

Handout 1: Introduction to Lambda Calculus

1. Historical background The lambda calculus was developed by Alonzo Church in the early 1930's to serve as a foundation for higher-order logic. The term "higher-order" refers to the fact this system has not only functions and predicates on ordinary values, but also functions and predicates on other functions and predicates. Church actually developed a typed version of the lambda calculus first and later considered the calculus without types. Unfortunately, the untyped calculus was not suitable as a foundation for logic because Kleene found a paradox. However, the calculus became successful as a pure calculus of functions, ignoring the logic aspect.

The emphasis on "pure calculus of functions" was due to Haskell Curry, who independently invented a system called *combinatory calculus* in order to study variables and substitutions. (Treating variables correctly was a challenge in the early days of mathematical logic. In fact, Church's initial definitions had bugs!) It turned out that the combinatory calculus was equivalent to the lambda calculus and have very similar ideas. So, in modern treatments, we regard the lambda calculus and combinatory calculus as being essentially the same system.

Church also noticed that the lambda calculus was able to express computable functions and considered the question of the expressive power of the various formalisms for computable functions (lambda calculus, Turing machines, Post systems etc.) He proved that the functions on integers expressible in all the formalisms were exactly the same, and formulated the famous *Church's thesis*, which states that the class of computable functions is independent of the formalism used for expressing them.

Church's ideas were in currency in the early development of Computer Science. Alan Turing, Stephen Kleene, Dana Scott, John McCarthy and others were associated with Church and knew about the lambda calculus. In Britain, Roger Penrose is said to have popularised the ideas of lambda calculus among computer scientists. As a result, the theory of programming languages came to use the lambda calculus as the foundational system for studying all the concepts related to programming. This influence persists till this day.

2. Basic intuitions We can say that there are three basic intuitions underlying the formulation of lambda calculus:

- *Functions are values.*
- *Functions need not be named.* (More positively, functions can be formulated using expression notations without having to name them.)
- *Functions are all that one needs.*

We consider each of these ideas in turn.

3. Functions are values In ordinary mathematical notation, we use function application as a building block in writing expressions. Here is a simple example of function application:

$$\sin(x)$$

Here, \sin is the name of a function and x is an expression that denotes a value (a real number, in this case). The result of the function application is again a value. What then is a function? Looking at the example, we can say that functions are some form of things that take values as arguments and yield values as results. This idea is captured in the set-theoretic notation for declaring a function:

$$\sin : R \rightarrow R$$

Here R stands for the type of real numbers. The declaration says basically that \sin takes real numbers as arguments and yields real numbers as results. This gives us some idea of what functions are like. But, what exactly are functions? What do we mean by "some form of things"? There are two views on this.

4. Two views of functions One view of functions is that they are *rules for calculation*, using which we can calculate their results from the values of the arguments. A second view of functions is that they are *mappings* that associate, for each argument value, a corresponding result value.

The second view is more *abstract*. Why? We can imagine many rules for calculation for what is essentially the same function. Here are two rules for a function f :

$$\begin{aligned} f(x) &= x^2 - 4 \\ f(x) &= (x + 2)(x - 2) \end{aligned}$$

The two rules involve quite different operations. However, for every real number x , both the rules produce the same result. Abstractly, we say that they represent the same “mapping”. For a more computing-oriented example, consider a function *sort* that maps lists of integers to sorted lists of integers. For example,

$$\text{sort}([4, 8, 2]) = [2, 4, 8]$$

There are many ways to define *sort*, i.e., there are many sorting procedures. However, all of them denote the same mapping from lists to lists.

The second view is also more *general*. There are mappings for which there are no rules for calculation. Only computable functions have rules for calculation. The halting problem, for instance, is the problem of finding a rule for function $H(t, x)$ whose result is 1 if the Turing machine encoded by t halts when run on the input x , 0 if it does not halt when run on x . H is a perfectly well-defined mapping. But there is no rule or procedure that can calculate the result.

In Computer Science, we often alternate between the two views of functions.

5. Functions as values

Using either view, we can regard functions as entities that can be thought of as “values”. By saying that they are “values”, we are admitting the possibility that we can apply other functions to such “values” and obtain such “values” as results of other functions. So, we can start off with some basic values and first consider functions on such basic values. We call them “first-order functions”. We can then think of all first-order functions as values, and consider functions that take such functions as arguments or produce such functions as results. Doing so, we obtain “second-order functions”, i.e., functions that operate on first-order functions. Similarly, we can consider functions that operate on second-order functions and so on *ad infinitum*.

In Calculus, we have an operation of “derivative”, e.g.,

$$\mathcal{D}(\sin) = \cos$$

The derivative of the sin function (which is a first-order function) is the cos function. So, the derivative operation is a second-order function. For a more routine programming example, consider a sorting function that takes, in addition to a list argument an additional argument for the comparison function that it should use to arrange the list elements. If we pass $<$ as the comparison function then we get lists sorted in ascending order. If we pass $>$ as the comparison function then we get lists sorted in descending order:

$$\begin{aligned}\text{sort}(<, [4, 8, 2]) &= [2, 4, 8] \\ \text{sort}(>, [4, 8, 2]) &= [8, 4, 2]\end{aligned}$$

Notice that *sort* is now a second-order function. One of its arguments is a first-order function.

One might wonder if we really need functions of arbitrary higher orders. Second-order functions are already pretty fancy. Why would one need third-order functions or fourth-order functions etc.? Indeed, such functions are quite rare in practical use. However, saying that we will admit functions of arbitrary order simplifies our theory. Once we admit functions as values, there is no particular advantage to limiting the orders of functions. So, we might as well treat functions of all orders.

6. Functions need not be named

In ordinary mathematical notation, we give names to functions as soon as we consider them. For instance, we say let f be the function such that

$$f(x) = x^2 - 4$$

In explicit words, we are saying “let f be the function that maps every argument value x to the corresponding result value of $x^2 - 4$.” Church devised a simple notation that can express the locution “the function that maps ... to ...”, using the Greek letter λ . Using Church’s notation, the function that maps x to $x^2 - 4$ can be expressed as:

$$\lambda x. x^2 - 4$$

Here λ is to be thought of as “the function that maps”. The variable x is the parameter for the function. This is what we are “mapping”. The expression $x^2 - 4$ is the result of the function. This is what x is getting mapped to. The form of the expression constructed in this way is called a “ λ -abstraction.” It is also common to call it an “anonymous function” emphasizing the fact that we haven’t given the function a name. (But don’t take this idea too seriously. Nothing stops us from giving a name to the function from the outside. What we have done is to separate the naming issue from that of constructing functions themselves.)

The idea of constructing functions without giving names is just a notational trick. But by using the right notation, we are able to unleash the power of what can be expressed.

7. Parameter symbols are unimportant

Note that the variable symbol x in the expression $\lambda x. x^2 - 4$ is quite immaterial. We could have used any other variable symbol in its place. For example $\lambda y. y^2 - 4$ means exactly the same thing. (The function that maps “any given x to $x^2 - 4$ ” and the one that maps “any given y to $y^2 - 4$ ” are the same thing!) Some argument is going to be given. We are giving it a temporary name just so we can refer to it in the result expression. What name we give it makes no difference.

This idea is expressed as the principle called “ α equivalence” for λ -abstraction terms. We can freely rename a parameter symbol to another symbol and use the new symbol in place of the old one throughout the result term. The term obtained by doing this translation is equivalent to the original term. For example:

$$(\lambda x. x^2 - 4) = (\lambda y. y^2 - 4)$$

8. Parameter symbols are bound variables

Another point to note about the meaning of the lambda notation is that the parameter symbol is a *temporary name* given to the argument the function will be provided with. The scope of this parameter is just the body of the function. Outside this scope, the parameter symbol has no meaning. It can be used for other purposes. For example, consider the expression:

$$(\lambda x. x^2 - 4) ((\lambda x. x/8) (29))$$

We are applying to the argument 29 the function $\lambda x. x/8$ (the function that divides by 8) and the result of the function is then given as argument to $\lambda x. x^2 - 4$. There is nothing unusual about the same parameter symbol x being used in both the function terms. The scope of each x is delineated by the brackets enclosing each lambda term.

We say that the symbol x is “bound” in the λ -abstraction term $\lambda x. x^2 - 4$. “Bound” is used in the sense of “being given a specific meaning”. Outside the scope of the λ -abstraction term, the meaning is not present.

A lambda term can also have variables that are not bound. Such variables are said to be “free”. For example, in the term $\lambda x. x^2 - y$, the variable x is bound and the variable y is free. Being “free” means “not given any specific meaning”. Since it is free, we can say from the outside “set $y = 4$ in the term $\lambda x. x^2 - y$ ” and that would mean the same as $\lambda x. x^2 - 4$. On the other hand, we can’t say “set $x = 4$ in the term $\lambda x. x^2 - y$ ”. Why not? The symbols x is owned by the λ -abstraction. It is not free to be given some other meaning from the outside.

The idea of bound variables is similar to that of “local” variables in programming languages. Free variables are correspondingly “non-local” variables.

9. Functions are all that we need

This is a very deep (and also very strange) principle that Church used in devising lambda calculus. What it means is that we don’t need to postulate any entities that we might call “basic” values. We just have functions and they are good enough for everything we want. So, one might wonder what are these functions acting on, if there are no basic values at all? That is mysterious. Like the old lady said, “it is turtles all the way!”

In reality, what happens is that we can write lambda terms with free variables x, y, z, \dots , and these variables can stand for anything. So, we don’t have to explicitly postulate that there are some basic values. The free variables *could* be used to plug in basic values if we wanted. So, it is not all that mysterious after all.

Church was able to represent all kinds of data values, such as boolean values, integers, lists etc., using functions in the lambda calculus. So, lambda calculus can remain a very compact language without giving up any expressive power.

10. Curried functions

One particular idea we make use of immediately is to use functions to represent multiple argument situations. Consider, for example, the following function that takes two real number arguments:

$$f(x, y) = \sqrt{x^2 + y^2}$$

We normally write the type of f as $R \times R \rightarrow R$. This is a prototypical example of a multiple-argument function. Church and Curry did not wish to complicate their language by adding multiple-argument functions of this kind. All functions in the lambda calculus are unary. However, we can still represent the effect of multiple argument functions using multiple λ -abstractions. For example, the effect of the function f above can be represented in the lambda calculus as follows:

$$\lambda x. (\lambda y. \sqrt{x^2 + y^2})$$

On the surface, this seems straightforward. There are two λ ’s which accept two arguments x and y respectively, and, eventually, the right result is being returned. However, underneath the surface, this term is quite sophisticated.

The outer λ represents a function that takes a real number x as argument, but its result is a *function*, represented by the inner λ term. This result function takes another real number y as an argument, and returns a result which is calculated by combining the original argument x of the first function and the argument y of the current function. To give some intuition for how this works, we show the effect of this term in a Java-like notation with nested function declarations:

```

T f(float x) {
    float g(float y) {
        return sqrt(x*x + y*y);
    }
    return g;
}

```

Note that the outer function f creates an inner function g and returns it as its result. This inner function remembers the original argument x . But it must be called with another argument y in order to obtain the final result. For example, one might see the sequence of calls:

```
g3 = f(3); z = g3(4);
```

to calculate the diagonal of a rectangle with sides 3 and 4. The result would be 5.

In lambda calculus, we don't use such a laborious notation. If f stands for the double abstraction term displayed above, then all we need to say is:

$$(f(3))(4)$$

This means that we first apply f to argument 3. The result is a function which we then apply to 4. (We will shortly introduce bracketing conventions that simplify the notation further.)

Representing multiple argument functions in terms of unary functions in this way is referred to as "Currying", named after Haskell Curry, who popularised the notation.

11. The syntax of lambda calculus The set of lambda terms is inductively defined as follows:

- Any variable x is a lambda term.
- If M is a lambda term and x is a variable (typically a variable that occurs in M), then $\lambda x. M$ is a lambda term.
- If M_1 and M_2 are lambda terms, then $M_1 M_2$ is a lambda term.

The notation $M_1 M_2$ means " M_1 applied to M_2 ." The term M_1 denotes a function and the term M_2 is being provided as an argument. In conventional mathematical notation, this would have been written as $M_1(M_2)$. But the additional brackets around the argument term are not used in the lambda calculus because they add notational clutter.

The syntax can be written using production rule notation as:

$$M ::= x \mid \lambda x. M' \mid M_1 M_2$$

As you can see, this is a very small and compact language.

This is Church's lambda calculus, which is also referred to as the "pure lambda calculus." The emphasis "pure" refers to the fact that there is nothing else other than functions in this calculus. In practice, however, we use lambda calculus with additional constants for integers, truth values, and the primitive operations on them. Let c stand for constant symbols. Then we use the syntax:

$$M ::= x \mid c \mid \lambda x. M' \mid M_1 M_2$$

A lambda calculus extended with constants is called an "applied lambda calculus".

12. Bracketing conventions

Brackets must be used to disambiguate the syntax of terms, as is usual in any formal language. However, since the lambda calculus is so sparse, rather too many brackets would be needed. Two powerful bracketing conventions have been developed to minimize the number of brackets needed. These are as follows:

- A λ -abstraction term extends as far to the right as possible.
- Whenever there are two application operations side by side, we imagine brackets to the left, i.e., $M N K$ means $(M N) K$.

Let us consider the term

$$(\lambda x. x^2 - 4) ((\lambda x. x/8) (29))$$

and see what brackets can be dropped.

- The brackets around the first λ -abstraction are quite necessary. If we dropped them, we would have

$$\lambda x. x^2 - 4 ((\lambda x. x/8) (29))$$

Since the λ -abstraction term extends as far to the right as possible, it would mean that the body of outermost λ function is

$$x^2 - 4 ((\lambda x. x/8) (29))$$

which is not very sensible. (It appears as if the constant 4 is being applied to a lambda term, but 4 does not denote a function.)

- We do not need additional brackets around the body of the λ abstraction, i.e., we do not need to write $(\lambda x. (x^2 - 4))$. Why need brackets around a term only when it participates in a function application operation.
- The second pair of brackets in the term are also quite necessary. If we dropped them, we would obtain:

$$(\lambda x. x^2 - 4) (\lambda x. x/8) (29)$$

By bracketing convention this means that the term $(\lambda x. x^2 - 4)$ is applied to $(\lambda x. x/8)$ and the result of this application is then applied to 29. But that is not our intent.

- The brackets around the second λ -abstraction term are also necessary. If we drop them then “(29)” would become part of the body of the λ -function.
- The brackets around 29 can be dropped. There is rarely any need to put brackets around a single symbol.

So, a slightly more economical form of the term is:

$$(\lambda x. x^2 - 4) ((\lambda x. x/8) 29)$$

It would seem that the brackets are used in lambda terms in quite an opposite way to normal notation. Normally, we put brackets around the arguments of a function and none around the function, e.g., $f(29)$. But in writing lambda terms, we often put brackets around the function but not the argument, e.g., $(\lambda x. x/8) 29$.

This requires some *practice* and *careful parsing* before you will get used to it.

13. Sample applied lambda calculus

We will use, for examples, a sample applied lambda calculus with the following constants:

- boolean values: true and false, and the operation “if”.
- integers: $\dots, -2, -1, 0, 1, 2, \dots$
- arithmetic operations: “+”, “-”, “*”, “/”, “mod”.
- comparison operations: “=”, “≠”, “<”, “≤”, “>”, “≥”.

Officially, all of these are constants in the syntax of applied lambda calculus. Binary operations like “+” and “=” are treated as Curried functions. So, the official syntax for the expression $x^2 - 4$ is

$$- (* x x) 4$$