \mathsf{E_3} \right) \ldots \mathsf{E_n} \right)$$

As can be seen from the definition of λ-expressions, a well-formed example of a function is the previously mentioned string, enclosed or not in parentheses:

$$\lambda x.x \equiv (\lambda x.x)$$

We use the equivalence symbol "$\equiv$" to indicate that when $\mathsf{A} \equiv \mathsf{B}$, A is just a synonym for B. As explained above, the name right after the λ is the identifier of the argument of this function. The expression after the point (in this case a single x) is called the "body" of the function's definition.

Functions can be applied to expressions. A simple example of an application is

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

This is the identity function applied to the variable y. Parenthesis help to avoid ambiguity. Function applications are evaluated by substituting the "value" of the argument x (in this case the y being processed) in the body of the function definition. Fig. 1 shows how the variable y is "absorbed" by the function (red line), and also shows where it is used as a replacement for x (green line). The result is a reduction, represented by the right arrow, with the final result y.

Since we cannot always have pictures, as in Fig. 1, the notation $[y/x]$ is used to indicate that all occurrences of x are substituted by y in the function's body. We write, for example, $(\lambda x.x)y \to [y/x]x \to y$. The names of the arguments in function definitions do not carry any meaning by themselves. They are just "place holders", that is, they are used to indicate how to rearrange the arguments of the function when it is evaluated. Therefore all the strings below represent the same function:

$$(\lambda z.z) \equiv (\lambda y.y) \equiv (\lambda t.t) \equiv (\lambda u.u)$$

This kind of purely alphabetical substitution is also called α-reduction.

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

$$\rightarrow y \qquad\qquad \rightarrow y$$

Figure 1: The same reduction shown twice. The symbol for the function's argument is just a place holder.

1.1 Free and bound variables

If we only had pictures of the plumbing of λ-expressions, we would not have to care about the names of variables. Since we are using letters as symbols, we have to be careful if we repeat them, as shown in this section.

In λ-calculus all names are local to definitions (like in most programming languages). In the function $\lambda x.x$ we say that x is "bound" since its occurrence in the body of the definition is preceded by λx. A name not preceded by a λ is called a "free variable". In the expression

$$(\lambda \mathbf{x}.\mathbf{x})(\lambda \mathbf{y}.\mathbf{y}x)$$

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 following x is free. Bound variables are shown in bold face. It is very important to notice that this x in the second expression is totally independent of the x in the first expression. This can be more easily seen if we draw the "plumbing" of the function application and the consequent reduction, as shown in Fig. 2.

In Fig. 2 we see how the symbolic expression (first row) can be interpreted as a kind of circuit, where the bound argument is moved to a new position inside the body of the function. The first function (the identity function) "consumes" the second one. The symbol x in the second function has no connections with the rest of the expression, it is floating free inside the function definition.

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

$$(\lambda \blacksquare . \blacksquare)(\lambda \square . x)$$

$$\rightarrow (\lambda \square . x)$$

Figure 2: In successive rows: The function application, the "plumbing" of the symbolic expression, and the resulting reduction.

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

- <name> is free in <name>.
  (Example: a is free in a).

- <name> is free in λ<name$_1$>. <exp> if the identifier <name>$\neq$<name$_1$> and <name> is free in <exp>.
  (Example: y is free in $\lambda x.y$).

- <name> is free in $E_1 E_2$ if <name> is free in E_1 or if it is free in E_2.
  (Example: x is free in $(\lambda x.x)x$).

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

- <name> is bound in λ <name$_1$>. <exp> if the identifier <name>=<name$_1$> or if <name> is bound in <exp>.
  (Example: x is bound in $(\lambda y.(\lambda x.x))$).

- <name> is bound in $E_1 E_2$ if <name> is bound in E_1 or if it is bound in E_2.
  (Example: x is bound in $(\lambda x.x)x$).

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

$$(\lambda\mathbf{x}.\mathbf{x}y)(\lambda\mathbf{y}.\mathbf{y})$$

the first y is free in the parenthesized subexpression to the left, but it is bound in the subexpression to the right. Therefore, it occurs free as well as bound in the whole expression (the bound variables are shown in bold face).

1.2   Substitutions

The more confusing part of standard λ-calculus, when first approaching it, is the fact that we do not give names to functions. Any time we want to apply a function, we just write the complete function's definition and then proceed to evaluate it. To simplify the notation, however, we will use capital letters, digits and other symbols (san serif) as synonyms for some functions. The identity function, for example, can be denoted by the letter I, using it as shorthand for $(\lambda x.x)$.

The identity function applied to itself is the application

$$\mathsf{II} \equiv (\lambda x.x)(\lambda x.x).$$

In this expression, the first x in the body of the first function in parenthesis is independent of the x in the body of the second function (remember that the "plumbing" is local). Just to emphasize the difference we can in fact rewrite the above expression as

$$\mathsf{II} \equiv (\lambda x.x)(\lambda z.z).$$

The identity function applied to itself

$$\mathsf{II} \equiv (\lambda x.x)(\lambda z.z)$$

yields therefore

$$[(\lambda z.z)/x]x \rightarrow \lambda z.z \equiv \mathsf{I},$$

that is, the identity function again.

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

y bound in the subexpression

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

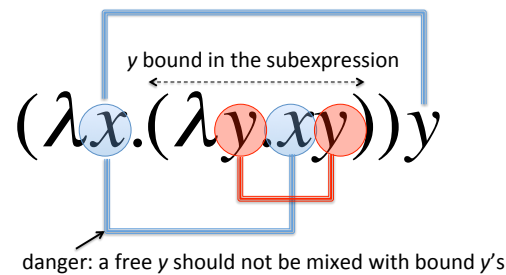danger: a free *y* should not be mixed with bound *y*'s

Figure 3: A free variable should not be substituted in a subexpression where it is bound, otherwise a new "plumbing", different to the original, would be generated.

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

$$\big(\lambda \mathbf{x}.(\lambda \mathbf{y}.\mathbf{xy})\big)y$$

the function on the left contains a bound y, whereas the y on the right is free. An incorrect substitution would mix the two identifiers in the erroneous result

$$(\lambda \mathbf{y}.y\mathbf{y}).$$

Simply by renaming the bound y to t we obtain

$$\big(\lambda x.(\lambda t.xt)\big)y \rightarrow (\lambda t.yt)$$

which is a completely different result but nevertheless the correct one.

Therefore, if the function $\lambda x. <\exp>$ is applied to E, we substitute all *free* occurrences of x in $<\exp>$ with E. If the substitution would bring a free variable of E in an expression where this variable occurs bound, we rename

the bound variable before performing the substitution. For example, in the expression

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

we associate the first x with y. In the body

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

only the first x is free and can be substituted. Before substituting though, we have to rename the variable y to avoid mixing its bound with its free occurrence:

$$[y/x]\,(\lambda t.(x(\lambda x.xt))) \rightarrow (\lambda t(y(\lambda x.xt)))$$

In normal order reduction we reduce always the left most expression of a series of applications first. We continue until no further reductions are possible.

2 Arithmetic

A programming language should be capable of specifying arithmetical calculations. Numbers can be represented in the λ-calculus starting from zero and writing "successor of zero", that is "suc(zero)", to represent 1, "suc(suc(zero))" to represent 2, and so on. Since in λ-calculus we can only define new functions, numbers will be defined as functions using the following approach: zero can be defined as

$$\lambda s.(\lambda z.z)$$

This is a function of two arguments s and z. We will abbreviate such expressions with more than one argument as

$$\lambda sz.z$$

It is understood here that s is the first argument to be substituted during the evaluation and z the second. Using this notation, the first natural numbers can be defined as

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

and so on.

The big advantage of defining numbers in this way is that we can now apply a function f to an argument a any number of times. For example, if we want to apply f to a three times we apply the function 3 to the arguments f and a yielding:

$$3fa \to (\lambda sz.s(s(sz)))fa \to f(f(fa)).$$

This way of defining numbers provides us with a language construct similar to an instruction such as "FOR i=1 to 3" in other languages. The number zero applied to the arguments f and a yields $0fa \equiv (\lambda sz.z)fa \to a$. That is, applying the function f to the argument a *zero times* leaves the argument a unchanged.

Our first interesting function, after having defined the natural numbers, is the successor function. This can be defined as

$$S \equiv \lambda nab.a(nab).$$

The definition looks awkward but it works. For example, the successor function applied to our representation for zero is the expression:

$$S0 \equiv (\lambda nab.a(nab))0$$

In the body of the first expression we substitute the occurrence of n with 0 and this produces the reduced expression:

$$\lambda ab.a(0ab) \to \lambda ab.a(b) \equiv 1$$

That is, the result is the representation of the number 1 (remember that bound variable names are "dummies" and can be changed).

Successor applied to 1 yields:

$$S1 \equiv (\lambda nab.a(nab))1 \to \lambda ab.a(1ab) \to \lambda ab.a(ab) \equiv 2$$

Notice that the only purpose of applying the number $1 \equiv (\lambda sz.sz)$ to the arguments a and b is to "rename" the variables used internally in the definition of our number.

2.1 Addition

Addition can be obtained immediately by noting that the body sz of our definition of the number 1, for example, can be interpreted as the application of the function s on z. If we want to add say 2 and 3, we just apply the successor function two times to 3.

Let us try the following in order to compute 2+3:

$$2S3 \equiv (\lambda sz.s(sz)))S3 \to S(S3) \to \cdots \to 5$$

In general m plus n can be computed by the expression mSn.

2.2 Multiplication

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

$$(\lambda xya.x(ya))$$

The product of 3 by 3 is then

$$(\lambda xya.x(ya))33$$

which reduces to

$$(\lambda a.3(3a))$$

Using the definition of the number 3, we further reduce the above expression to

$$(\lambda a.(\lambda sb.s(s(sb)))(3a)) \to (\lambda ab.(3a)((3a)((3a)b)))$$

In order to understand why this function really computes the product of 3 by 3, let us look at some diagrams. The first application $(3a)$ is computed on the left of Fig. 4 . Notice that the application of 3 to a has the effect of producing a new function which applies a three times to the function's argument.

Now, applying the function 3 to the result of $(3a)$ produces three copies of the function obtained in Fig. 4 , concatenated as shown on the right in Fig. 4
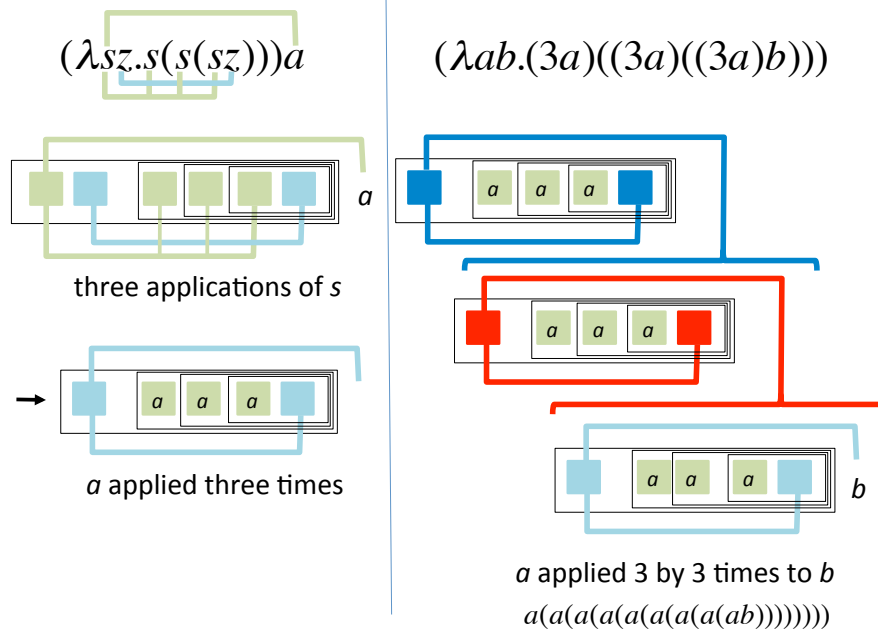
$(\lambda sz.s(s(sz)))a$ $(\lambda ab.(3a)((3a)((3a)b)))$

three applications of *s*

a applied three times

a applied 3 by 3 times to *b*

$a(a(a(a(a(a(a(a(ab))))))))$

Figure 4: Left: The number 3 applied to an argument a produces a new function. Right: The plumbing of the function 3 applied to 3a, and the result to b.

(where the result has been applied to b for clarity). Notice that we have a "tower" of three times the same function, each one absorbing the lower one as argument for the application of the function a three times, for a total of nine applications.

3   Conditionals

We introduce the following two functions which we call the values "true"

$$\mathsf{T} \equiv \lambda xy.x$$

and "false"

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

The first function takes two arguments and returns the first one. The second function returns the second of two arguments.

3.1 Logical operations

It is now possible to define logical operations using this representation of the truth values.

The AND function of two arguments can be defined as

$$\wedge \equiv \lambda xy.xyF$$

This definition works because given that x is true, the truth value of the AND operation depends on the truth value of y. If x is false (and selects thus the second argument in xyF) the complete AND is false, regardless of the value of y.

The OR function of two arguments can be defined as

$$\vee \equiv \lambda xy.xTy$$

Here, if x is true, the OR is true. If x is false, it picks the second argument y and the value of the OR function depends now on the value of y.

Negation of one argument can be defined as

$$\neg \equiv \lambda x.xFT$$

For example, the negation function applied to "true" is

$$\neg T \equiv (\lambda x.xFT)T$$

which reduces to

$$TFT \equiv (\lambda cd.c)FT \rightarrow F$$

that is, the truth value "false".

Armed with this three logic functions we can encode any other logic function and reproduce any given circuit without feedback (we look at feedback when we deal with recursion).

3.2　A conditional test

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 fulfills this role:

$$\mathsf{Z} \equiv \lambda x.x \mathsf{F} \neg \mathsf{F}$$

To understand how this function works, remember that

$$\mathsf{0} f a \equiv (\lambda s z.z) f a = a$$

that is, the function f applied zero times to the argument a yields a. On the other hand, F applied to any argument yields the identity function

$$\mathsf{F} a \equiv (\lambda x y.y) a \to \lambda y.y \equiv \mathsf{I}$$

We can now test if the function Z works correctly. The function applied to zero yields

$$\mathsf{Z0} \equiv (\lambda x.x \mathsf{F} \neg \mathsf{F}) \mathsf{0} \to \mathsf{0F} \neg \mathsf{F} \to \neg \mathsf{F} \to \mathsf{T}$$

because F applied $\mathsf{0}$ times to $\neg$ yields $\neg$. The function Z applied to any other number N yields

$$\mathsf{ZN} \equiv (\lambda x.x \mathsf{F} \neg \mathsf{F}) \mathsf{N} \to \mathsf{NF} \neg \mathsf{F}$$

The function F is then applied N times to $\neg$. But F applied to anything is the identity (as shown before), so that the above expression reduces, for any number N greater than zero, to

$$\mathsf{IF} \to \mathsf{F}$$

3.3 The predecessor function

We can now define the predecessor function combining some of the functions 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 λ-calculus using the function

$$(\lambda z.zab)$$

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

$$(\lambda z.zab)\mathsf{T} \to \mathsf{T}ab \to a,$$

and the second applying the function to F

$$(\lambda z.zab)\mathsf{F} \to \mathsf{F}ab \to b.$$

The following function generates from the pair $(n, n - 1)$ (which is the argument p in the function) the pair $(n + 1, n)$:

$$\Phi \equiv (\lambda pz.z(\mathsf{S}(p\mathsf{T}))(p\mathsf{T}))$$

The subexpression $p\mathsf{T}$ extracts the first element from the pair p. A new pair is formed using this element, which is incremented for the first position of the new pair and just copied for the second position of the new pair.

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

$$\mathsf{P} \equiv (\lambda n.(n\Phi(\lambda z.z00))\mathsf{F})$$

Notice that using this approach the predecessor of zero is zero. This property is useful for the definition of other functions.

3.4 Equality and inequalities

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:

$$\mathsf{G} \equiv (\lambda xy.\mathsf{Z}(x\mathsf{P}y))$$

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

If $x \geq y$ and $y \geq x$, then $x = y$. This leads to the following definition of the function E which tests if two numbers are equal:

$$\mathsf{E} \equiv (\lambda xy. \wedge (\mathsf{Z}(x\mathsf{P}y))(\mathsf{Z}(y\mathsf{P}x)))$$

In a similar manner we can define functions to test whether $x > y$, $x < y$ or $x \neq y$.

4   Recursion

Recursive functions can be defined in the λ-calculus using a function which calls a function y and then regenerates itself. This can be better understood by considering the following function Y:

$$\mathsf{Y} \equiv (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx)))$$

This function applied to a function R yields:

$$\mathsf{YR} \equiv (\lambda x.\mathsf{R}(xx))(\lambda x.\mathsf{R}(xx))$$

which further reduced yields

$$\mathsf{R}((\lambda x.\mathsf{R}(xx))(\lambda x.\mathsf{R}(xx)))$$

but this means that $\mathsf{YR} \rightarrow \mathsf{R}(\mathsf{YR})$, that is, the function R is evaluated using the recursive call YR as the first argument.

An infinite loop, for example, can be programmed as YI, since this reduces to I(YI), then to YI and so ad infinitum.

A more useful function is one which adds the first n natural numbers. We can use a recursive definition, since $\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i$. Let us use the following definition for R:

$$R \equiv (\lambda rn.\mathsf{Z}n\mathsf{0}(n\mathsf{S}(r(\mathsf{P}n))))$$

This definition tells us that the number n is tested: if it is zero the result of the sum is zero. In n is not zero, then the successor function is applied n times to the recursive call (the argument r) of the function applied to the predecessor of n.

How do we know that r in the expression above is the recursive call to R, since functions in λ-calculus do not have names? We do not know and that is precisely why we have to use the recursion operator Y. Assume for example that we want to add the numbers from 0 to 3. The necessary operations are performed by the call:

$$\mathsf{YR3} \rightarrow \mathsf{R(YR)3} \rightarrow \mathsf{Z30(3S(YR(P3)))}$$

Since 3 is not equal to zero, the evaluation is equivalent to

$$\mathsf{3S(YR(P3))}$$

that is, the sum of the numbers from 0 to 3 is equal to 3 plus the sum of the numbers from 0 to the predecessor of 3 (that is, two). Successive recursive evaluations of YR will lead to the correct final result.

Notice that in the function defined above the recursion will be stopped when the argument becomes 0. The final result will be

$$\mathsf{3S(2S(1S0))}$$

that is, the number 6.

5   Projects for the reader

1. Define the functions "less than" and "greater than" of two numerical arguments.

2. Define the positive and negative integers using pairs of natural numbers.

3. Define addition and subtraction of integers.

4. Define the division of positive integers recursively.

5. Define the function $n! = n \cdot (n-1) \cdots 1$ recursively.

6. Define the rational numbers as pairs of integers.

7. Define functions for the addition, subtraction, multiplication and division of rationals.

8. Define a data structure to represent a list of numbers.

9. Define a function which extracts the first element from a list.

10. Define a recursive function which counts the number of elements in a list.

11. Can you simulate a Turing machine using λ-calculus?

References

[1] P.M. Kogge, *The Architecture of Symbolic Computers*, McGraw-Hill, New york 1991, chapter 4.

[2] G. Michaelson, *An Introduction to Functional Programming through λ-calculus*, Addison-Wesley, Wokingham, 1988.

[3] G. Revesz, *Lambda-Calculus Combinators and Functional Programming*, Cambridge University Press, Cambridge, 1988, chapters 1-3.