# Advanced Topics in Programming Languages
# Lecture 3 - Models of Computation

Andrey Leshenko
Maxim Borin

16/11/2017

## 1 Lambda Calculus

At the beginning of the last century, several attempts were made to theoretically define computation. The most famous of them is the Turing Machine - a read/write head moving across an infinite tape, executing programs. Like Alan Turing, the American mathematician Alonzo Church was also very interested in the question "What is a computable function?". The formal system he developed during the 1930s—called Lambda Calculus—can be used to compute any computable function. Unlike the "mechanical" Turing Machine, Lambda Calculus takes a mathematical approach to computation, defining computation using higher order functions.

The usefulness of functions is not a secret in the world of programming. Functions can be used to avoid repetition and shorten expressions, for example:

$$(5 \cdot 4 \cdot 3 \cdot 2 \cdot 1) + (7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1) = fac(5) + fac(7)$$

Lambda Calculus takes this idea to the extreme, showing that functions alone can be used to compute anything. It is not meant as a practical programming language ($\lambda$-programs are usually cryptic and hard to understand), but more as a universal Assembly Language that any computation can be translated to. This makes Lambda Calculus a useful tool in programming language research.

## 2 Syntax of Lambda Expressions

Expressions in Lambda Calculus define a tree of the following elements: variables, anonymous function definitions, and applications of anonymous functions to arguments.

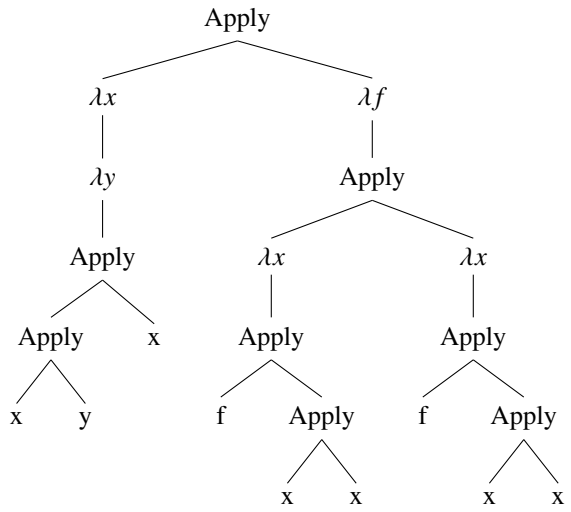
| | | | |
|---|---|---|---|
| t | ::= | | Term |
| | | x | Variable |
| | | $\lambda$x . t | Abstraction (Anonymous function) |
| | | t t | Function application |

An abstraction is similar to anonymous functions in many languages. It receives exactly one argument and returns an expression that can use this argument. Function applications are similar to what is found in other languages, except they don't use braces. Here are the same definitions in JavaScript style:

| | | | |
|---|---|---|---|
| t | ::= | | Term |
| | | x | Variable |
| | | `function(x) {return t;}` | Anonymous function |
| | | `t(t)` | Function application |

Some conventions are used to minimize the number of braces: function application associate to the left, meaning that `(a b c d)` will be interpreted as `(((a b) c) d)`. Also, the body of an abstraction (function) extends to the right as far as possible. Writing $\lambda$x. $\lambda$y. x y z is the same as writing ($\lambda$x. ($\lambda$y. x y z)). Here is a complex expression and its syntax tree:

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




Finally, some examples of $\lambda$-expressions and their meaning are in Table 1:

| $\lambda$ expression | Meaning |
|---|---|
| $\lambda$x. "1" | Always returns "1". |
| $\lambda$x. x | The identity function. |
| $\lambda$x. y | Returns the value of $y$. |
| $\lambda$x. s x | Applies the function $s$ (defined elsewhere) to its argument $x$. |
| $\lambda$x. s (s x) | Applies the function $s$ to its argument $x$ twice. |
| $\lambda$f. $\lambda$g. f g | Takes one argument $f$, returns a function which takes one argument $g$ and returns the result of applying $f$ to $g$. |

Table 1: $\lambda$-expressions.

# 3 Evaluating Expressions

Expressions are evaluated like you would expect from Python or JavaScript code: When the abstraction $(\lambda x.\ x + x \cdot 2)$ is applied to a term $t$, all the occurrences of $x$ in the body of the abstraction are replaced by $t$, giving us $t + t \cdot 2$. There are some subtleties, for example, inner abstractions can reuse variable a name used in an outer abstraction, creating a new variable and "shadowing" the outer variable. The substitution process needs to be smart, and replace only the $x$-s that were bound by the abstraction, ignoring $x$-s that represent other variables. For example applying $(\lambda x.\ x\ \lambda x.\ x)$ to $t$ will equal $(t\ \lambda x.\ x)$. The inner $x$ was not changed. These rules are not surprising, but we will still define them formally:

**Definition 1** *An occurrence of $x$ is bound in a term $t$ if it occurs in $(\lambda x.\ t)$. Otherwise it is free. $\lambda x$ is its binder. $FV : t \to 2^{Var}$ is the set of free variables of $t$.*

$$
\begin{aligned}
&FV(x) = x \\
&FV(\lambda x.t) = FV(t) - x \\
&FV(t_1 t_2) = FV(t_1 \cup f_2)
\end{aligned}
$$

**Definition 2** *Substitution:*

$$
\begin{aligned}
&[x \to s]x = s && \\
&[x \to s]y = y && \text{if } y \neq x \\
&[x \to s](\lambda y.\ t_1) = \lambda y.[x \to s]t_1 && \text{if } y \neq x \text{ and } y \notin FV(s) \\
&[x \to s](t_1\ t_2) = ([x \to s]t_1)\ ([x \to s]t_2) &&
\end{aligned}
$$

**Definition 3** *Beta Reduction:*

$$(\lambda x.\ t_1)\ t_2 \Longrightarrow_\beta [x \to t_2]t_1$$

**Definition 4** *Alpha Renaming:*

$$(\lambda x.\ t) \Longrightarrow_\alpha \lambda y.[x \to y]t \qquad \text{if } y \notin FV(t)$$

The definition of free and bound variables is self-explanatory. Substitution works recursively, changing the substituted variable and leaving everything else unchanged. Beta reduction executes function application using substitution. The Alpha renaming rule requires more explanation.

Because of "shadowing", different variables can have the same name. In case of $[x \to s](\lambda y.\ t_1)$ the substitution can only take place if $y \neq x$ and $y \notin FV(s)$. If $y = x$ then $y$ is a different variable that has the same name as $x$, and so it cannot be changed during the substitution. If $y \in FV(s)$ and we still do the substitution, then during substitution we may add variables that are called $y$, but were bound in another scope; in the new expression they will be bound by the current $\lambda y$, changing their meaning and value. In both of these cases we can't continue with the substitution because different variables share the same name.

Whenever this happens, $\beta$-reduction gets stuck, and $\alpha$-renaming must be used to continue. In $\alpha$-renaming we just rename a variable and its bound occurrences, with

the goal of resolving the conflicts. For example when $(\lambda x.\ \lambda x.\ x)\ t$ gets stuck during $\beta$-reduction, and we use $\alpha$-renaming, getting $(\lambda x.\ \lambda y.\ y)\ t$.

We now give a few examples of using what we just defined:

$$
\begin{aligned}
&FV(\lambda x.\ \lambda y.\ t) && = \{t\} \\
&FV(a\ b\ (\lambda a.\ \lambda b.\ c)) && = \{a, b, c\} \\
&FV((\lambda a.\ x\ a)\ (\lambda x.\ \lambda y.\ y\ z)) && = \{x, z\}
\end{aligned}
$$

$$
\begin{aligned}
&[x \to s]\ x\ (y\ x)\ (x\ (y\ y\ y)) && = s\ (y\ s)\ (s\ (y\ y\ y)) \\
&[x \to s]\ (\lambda y.\ x\ (\lambda z.\ y\ z\ x)) && = (\lambda y.\ s\ (\lambda z.\ y\ z\ s)) \\
&[x \to s](\lambda y.x\ (\lambda x.\ y\ z\ x)) && = \text{stuck at } \lambda x.\ y\ z\ x \\
&(\lambda y.x\ (\lambda x.\ y\ z\ x)) && \Longrightarrow_{\alpha} (\lambda y.x\ (\lambda a.\ y\ z\ a)) \\
&[x \to s](\lambda y.x\ (\lambda a.\ y\ z\ a)) && = (\lambda y.s\ (\lambda a.\ y\ z\ a))
\end{aligned}
$$

$$
\begin{aligned}
&(\lambda x.\ x\ x)\ t && \Longrightarrow_{\beta} t\ t \\
&(\lambda x.\ \lambda y.\ y\ x)(\lambda z.\ z) && \Longrightarrow_{\beta} \lambda y.\ y\ (\lambda z.\ z)
\end{aligned}
$$

# 4 Order of Evaluation

The rules of Beta reduction do not dictate a specific order for applying them; but this order can often matter. For example, let us define a $\lambda$-expression that evaluates to itself:

$$
\begin{aligned}
\Omega \equiv &(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \\
\Longrightarrow &(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)
\end{aligned}
$$

When we evaluate $(\lambda x.\ y)\ \Omega$, if we evaluate the argument to the function first, it will expand infinitely and we will never get the result (the functional equivalent to an infinite loop); but if we substitute $\Omega$ for $x$ we will see that $\Omega$ is never used, and get just $y$.

Different evaluation orders make the computation either terminate or continue indefinitely. Here is a list of the common ones:

**Full-beta-reduction** All possible orders.

**Applicative order, call by value (Eager)** Left to right; fully evaluate arguments before function.

**Normal Order** The leftmost, outermost redex is always reduced first.

**Call by Name (Lazy)** Evaluate arguments as needed.

**Call by Need** Evaluate arguments as needed and store for subsequent usages. (Implemented in Haskell)

## 4.1 Eager Evaluation (Call by Value)

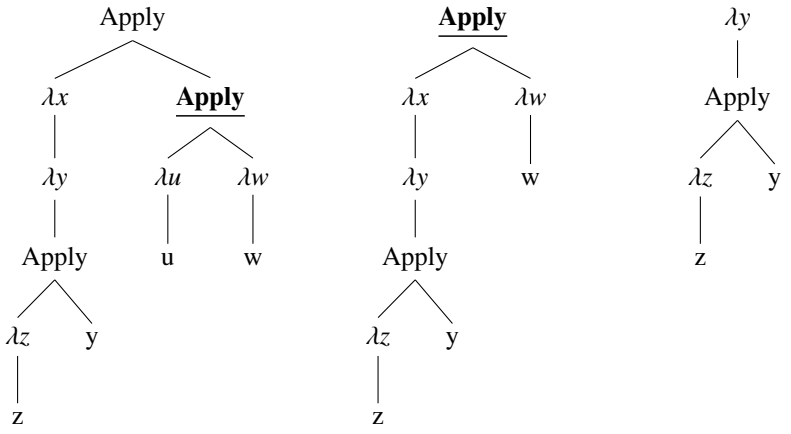
Call by Value evaluation, also known as Eager evaluation is the evaluation order used in most imperative programming languages. The arguments to functions are fully evaluated before applying the function, from left to right. The full formal rules are given below. They are given in the order of their precedence. (Apply the later rules only if the first ones cannot be applied)

$$\frac{t_1 \Longrightarrow t_1'}{t_1\ t_2 \Longrightarrow t_1'\ t_2} \qquad \text{(E-APPL1)}$$

$$\frac{t_2 \Longrightarrow t_2'}{v_1\ t_2 \Longrightarrow v_1\ t_2'} \qquad \text{(E-APPL2)}$$

$$(\lambda x.\ t_1)\ v_2 \Longrightarrow [x \to v_2]t_1 \qquad \text{(E-AppAbs)}$$

In a function application, first the function term itself is evaluated. When it becomes a $\lambda$-definition and can't be evaluated further, its argument will be evaluated. After both the function and its argument have been evaluated, the function is applied. Example:



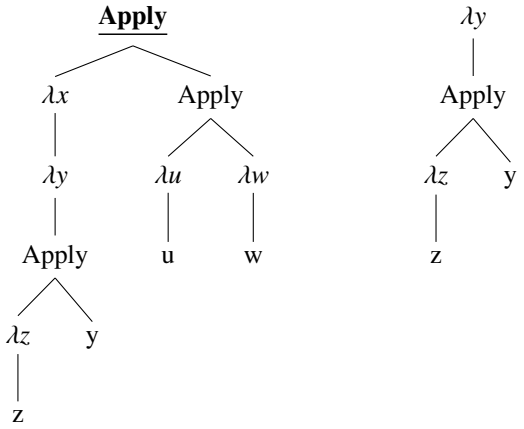

## 4.2 Lazy Evaluation (Call by Name)

In Lazy Evaluation, values are computed only when they are really needed. There are only two rules: (no need to define precedence because in any context only a single rule applies)

$$(\lambda x.\ t_1)\ v_2 \Longrightarrow [x \to v_2]t_1 \qquad \text{(E-AppAbs)}$$

$$\frac{t_1 \Longrightarrow t_1'}{t_1\ t_2 \Longrightarrow t_1'\ t_2} \qquad \text{(E-APPL1)}$$

The E-APPL2 rule that was in Eager Evaluation is now gone—we do not evaluate arguments but just pass then in as-is. Only the arguments that are used will be evaluated later.




## 4.3 Normal Order

Normal Order Evaluation is similar to Lazy Evaluation, except it includes additional evaluation steps that may reduce the result further. Here is the rules list, given in the order of precedence:

$$(\lambda x.\, t_1)\; v_2 \Longrightarrow [x \to v_2]t_1 \qquad \text{(E-AppAbs)}$$

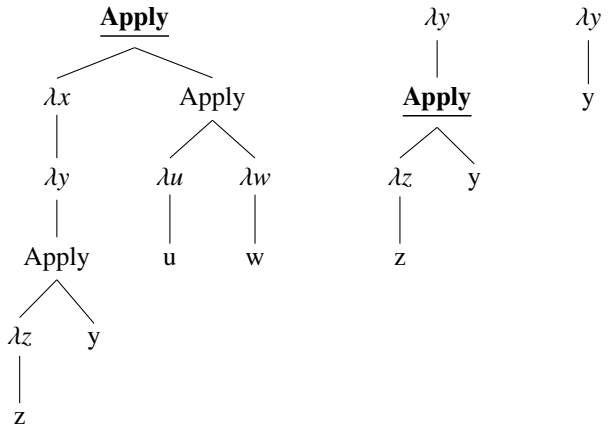
$$\frac{t_1 \Longrightarrow t_1'}{t_1\; t_2 \Longrightarrow t_1'\; t_2} \qquad \text{(E-APPL1)}$$

$$\frac{t_2 \Longrightarrow t_2'}{v_1\; t_2 \Longrightarrow v_1\; t_2'} \qquad \text{(E-APPL2)}$$

$$\frac{t \Longrightarrow t'}{\lambda x.t \Longrightarrow \lambda x.t'} \qquad \text{(E-ABS)}$$

The rules are similar to eager evaluation, except the E-Abs rule which allows us to evaluate the insides of functions, even if they are never applied.

Normal Order has mostly theoretical value. Here we can see how it reduced our example even further:



## 4.4 More About Evaluation Orders

In practice, eager evaluation is more common because it is easier to implement efficiently, and it blends more easily with side effects. Most mainstream languages like C Java and ML use it.

Lazy evaluation has nice theoretical properties, and lets you play with infinite objects. It is used in Haskell and some domain specific languages. Some forms of it can be found in languages with eager evaluation, for example generators in Python.

There are also other ways do define the evaluation orders. Lazy evaluation can be defined using relations:

$$\lambda x.\, e \to^{l} \lambda x.\, e \qquad \text{(L-Reflexive)}$$

$$\frac{e_1 \to^{l} \lambda x.\, e \quad [x \mapsto e_2]e \to^{l} e'}{e_1\ e_2 \to^{l} e'} \qquad \text{(L-Apply)}$$

Eager evaluation can also be defined in this way. Notice the subtle difference between the lazy and eager definitions:

$$\lambda x.\, e \to^{eg} \lambda x.\, e \qquad \text{(E-Reflexive)}$$

$$\frac{e_1 \to^{eg} \lambda x.\, e \quad e_2 \to^{eg} e' \quad [x \mapsto e_2]e \to^{eg} e''}{e_1\ e_2 \to^{eg} e'} \qquad \text{(E-Apply)}$$

## 4.5 The Church-Rosser Theorem

We have seen that evaluating Lambda expressions in different orders can bring us different results. But does that means that Lambda expressions don't have a defined value?

The Church-Rosser Theorem states that if there two sequences of reductions that can be applied to the same term, then there exists a term that is reachable from both results, by applying sequences of additional reductions. In other words:

$$\text{if } a \Longrightarrow^{*} b, a \Longrightarrow^{*} c$$
$$\text{then } \exists d : b \Longrightarrow^{*} d, b \Longrightarrow^{*} d$$

This theorem shows that expressions in Lambda Calculus do have defined values, and that while different evaluations orders can bring us to different states, these different expressions can be evaluated to converge again.

# 5 Translating Concepts to Lambda Calculus

We said before that Lambda Calculus can be used to compute anything. We will now look at a few examples of how different programming concepts can be translated into $\lambda$-expressions.

## 5.1 Multiple Argument Functions - Currying

Functions in $\lambda$-calculus receive exactly one argument, but this limitation can be overcome using a technique called currying. A function with two arguments can be thought of as a function which receives the first argument and returns another function. The second function receives the second argument and returns the result. Here is an example of a three argument function implemented using currying:

$$
\begin{aligned}
&(\lambda x.(\lambda y.(\lambda z.x + y + z))) \; 5 \; 6 \; 7 \\
&= (\lambda y.(\lambda z.5 + y + z)) \; 6 \; 7 \\
&= (\lambda z.5 + 6 + z) \; 7 \\
&= 5 + 6 + 7
\end{aligned}
$$

Because function application associates to the left, curried function can be applied to their arguments without adding braces.

## 5.2 Name Binding

The syntax of Lambda Calculus does not support giving names to expressions. We would like to be able to do something like let $a = t_1$ in $t_2$. For example let $a = (\lambda x.\ x)$ in $(a\ a\ a)$. This limitation can be overcome by writing a function that uses the variable, and applying it to the wanted value:

$$
\begin{aligned}
&(\lambda a.\ t_2)\ t_1 \\
&(\lambda a.\ a\ a\ a)\ (\lambda x.x)
\end{aligned}
$$

## 5.3 Church Booleans

For a while now we assumed that we could write $\lambda$-expressions that manipulate numbers. But the syntax of Lambda Calculus never defines numbers or any other data types except functions. This limitation can be overcome by defining numbers and other data types such as booleans to be agreed upon functions. These values can then be manipulated by applying and defining functions, and testing their values.

We will now demonstrate how booleans can be implemented using a technique called Church Booleans:

$$
\begin{aligned}
\text{true} &\equiv \lambda a.\lambda b.a \\
\text{false} &\equiv \lambda a.\lambda b.b
\end{aligned}
$$

Both True and False functions receive two arguments. True returns the first, while False returns the second. We can now define useful functions that can operate on these booleans:

$$
\begin{aligned}
\text{test} &\equiv \lambda b.\lambda m.\lambda n.\ b\ m\ n \\
\text{and} &\equiv \lambda b_1.\lambda b_2.\ b_1\ b_2\ False \\
\text{or} &\equiv \lambda b_1.\lambda c_2.\ b_1\ True\ b_2 \\
\text{not} &\equiv \lambda b.\lambda x.\lambda y.\ b\ y\ x \equiv \lambda b.\ b\ False\ True
\end{aligned}
$$

The "test" function works like the ternary operator in other languages:

$$
\begin{aligned}
\text{test True}\ x\ y &= \text{test}\ (\lambda a.\lambda b.a)\ x\ y \\
&= (\lambda b.\lambda m.\lambda n.\ b\ m\ n)\ (\lambda a.\lambda b.a)\ x\ y \\
&= (\lambda m.\lambda n.\ (\lambda a.\lambda b.a)\ m\ n)\ x\ y \\
&= (\lambda a.\lambda b.a)\ x\ y \\
&= (\lambda b.x)\ y \\
&= x \\
\text{test False}\ x\ y &= \text{test}\ (\lambda a.\lambda b.b)\ x\ y \\
&= (\lambda b.\lambda m.\lambda n.\ b\ m\ n)\ (\lambda a.\lambda b.b)\ x\ y \\
&= (\lambda m.\lambda n.\ (\lambda a.\lambda b.b)\ m\ n)\ x\ y \\
&= (\lambda a.\lambda b.b)\ x\ y \\
&= (\lambda b.b)\ y \\
&= y
\end{aligned}
$$

## 5.4 Church Numerals

Unlike booleans, there are infinitely many natural numbers. But we can still represent them in $\lambda$-calculus using a technique called Church Numerals. Each number will be defined as a specific function. We can execute mathematical operations by manipulating these function, and given a number we can test its properties.

| | |
|---|---|
| $c_0 = \lambda s.\lambda z.\ z$ | "zero" numeral |
| $c_1 = \lambda s.\lambda z.\ s\ z$ | "one" numeral |
| $c_2 = \lambda s.\lambda z.\ s\ (s\ z)$ | ... |
| $c_3 = \lambda s.\lambda z.\ s\ (s\ (s\ z))$ | |
| $c_n = \lambda s.\lambda z.\ s^n\ z$ | |

The number $n$ is function which applies its first argument $n$ times to its second argument. You can think that the $z$ argument is the zero number, and $s$ is a successor function that increments its input, but beware: the definition of numbers does not care what the two arguments are. The number is stored in the *structure* of the function, before it is applied to anything. We can define our own successor, addition and multiplication functions:

$$
\begin{aligned}
\text{succ} &\equiv \lambda n.\lambda s.\lambda z.\ s\ (n\ s\ z) \\
\text{plus} &\equiv \lambda m.\lambda n.\lambda s.\lambda z.\ m\ s\ (n\ s\ z) \\
\text{mult} &\equiv \lambda m.\lambda n.\ m\ (\text{plus}\ \ n)\ c_0 \\
\text{iszero} &\equiv \lambda n.n\ (\lambda x.\text{False})\ \text{True}
\end{aligned}
$$

The "iszero" operator requires more explanation. We use the argument $n$ to apply a function $n$ times to the value true. That function we apply always returns false, meaning the result will be true only the function was applied zero times. (The number was zero)

Church numerals show us that functions can be very powerful at representing information. We will now look at a more algorithmic challenge.

## 5.5 Recursive Functions - Combinators

Recursion is very important in functional languages. It is not straightforward to get it in $\lambda$-calculus because we need to allow functions to refer to themselves, even before they are defined. That trick that allows us to do this makes use of something called *combinators*.

A combinator is just a function with no free variables. For example $(\lambda x.\ x)$ and $(\lambda x.\ \lambda y.\ (x\ y))$ are combinators, while $(\lambda x.\ \lambda y.\ (x\ z))$ is not (because $z$ is free). Combinators can serve as modular building blocks for more complex expressions. We have already seen this in Church numerals and booleans, where different combinators described different values.

To get recursion we will use specific combinators, called the Y and Z combinators. These combinators allow a function to receive itself as an argument.

We have seen the $\Omega$-combinator before. It always evaluates to itself:

$$
\begin{aligned}
\Omega \equiv &(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \\
&\Longrightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)
\end{aligned}
$$

The Y-combinator looks a lot like $\Omega$. Here its definition and an example usage:

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

$$
\begin{aligned}
Y\ g\ v &= (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ g\ v \\
&\Longrightarrow ((\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x)))\ v \\
&\Longrightarrow g\ ((\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x)))\ v \\
&\Longrightarrow g\ (Y\ g)\ v
\end{aligned}
$$

We can see that when $(Y\ g)$ is evaluated, the result is $g$ applied to the initial expression. This can be used to implement recursion: a recursive function will expect its first argument to be a recursive version of itself. This recursive version can be applied to other arguments as needed.

Here is an example of defining a factorial function using Y-combinators:

$$g = \lambda f.\ \lambda n.\ \text{iszero}\ n\ 1\ (n \cdot (f\ (n-1)))$$

$$\begin{aligned}
(Y\ g)\ 3 &\Longrightarrow g\ (Y\ g)\ 3 \\
&\Longrightarrow 3 \cdot (Y\ g)\ 2 \\
&\Longrightarrow 3 \cdot g\ (Y\ g)\ 2 \\
&\Longrightarrow 3 \cdot 2 \cdot (Y\ g)\ 1 \\
&\Longrightarrow 3 \cdot 2 \cdot g\ (Y\ g)\ 1 \\
&\Longrightarrow 3 \cdot 2 \cdot 1 \cdot (Y\ g)\ 0 \\
&\Longrightarrow 3 \cdot 2 \cdot 1 \cdot g\ (Y\ g)\ 0 \\
&\Longrightarrow 3 \cdot 2 \cdot 1 \cdot 1
\end{aligned}$$

The Y-combinator allowed us to write recursive functions, but it has a flaw. The expression $(Y\ g)$ will expand infinitively under eager evaluation. The Z-combinator removes this limitation. To understand how, let's look at a similar problem, written in Python:

```
def factorize(n):
    # Very expensive computation

def g(x):
    if askUser("Hey kid, wanna see some factors?"):
        print x

def main():
    g(factorize(233108530344407544527637656...))
```

We have a very long computation that may or may not be used. The way the code is written right now, the computation will be performed anyway. What is the solution for this in the work of software? The answer is "Wrap it in a function"! This way, if someone needs the result, he can call the function and get it computed, but otherwise the calculation is avoided. Here is the corrected code:

```
def factorize(n):
    # Very expensive computation

def g(x):
    if askUser("Hey kid, wanna see some factors?"):
        print x()

def main():
    g((lambda n: factorize(233108530344407544527637656...)))
```

This idea that is used to create the Z-combinator: it is like the Y-combinator but wrapped in a function.

$$Z = (\lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y)))$$

$$\begin{aligned} Z\ g\ v &= (\lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y)))\ g\ v \\ &\Longrightarrow ((\lambda x.\ g\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ g\ (\lambda y.\ x\ x\ y)))\ v \\ &\Longrightarrow g\ (\lambda y.\ (\lambda x.\ g\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ g\ (\lambda y.\ x\ x\ y))\ y)\ v \\ &\Longrightarrow g\ (\lambda y.\ (Z\ g)\ y)\ v \end{aligned}$$

The Z-combinator behaved almost exactly like the Y-combinator, but after a single expansion it wrapped itself in a function. The next expansion will occur only if this function is used and applied to something.

## 5.6 The Church-Turing Thesis

Now that we have seen many programming concepts—from numerals to recursion—implemented in Lambda Calculus, it should be easier to believe that Lambda Calculus really can be used to compute anything.

Church and Turing proved that the class of functions computable by a Turing Machine coincides with the class of $\lambda$-computable functions (and also another class called *general recursive functions*). The Church–Turing thesis states that these formally defined classes coincide with the *informal* definition of a computable function.

## 5.7 Normal Form and Halting Problem

A term is said to be in normal form if it cannot be reduced any further in normal order semantics. Under normal order every term is either reduced to normal form or reduced infinitely. The halting problem for Lambda expressions is to decide which is the case for a given term.

Just like the Halting problem for Turing Machines, this problem is undecidable.

# 6 Continuations

## 6.1 Introduction

Continuation is a term used to represent the "current state" of a computation or the "remaining steps" required for finishing it. Depending on the way a function is implemented, the continuation of the said function may be easy or difficult to define effectively. When we talk about programming with continuations, we mean that we want to implement our functions in such a way that it will be easy to represent their continuation. Here is an example of a simple function:

```
int pow(int x, int n)
{
  if (n == 0)
    return 1;

  return x * pow(x,n-1);
}
```

The function receives the parameters $x$ and $n$ and recursively calculates $x^n$. The main call **pow(x, n)** waits for the evaluation of **pow(x,n-1)**, then uses the result to finish computation by returning **x * pow(x,n-1)**. Until **pow(x,n-1)** is evaluated, the call **pow(x,n)** is stuck waiting.

In this example, the continuation of the call **pow(x,n)** has two parts. The first part is "wait for the evaluation of pow(x,n-1)" and the second part is "return x * pow(x,n-1)". In order to represent the remaining computation steps, we need to remember both the state of the call **pow(x,n)** and the states of all subsequent recursive calls. The continuation of this function is difficult to represent. Now, let us look at an implementation where the continuation can be represented effectively and compactly:

```
int pow(int x, int n)
{
  return powCont(x, n, 1);
}

int powCont(int x, int n, int acc)
{
  if (n == 0)
    return acc;

  return powCont(x, n-1, acc * x);
}
```

This function also recursively calculates $x^n$. However, instead of finishing the work on its own, the call **pow(x,n)** pushes the rest of the work on **powCont(x,n,1)**. In this case, we can think of the call **powCont(x,n,1)** as the whole continuation of **pow(x,n)**, because the rest of the computation steps are entirely defined in **powCont(x,n,1)**.

Similarly, **powCont(x,n,1)** also doesn't compute the final result by itself. Tt pushes the rest of the work on **powCont(x,n-1,x)**, thus, **powCont(x,n-1,x)** is a full representation of the continuation for **powCont(x,n,1)**. We get the following recursion, where each call can be used to represent the whole continuation of the call that precedes it:

$$\text{pow}(x,n) = \text{powCont}(x,n,1) = \text{powCont}(x,n-1,x) = \text{powCont}(x,n-2,x^2) = \\ = \ldots = \text{powCont}(x,0,x^n) = x^n$$

Each recursion call pushes the rest of the work to its continuation and considers its job done. **powCont(x,n,1)** doesn't need to wait for the evaluation of **powCont(x,n-1,x)**, because **powCont(x,n-1,x)** has all the tools it needs to compute and return the final result. The parameter *acc* accumulates the "result so far" and is passed to the continuation in each step. The final recursive call will simply return the value of *acc*.

Function which are written to effectively use continuation are sometimes called *continuation functions*.

## 6.2 Advantages and Disadvantages

Often times, writing a continuation function is less intuitive and requires more code. But in exchange, a continuation function gives us more control over the execution of the computation as we will see later. For example, it allows us to better optimize the execution of simultaneous computations across multiple threads.

In programming, we generally take advantage of continuation functions by catching them at a certain computation step. We can store the current state of the computation by writing the information of the next continuation to memory. Then, we can put the computation aside and jump to execute a different part of the code. Later, we can jump back and continue the computation from where we left it, by using the continuation we stored in memory.

For example, we may begin our runtime execution by computing **pow(3,4)**. First we call **powCont(3,4,1)**, then we call **powCont(3,3,3)**. However, after these two steps, we may want to put this computation on hold and jump to execute a different part of the code. We'll only need to store the representation of the next continuation (**powCont(3,2,9)**) in our memory as the "current state" of the whole computation. Some time later, once we decide to resume the computation, we'll simply continue from **powCont(3,2,9)**.

Due to the way continuations are designed to be jumped from and back to, they're considered a functional expression of the "goto" statement. Some even call it "goto with arguments". As such, the caveats that apply to using goto statements often apply to using continuations as well.

## 6.3 Memory Optimization

Continuation is a common way to optimize memory allocation when using recursive functions.

In Lecture 2, we saw two different implementations for the factorial function in Haskell. The first implementation was straightforward recursion:

```
fac1 n = if n = 0 then 1 else n * fac n -1
```

In C, it's equivalent to the following implementation:

```
int fac1(int n)
{
  if (n == 0)
    return 1;
```

```
  return n * fac1(n-1);
}
```

The second implementation made better use of continuation by adding an accumulator parameter *acc*:

```
fac2 n =
  let aux n acc = if n = 1 then acc else aux n-1 n * acc
in
  aux n 1
```

In C, it's roughly equivalent to:

```
int fac2(int n)
{
  return aux(n, 1);
}

int aux(int n, int acc)
{
  if (n == 1)
    return acc;

  return aux(n-1, n * acc);
}
```

Even though the second implementation requires more code and may be more difficult to read, it has memory optimization advantages in runtime. In this particular example, the concept of continuation also happens to overlap with the term *tail recursion*.

In normal recursion, each recursion call needs memory allocation in order to remember its current state. We can only free the memory of a recursion call once all computation is done for that call, which normally won't happen until all computation is done for all subsequent recursive calls.

For example, in the first factorial implementation, when we call **fac1(3)** we allocate memory for this call. Then, we proceed to **fac1(2)** and allocate memory for it as well. We can't free the memory previously allocated for **fac1(3)**, because **fac1(3)** must wait for **fac1(2)** to be evaluated. In this manner, a call to **fac1(3)** ends up allocating "call state" memory for **fac1(3)**, **fac1(2)** **fac1(1)** and **fac1(0)**. This means that fac1 requires $O(n)$ memory,

In tail recursion, each recursive call pushes the rest of the remaining computation work on the next recursive call. As soon as we invoke the next recursive call, we can free all the memory of the current call.

For example, in the second factorial implementation, when we call **fac2(3)** we allocate memory for this call. Once **fac2(3)** calls **aux(3,1)** and allocates memory for it, we no longer need to remember anything about **fac2(3)**. We can trust **aux(3,1)** to finish the rest of the computation and to return the final value. Hence, we can free the memory allocated for **fac2(3)**. Similarly, once **aux(3,1)** calls **aux(2,3)**, we can free memory for **aux(3,1)**. This means that fac2 requires $O(1)$ memory.

## 6.4 Continuation Passing Style (CPS)

Some languages, such as Haskell, support the Continuation Passing Style (CPS for short). It's a style of implementation in which we pass a function as an argument during calls. Said function will represent the next steps required for the computation. In other words, the function stores a partial representation of the continuation.

To demonstrate the concept, here is another implementation for factorial in Haskell:

```
fac3 (n, k) = if n = 0 then k(1) else fac3(n-1, λx. k (n*x))
```

If we call fac3(n, $\lambda$x. x), it will calculate ($n!$) as we wanted. Here's an example of how computation will work for fac3(3, $\lambda$x. x):

$$\begin{aligned}
\text{fac } (3, \lambda x.x) &= \text{fac } (2, \lambda y.\ (\lambda x.x)\ (3 \cdot y)) \\
&= \text{fac } (1, \lambda z.(\lambda y.\ (\lambda x.x)\ (3 \cdot y))\ (2 \cdot z)) \\
&= \text{fac } (0, \lambda w.(\lambda z.(\lambda y.\ (\lambda x.x)\ (3 \cdot y))\ (2 \cdot z)\ (1 \cdot w)) \\
&= (\lambda w.(\lambda z.(\lambda y.\ (\lambda x.x)\ (3 \cdot y))\ (2 \cdot z)\ (1 \cdot w)))\ 1 \\
&= (\lambda z.(\lambda y.\ (\lambda x.x)\ (3 \cdot y))\ (2 \cdot z)\ (1 \cdot 1)) \\
&= (\lambda y.\ (\lambda x.x)\ (3 \cdot y))\ (2 \cdot 1) \\
&= (\lambda x.x)\ (3 \cdot 2) \\
&= (3 \cdot 2) = 6
\end{aligned}$$

Here $k$ was the "Call this function with your result when you finish" function. Each step of the factorial wants to get the result of the next steps $x$, multiply it by $n$ to get the value at the current step, and then pass it to the $k$ left by the previous steps, hence we pass $(\lambda x.\ k\ (n \cdot x))$ to the next steps.

While this was a lot of work without much visible benefit, later it will allow us to solve complicated problems elegantly.

## 6.5 Error Handling

Continuation is sometimes used in error handling. A common technique involves calling a validator function which checks whether there's an error in the current computation step. The validator receives two continuation functions as arguments, normal_continue() and error_continue(), and decides with which of them to continue depending on the result of the error validation.

For example, let's say we have function $f(x) = x \cdot 1/x$. As we know, $1/x$ is a problematic expression in case $x = 0$. We'd like to add special error handling for this scenario, using continuation techniques. Here's an example for such an implementation:

```
fun f(x) let
  val before = x
  fun normal_continue(quote) = before + quote
  fun error_continue() = 2147483647
  in
    divide(1.0, x, normal_continue, error_continue)
  end
```

Once $f(x)$ is called, we define two possible continuation functions for it. normal_continue(quote) will be invoked in case there's no error and will calculate $f(x)$ as intended. We make sure "before = x" and "quote = 1/x", thus "before + quote = x + 1/x".

The other function we define is error_continue(), which will be invoked in case there's an error. This method returns the value 2147483647, which we'll think of as "infinite".

Finally, the actual continuation of $f(x)$ is a call to the function divide(), in which we'll implement our error handling. divide() receives the two continuation functions descried above and will decide with which of them to proceed. Here's the implementation of divide():

```
fun divide (numerator, denominator, normal_continue, error_continue) =
  if denominator > 0.0001
    then normal_continue(numerator/denominator)
    else error_continue()
```

The function checks whether the denominator, which is $x$ in our example, is greater than 0.0001. If true, then it'll proceed with the normal_continue() function it received an argument, with numerator/denominator = 1/x in our case. In our example, it'll calculate $f(x) = x \cdot 1/x$.

In case denominator $\leq 0.0001$, the function treats it as a "$x = 0$" error and proceeds with error_continue(). In our example, it'll return 2147483647.

That was again a lot of work, and not much visible benefit. But the next example shows why the extra effort of writing functions in CPS style is sometimes well worth it.

## 6.6 Web Programming Example

We will now present an example that showcases how continuation can be used to enhance the responsiveness of a webpage. When a long computation is being performed on the web page, the page may become unresponsive for the user and look as if it's "frozen". One way to prevent this scenario is by using smart Continuation Style Passing

Here's a basic JavaScript implementation for a function which goes over all the elements in a document and capitalizes their text:

```
function traverseDocument(node, func) {
  func(node);
  var children = node.childNodes;
  for (var i = 0; i < children.length; i++)
    traverseDocument(children[i], func);
}

function capitaliseText(node) {
  if (node.nodeType == 3) // A text node
    node.nodeValue = node.nodeValue.toUpperCase();
}

traverseDocument(document.body, capitaliseText);
```

The function traverseDocument() recursively goes over all elements in the document when it receives node = document.body as its first input. It also receives the function capitaliseText() as an argument and applies it on every node it passes through. capitaliseText() simply changes the text in a node to upper case.

As it stands, this implementation doesn't use continuation programming. Here's how we'd change traverseDocument() into Continuation Passing Style:

```
function traverseDocument(node, func, c) {
  var children = node.childNodes;
  function handleChildren(i, c) {
    if (i < children.length)
      traverseDocument(children[i], func,
        function(){handleChildren(i + 1, c);});
    else
      c();
  }
  return func(node, function(){handleChildren(0, c);});
}

function capitaliseText(node, c) {
  if (node.nodeType == 3)
    node.nodeValue = node.nodeValue.toUpperCase();
  c();
}

traverseDocument(document.body, capitaliseText, function(){});
```

In this implementation, traverseDocument() won't call capitaliseText() right away. It first defines a continuation function handleChildren(). Then, traverseDocument() makes a full return with a call to capitaliseText() and passes it the continuation function. In other words, with this return statement, traverseDocument() passes the control over the computation to capitaliseText(), unlike in the first implementation where capitaliseText() was merely used as a subroutine of traverseDocument().

Just like before, capitaliseText() updates the node it receives the uppercase. Then, it'll invoke the continuation function it received. In our example, it'll invoke the handleChildren() which was defined in the specific traverseDocument() that invoked this call of capitaliseText(). Then, in turn, handleChildren() will make sure to invoke traverseDocument() again as needed by passing it a new node.

As it stands, this implementation is just more difficult to understand compared to the first. It doesn't make any sophisticated use of the concept of continuation just yet. Now we get to the magic part. Let's update the implementation of capitaliseText() to process only part of the elements at a time, and continue capitalizing the rest after a timeout:

```
var nodeCounter = 0;

function capitaliseText(node, c) {
  if (node.nodeType == 3)
    node.nodeValue = node.nodeValue.toUpperCase();

  nodeCounter++;
  if (nodeCounter % 20 == 0)
    setTimeout(c, 100);
  else
    c();
}
```

We added a global variable nodeCounter which will count how many nodes we have applied capitaliseText() on so far. Each time capitaliseText() finishes turning a node into upper case, it increments nodeCounter. In case we're dealing with a huge document with many nodes, capitalizing the whole page may take a long time. Doing so will "freeze" the page for the user for a while, which is something we'd like to prevent. Therefore, this new capitaliseText() "takes a break" after every 20 capitalization. Its uses the new nodeCounter variable in order to detect when it has finished 20 capitalizations since the previous "break time".

When it's time to "take a break", capitaliseText() continues with a call to setTimeout() instead of continuing as usual with its continuation function c(). setTimeout() receives the continuation function c() and will be the one responsible for invoking it in order to continue the computation.

What we did here would be very complicated to do if the code was written in a different style. Because computations were represented using continuations, the capitalization function could schedule them to execute later.

This example demonstrated the strength of continuations. Continuations are available for use in many languages (in different forms), and often be used to provide elegant solutions to difficult high-level problems.

# References


[1] Advanced Topics in Programming Languages, *Lecture 3 - Functional Programming*, 2017.

[2] Advanced Topics in Programming Languages, *Recitation - Lambda Calculus*, 2017.

[3] Wikipedia, *Lambda Calculus*, (Online)

[4] Wikipedia, *Church–Rosser Theorem*, (Online)

[5] John Quigley, *Computational Continuations*, Chicago Linux, 2007.

[6] Marijn Haverbeke, *Continuation-Passing Style*, marijnhaverbeke.nl, 2007. (Online)

[7] Wikipedia, *Continuations*, (Online)