Computer Science G12

Summary of class Thu. Nov. 16 2017

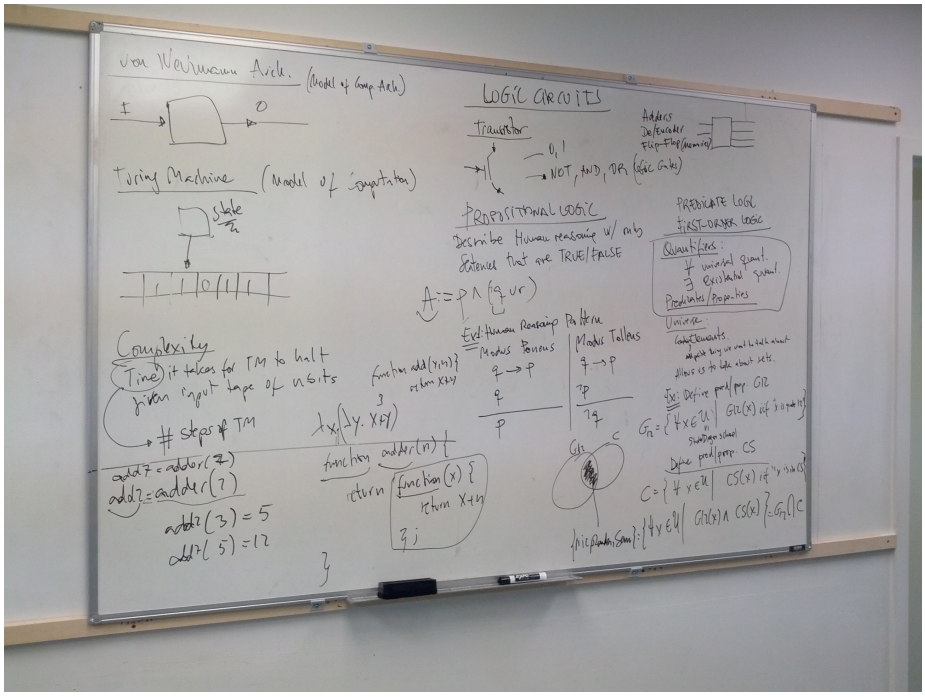

Figure 1: Whiteboard: summary of what we have seen in Term1

λ : Lambda Calculus

Video: Lambda Calculus in Youtube, by Computerphile. Explains, Prof. Graham Hutton, from the University of Nottingham.

1. What is Lambda Calculus?
2. Why is it important?
3. Where did it come from?

Where did Lambda Calculus come from?

- Who *invented* λ–calculus? Alonzo Church (1903 - 1995). He was born the same year as John von Neumann (so, no he wasn't the PhD. advisor of A.

Church)

- Where? Princeton (well, many bright minds coincided there during the same time, von Neumann and Einstein among others. But also Alan Turing, who was a student of A. Church)
- How did he come up with this? **A. Church set out to formalize** (i.e., to describe in the language of logic) **the concept of a *function***.

Why is it important?

Alan Turing was the father of what we studied as Turing Machine. As we said, this is a *model of computation*. Well, it so happens that λ-calculus turned out to gives rise to an *alternative model of computation*.

The Church-Turing Hypothesis: Here alternative is meant simply as *looking differently* or as *offering a different perspective*. Namely, it was later shown, that both views are equivalent (as in equivalence in Propositional logic!). That is, if a function is computable as a Turing Machine, it can be translated into λ-calculus, and vice-versa, if it can be written in the latter, we can find a Turing Machine that computes the same function! This is what is meant by the *Church-Turing hypothesis*[1] in computer science.

That is what we mean by both models being equivalent: *Neither of them gives you more or less juice than the other*.

What is λ Calculus?

First, what is a Function ? (for Church, but also for everybody)

Is a black box which takes, say, one input (x) and gives one output (x+1).

It can also take more inputs, but it will still be given just one output -we don't need more.

The following properties are essential to what we consider as a function:

1. **They are black boxes**, i.e., we don't know what the internals are, just what they do for each input! Example: "I feed in a 3 and it spits out a 4; for a 4 I get a 5; for 5, 6; etc."
2. **No side effects!** That means, **no internal concept of *state*** . This is a key difference with a Turing Machine.

[1] Of course, once the Church-Turing Hypothesis was proved, it no longer was an hypothesis, but a theorem. Nevertheless, it's still referred to as hypothesis just for historical reasons -kind of saying, "yo, you remember those exciting and glorious days when people didn't have yet a full view of things and wandered around like explorers in the realm of Logic and Mathematics? All full of mysteries".

λ Calculus

In λ-calculus, a function is written formally as

$$\lambda x.\text{expression containing x}$$

For instance, consider the function mentioned above. In λ-calculus it is written as

$$\lambda x.x + 1$$

Period. No parenthesis, no equal signs... Yet, it's clearly equivalent to *the way we express things* in mathematics, isn't? There it would be $f(x) = x + 1$. Somehow, you could see λ-calculus as a ***new notation*** **system for talking about functions!**

> **Connecting dots**[2]:
>
> Now, during class, we mentioned in passing, and already twice, what First-Order (Predicate) Logic (FOL) is. It's an extension of PL (propositional logic) that allows us to talk as well about which students do what, say, and this leads immediately to the concept of *set* and the operations and relations that we can define among sets (e.g., intersection, or inclusion). There, in FOL a predicate is nothing more than a property that the members of a given set satisfy. And in expressing this last sentence in an as clear and unambiguous way as possible, we end up writing things like $P(a)$ for expressing that an element a (of a given *universe of discourse* $\mathcal{U}$) satisfy the property P. But this is exactly how we write functions!!
>
> Evidently, the coincidence is not random: Basically, any function can be understood as a predicate, defining therefore a particular set of elements.
>
> On the other hand, from what you already know about math, a function describes a *calculation*. And, remember, calculation and *computation* are synonyms.
>
> Whence, we have
>
> $$\text{Computation} \leftrightarrow \text{Functions} \leftrightarrow \text{Sets/Predicates} \leftrightarrow FOL \leftrightarrow LogicCircuits \leftrightarrow$$
>
> $$\leftrightarrow \text{Computer Architecture} \leftrightarrow \text{Computation}$$
>
> $$FOL \leftrightarrow \text{HR} \leftrightarrow Computation$$

[2] ***Connecting dots*** are, well like this very same summary, namely, an opportunity to investigate a topic *breath wise* instead of *depth wise* and looking for relations to stuff we have already seen before. This kind of intellectual training is paramount to master a topic. Later on, if you end up doing research in any way, you will see that this very same exercise becomes a driving force for opening new lines of research. But I digress.

(Note: HR stands for Human-Reasoning; PL, Propositional Logic; FOL, First-Order or Predicate Logic)

And remember that PL is included within FOL, i.e., $PL \subset FOL$

Whence, we could say that a computer (as von Neumann architecture describes it) is nothing but a big *function*. Also, any logic gate can be seen as a function: they take as input one or more *truth values* (True/False, 0/1, etc.) and output one or more truth values (example, decoder/encoder, an adder, a flip-flop, ... and of course and AND, NOT and OR gates and all others). All these examples are in essence just examples of functions!

This is our dots connected!

Back to our λ-calculus.

A function of two variables, say that takes x and y and outputs the sum of both $x + y$, would be denoted as

$$\lambda x.\lambda y.x + y$$

Or a function of three variables, say the temperature in each point of your room, would look like $\lambda x.\lambda y.\lambda z.$expression with x,y,z.

What can we do with a function (in λ-calculus)?

The same as in math: we **apply the function to some (input) values**.

In λ-calculus this is done and written as

$$(\lambda x.\, x + 1)\, 4$$

clearly the output of that will be 5.

As you see, we only use parentheses for delimiting the function definition -from the left of the λ symbol to the end of the expression defining the function.

Also, any input arguments simply follow the function invocation.

Some common sense rules:

- Any wff in λ-calculus[3] can potentially be a valid input value.
- If an argument is not simple (e.g. 4334) but compound, then we should probably use parenthesis to clearly delimit what input value we mean. Consider the following example: $(\lambda x.\, x+1)5*3 = 18$ but $(\lambda x.\, x+1)(5*3) = 16$

[3] Yes, that would require to precisely define what the well-formed formulas in this system are

Summary

In summary, when we say λ-calculus we *just* mean three ingredients:

1. A notation for Variables
2. A way of building functions
3. A way of applying functions

But λ Calculus doesn't have any built-in *data type* as numbers, logical values (true/false), recursion, etc. If we want this things **we need to build them ourselves**! This is nothing short of *programming those things in λ-calculus*!.

Again we must recall the Church-Turing hypothesis: Just those three ingredients allows us to describe (*program*) any computable task, the same as a Turing Machine!

This is nothing short of amazing! After all, those are just three, seemingly harmless ingredients!

***Currying*: An alternative view of a λ function**

For the case of more than one input argument we have the option to look at a function from another point of view.

We may say that

$$\lambda x.\lambda y.\, x + y$$

is a function of two input variables/arguments. Whence we would apply it as

$$(\lambda x.\, \lambda y.\, x + y)\, 5\, 6 \;=\; 11$$

But we could also take the alternative view and look at that definition as

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

If you read carefully, it says that **this is a function of just one input argument, x, that *returns a function of one argument,* y.**

The function of one argument that it returns is, of course, $\lambda y.\, a + y$. **But here x already got a specific value** a (whatever we fed it in with)!!

Let's unfold this point of view when applying it on one and two arguments:

1. $(\lambda x.\, (\lambda y.\, x + y))\, 5 \;=\; (\lambda y.\, 5 + y)$ and the output is the `add5` function! (This is a function that adds 5 to whatever we feed it with).
2. $(\lambda x.\, (\lambda y.\, x + y))\, 5\, 6 \;=\; (\lambda y.\, 5 + y)\, 6 \;=\; 11$ and the output is obtain by feeding 6 to "add5"' function.

Viewing a function of two and more variables like in 1 is called *currying*.

Modeling *currying* in two programming languages

In JS (JavaScript) we can pass functions as if they were numbers. Let's use this in order to build a *factory* for *adders*, functions that add a fixed value to any input we feed them with.

1. JavaScript:

```
function adder(n){
 return function(x){ return x+n ; } ;
}

var add2 = adder(2) ;
var add7 = adder(7) ;

var x = add2(3) ; //x will contain the value 5
var y = add7(3) ; //y will contain the value 10
```

   Note that the inner function that is returned has **no name**. This is an example of a λ **function** -in this case in JS[4].

   Also, the function `adder`, you must admit, is a curious function: it outputs not a number, a string or say a list, but *another function*![5]

2. Haskell:

```
adder n x = x + n

let add2 = adder 2
let add7 = adder 7

let x = add2 3 --same value is in previous example.
let y = add7 3 --idem.
```

   Haskell is closest to λ-calculus, as we can see. When we feed `adder` with the value 2, we have that its first argument (n) is bound to 2, but its second is still free. Whence, `add2 = adder 2` has one and only one argument free to be fed with values, and it thus makes sense to call `add2 3`, which feeds `add2` with the value 3.

[4] This is called a **closure**: we say that the function, e.g., `add2` is a closure. What is meant is that the value, or rather, a copy of that `n` that appears in the return value of `adders` has been fixed/bound to the function `add2`. When we then create the instance `add7` we are binding *a new, different copy* of `n` to `add7`. **Strictly speaking, though, a closure is not currying**: given the same original function `f`, the *type* or *signature* of a closure of it and that of a curryied version are different. However, for a function of only two variables, they both effectively coincide.

[5] A function that takes as input another function and/or that outputs another function is called a **higher-order function**. Thus, `adders` is an example of a higher-order function.

Encoding Basic Data Types in λ Calculus

Achilles and the λ Turtle

Being a curious mind, as Achilles is, he asks his wise friend the Turtle about λ-calculus. Let's listen to their little chat...

Achilles: Bro, what's all the fuzz about this λ-calculus?

Turtle: I take it you meant *fuss*, or do you have hearing or sight problems?

Achilles: ...eh...

Turtle: Never mind. As we said, *bro*, in Lambda Calculus we only have the concept of function.

Achilles: Ok.

Turtle: Yet, the Church-Turing hypothesis claims we can build anything with functions that can also be built using a TM.

Achilles: eh...ok.

Turtle: You think this is just "ok"?? Let me restate that: we could build, for instance, the functions we saw before, *including everything we type in that's not among the three ingredients* (See Summary).

Achilles: ...uhm...for a moment I thought you were suggesting that λ-calculus has no numbers nor symbols like '+' for us to use...

Turtle: That exactly what *I am* implying!

Achilles: Whaaaaat!? But how can we not have numbers in λ-calculus, and yet be able to build a function that *adds* two numbers!? How can we possibly do such a thing??

Turtle: Well, *I can*...Guess I'm just so smart...

Achilles: ...uhm...let me see...

Turtle: oh, here it comes...

Achilles: ...the previous examples were then merely for showcasing the syntax? so when you wrote a 5 it was just a placeholder for something else?

Turtle: That's right, it was just meant as a placeholder for **the number 5 I hadn't yet defined within λ-calculus**.

Achilles: eh...wait a moment! Are you saying I can write a program that, say, adds two numbers *just with λ-calculus*??

Turtle: Slow, as usual, but you seem to be getting there...

Achilles: I think I'm starting to realize the breath of the implications in that first sentence of yours.

Turtle: ... slow, but steady,... oh well.

Achilles: I see... For instance, λ-calculus doesn't come with the boolean values of truth (true/false) either,isn't?

Turtle: That's right, my friend!

Achilles: However, you claim that we could eventually build those boolean values *from scratch* within the language of λ-calculus, right?

Turtle: That's right.

Achilles: It's almost as if numbers and truth values were just functions! That's mind blowing!!

Turtle: They *actually* are!

Achilles: Woooow!

Turtle: Finally, you got it.

Indeed, λ-calculus is just a syntax game: we only have

1. **An Alphabet**: letters, dots and parentheses
2. **Expressions**: These are composed of characters from the alphabet. We use expressions to describe *patterns*.
3. **Substitution rules**: That's the gist of a function, namely, "whenver you see this, you can write that and that".

Somewhere in there we need to add some rules that specify which expressions are valid. Example: $\lambda x.\, x$ is valid, but $\lambda.xx$ is not. That is, we would need to define rules for identifying the well-formed formulas of λ-calculus.

Encoding Logical Values in λ Calculus

We are going to build the logical values True/False (*booleans*) in λ-calculus. Remember, the latter doesn't come with any built-in data type.

The trick is to consider a paradigmatic example of the use of these boolean values, e.g., an if-then expression (see below on Conditionals).

$$TRUE := \lambda x.\, \lambda y.\, x$$
$$FALSE := \lambda x.\, \lambda y.\, y$$

Let's use these two booleans (functions!) by constructing a NOT logic gate:

$$NOT := \lambda b.\, b\, FALSE\, TRUE$$

Then we have that

$$NOT\,TRUE \equiv (\lambda b.\, b\,FALSE\,TRUE)\,TRUE = TRUE\;FALSE\;TRUE =$$
$$(\lambda x.\,\lambda y.\, x)\,FALSE\,TRUE = FALSE$$
$$NOT\,FALSE \equiv (\lambda b.\, b\,FALSE\,TRUE)\,TRUE = FALSE\;FALSE\;TRUE =$$
$$(\lambda x.\,\lambda y.\, y)\,FALSE\,TRUE = TRUE$$

Amazing! We just built the NOT gate using three functions! In λ-calculus, even a boolean value turns out to be a function!!

A value that actually is a function... Mind blowing!!

Exercises

1. Define the logic gate AND
2. Define the logic gate OR

Solutions:

The guiding idea: AND TRUE TRUE = TRUE. In any other case it outputs FALSE. Analogously for the other: OR FALSE FALSE is FALSE, and TRUE in any other case.

1. $AND := \lambda x.\lambda y.\, x\,y\,FALSE$. Then we have

   AND TRUE TRUE $=$ ($\lambda x.\lambda y.\, x\,y$ FALSE) TRUE TRUE $=$ TRUE TRUE FALSE $=$ TRUE

   AND TRUE FALSE $=$ ($\lambda x.\lambda y.\, x\,y$ FALSE) TRUE FALSE $=$ TRUE FALSE FALSE $=$ FALSE

   AND FALSE TRUE $=$ ($\lambda x.\lambda y.\, x\,y$ FALSE) FALSE TRUE $=$ FALSE TRUE FALSE $=$ FALSE

   AND FALSE FALSE $=$ ($\lambda x.\lambda y.\, x\,y$ FALSE) FALSE FALSE $=$ FALSE FALSE FALSE $=$ FALSE

2. $OR := \lambda x.\lambda y.\, y\,TRUE\,x$. Then we have

   OR TRUE TRUE $=$ ($\lambda x.\lambda y.\, y$ TRUE x) TRUE TRUE $=$ TRUE TRUE TRUE $=$ TRUE

   OR TRUE FALSE $=$ ($\lambda x.\lambda y.\, y$ TRUE x) TRUE FALSE $=$ FALSE TRUE TRUE $=$ TRUE

   OR FALSE TRUE $=$ ($\lambda x.\lambda y.\, y$ TRUE x) FALSE TRUE $=$ TRUE TRUE FALSE $=$ TRUE

   OR FALSE FALSE $=$ ($\lambda x.\lambda y.\, y$ TRUE x) FALSE FALSE $=$ FALSE TRUE FALSE $=$ FALSE

As we have now the logic operators NOT, AND, and OR, we can describe all of Propositional Logic using λ-calculus! This proves then that Propositional Logic can be understood with only the concept of *function*.

This should not be surprising: we already saw that logic circuits are but a set of input values being transformed into some output values by a (black) box (of logic gates). But this is the gist of a function!

Conditional

Assuming the first argument is one of our booleans just defined (TRUE/FALSE), we can build the conditional as

$$\text{if-then-else} := \lambda x.\, \lambda y.\, \lambda z.\, x\, y\, z$$

Let's test it. In the following, A and B are arbitrary functions expressed in λ-calculus

$$\text{if-then-else}\,\text{TRUE}\,\text{A}\,\text{B} = \text{TRUE}\,\text{A}\,\text{B} = \text{A}$$

$$\text{if-then-else}\,\text{FALSE}\,\text{A}\,\text{B} = \text{FALSE}\,\text{A}\,\text{B} = \text{B}$$

This justifies the way we defined our boolean values TRUE and FALSE: they are the ones that make this if-then-else structure work.

Numbers

$$ZER0 := \lambda f.\, \lambda x.\, x$$

$$\text{SUCC} := \lambda n.\, \lambda f.\, \lambda x.\, f\,(n\, f\, x)$$

We can then keep on defining abreviations for the rest of the numbers in the following way:

$$1 := \text{ONE} := \text{SUCC ZER0} = (\lambda f.\, \lambda x.\, f\,(\text{ZER0}\, f\, x)) = \lambda f.\, \lambda x.\, f\, x$$

$$2 := \text{TWO} := \text{SUCC ONE} = (\text{SUCC}\,(\text{SUCC ZER0})) = \ldots = \lambda f.\, \lambda x.\, f\,(f\, x))$$

$$3 := \text{THREE} := \text{SUCC TWO} = (\text{SUCC}\,(\text{SUCC}\,(\text{SUCC ZER0}))) = \ldots = \lambda f.\, \lambda x.\, f\,(f\,(f\, x)))$$

and so on and so forth. Here `SUCC` stands for "successor". Thus, `SUCC 1` refers to the successor of the number `1`.

This way of constructing and thinking about the Natural numbers is due to the italian mathematician and logician Giuseppe Peano in 1889.

Notice that we just provided a way to build each number, and we did so in a recursive way. However, we have not defined yet a way to **compare numbers**, that is, so far, we do not have any way of expressing the sentence "is 2 = 3?", or "is 2<3?", and get an answer in the form of a boolean.

There are more things we cannot do yet with numbers (e.g., arithmetic! "what is 2+3?"). If we would add all the definitions we would need in order to talk about the Natural numbers in the way we are already used, we would obtain what's called the **Peano axioms of the Natural numbers**, which is still an essential building block of mathematics.

This is not the only way to define numbers in λ-calculus. Alonzo Church gave the following construction:

$0 := \lambda f.\lambda x.\, x$

$1 := \lambda f.\lambda x.\, f\, x$

$2 := \lambda f.\lambda x.\, f\,(f\, x)$

$3 := \lambda f.\lambda x.\, f\,(f\,(f\, x))$

We could write things like $0\,\mathrm{NOT}\, x = x$, or $3\,\mathrm{NOT}\, x = NOT\,(NOT\,(NOT\, x)) = NOT\, x$.

It's a common mistake to conclude from here that 3 "is equal to" 1. However, this would be misunderstanding the definition of numbers in λ-calculus: It's not the result of applying 3 to `NOT` and `x`, but it's definition as a function.

EXERCISES

Using any of the languages C/C++, JavaScript, Python or Haskell, build as functions the

1. boolean values of truth using functions.
2. logic gate NOT.
3. logic gate AND.
4. logic gate OR.
5. conditional
6. numbers

Solutions

- 1-6 In JavaScript:

```
var TRUE = function(x,y){ return x;}
var FALSE = function(x,y){ return y;}

var NOT = function(b){ return b(FALSE,TRUE) ; }

var AND = function(x,y){ return x(y,FALSE);}
var OR = function(x,y){ return y(TRUE,x);}

var if_then_else = function(x,y,z){ return x(y,z); }

var zero = function(f,x){ return x; }
var succ = function(n){ return  function(f,x){
                                    return f(n(f,x)) ;
```

```
                                        };
                                }
```

```
var one = succ( zero ) ;
var two = succ( one );
```

Go to Firefox and open the scratchpad. Paste these definitions, add at the end

```
alert(NOT(TRUE).name);

alert(one);
```

and press Run. Experiment further.

- 1-6 in Python:

```
def TRUE(x,y):
    return x
def FALSE(x,y):
    return y

def NOT(b):
    return b(FALSE,TRUE)

print( NOT(FALSE).__name__ )
```

- 1-6 in C: (Caution. The following is a contrived example in C. See comment after code)

```
typedef void *(*bol)() ;

bol true(bol x, bol y){ return x ; }
bol false(bol x, bol y){ return x ; }

bol not(bol b ){ return b(false,true) ;}



int main(){
 bol ntru = not((bol)true);
 bol nfal = not((bol)fal);

 return 0;
}
```

C does allow passing functions as arguments of functions. However, the way it does so, compared to Javascript, Python or Haskell, is more of a hack, as function are not at the same level as other objects (e.g, `int` or `char`).

Whence, this *apparent solution* in C is, however, misleading. While it may compile -it does so for me-, we get warnings from the compiler that our definition of `not` involves returning incompatible pointers.

The issue is that some platforms could have 32-bit data pointers but 64-bit function pointers, or vice versa. In practice, it's probably pretty rare, so it really depends on how portable you want it to be. But if you use gcc and specify -pedantic, you will get a warning when using void * for function pointers.

For a more detailed discussing see this question in stackoverflow.

In short, this apparent solution is actually strictly non-compliant with the language specifications. **C doesn't gives enough *expressiveness* to build such an example**. It has to do with C not allowing for building *closures*. This example, therefore, doesn't count as a valid solution.

Recursion: The Y-Combinator

In mathematics the factorial of a number, say 3, is defined as $3! = 3 \cdot 2 \cdot 1 = 6$ and $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.

Here is an implementation of the factorial function in Haskell:

```
facc 1 = 1
facc n = n * facc (n-1)

main = print $ facc 5
```

This is a recursive implementation of the factorial function, which makes it way more elegant than one in terms of a for-loop say.

But this raises a question: *how can we build recursive functions and loops using λ-calculus*? The answer is the so called **Y-combinator** function.

This is arguably the most important function in λ-calculus. It is the function that allows us to do recursion.

$$Y := \lambda f.\,(\lambda x.\,f(x\,x))\,(\lambda x.\,f(x\,x))$$

The Y-combinator was invented by Haskell Curry. In his name we have the *currying method* and the functional programming language *HASKELL*.

Haskell Curry is also the father of what is called *Combinatory Logic*. This constitutes a third model of computation which is similar to λ-calculus. A key result proved by Curry is that *any computable function can be expressed as combinations of only two basic functions, the so called **K** and **S** functions*!

Give yourself some time to let this sink in. It's nothing short of amazing!

This logic is used for building compilers for some *functional programming languages*.