

Handout 10

Summary of this handout: History Mathematical Notation — Prefix Notation — Lambda Expressions — Abstractions — Applications — Abstract Syntax Trees — α -Renaming — β -Reductions — Church Rosser

VI. The Lambda Calculus

So far we were concerned with models of computation based on machine models. In particular, we used the Turing machine model to define the boundary between computability and non-computability, feasibility and intractability. Machine models are often rather contrived and when using them one often has to think in a formalism and work with algorithms very dissimilar to standard programming languages. Since there are many computational models around that equally expressive as Turing machines, we will now look at one alternative model, which differs in two important points: Firstly, it is more mathematical and does not depend on the notion of a machine. Secondly, it is more closely related to the algorithmic thinking we employ when designing programs. Let's first have a look at some mathematical motivation.

87. A Brief History of Mathematical Notation

Our notation for **numbers** was introduced in the Western World in the Renaissance (around 1200) by people like Fibonacci. It is characterised by a small fixed set of digits, whose value varies with their position in a number. This place-value system was adopted from the Arabs who themselves credit the Indians. We don't know when and where in India it was invented.

A notation for **expressions** and **equations** was not available until the 17th century, when Francois Viète started to make systematic use of placeholders for parameters and abbreviations for the arithmetic operations. Until then, a simple expression such as $3x^2$ had to be described by spelling out the actual computations which are necessary to obtain $3x^2$ from a value for x .

It took another 250 years before Alonzo Church developed a notation for arbitrary **functions**. His notation is called λ -calculus ("lambda calculus"). Church introduced his formalism to give a functional foundation for Mathematics but in the end mathematicians preferred (axiomatic) set theory. The λ -calculus was re-discovered as a versatile tool in Computer Science by people like McCarthy, Strachey, Landin, and Scott in the 1960s.

Incidentally, the history of programming languages mirrors that of mathematical notation, albeit in a time-condensed fashion: In the early days (1936-1950), computer engineers struggled with number representation and tried many different schemes, before the modern standard of 2-complement for integers and floating point for reals was generally adopted. Viète's notation for expressions was the main innovation in FORTRAN, the world's first high-level programming language (Backus 1953), thus liberating the programmer from writing out tedious sequences of assembly instructions. Not too long after this, 1960, McCarthy came out with his list processing language Lisp. McCarthy knew of the λ -calculus, and his language closely resembles it.

Today, not many languages offer the powerful descriptive facilities of the λ -calculus, in particular, the mainstream languages Java and C++ make a strict distinction between primitive data types and objects, on the one hand, and functions (= methods), on the other hand. However, the line of development started with Lisp, has it led to some truly remarkable languages such as ML and Haskell. OCaml, a dialect of ML, that incorporates object oriented features, performs on many standard benchmarks better than C++.

VI.1 The Simple Lambda Calculus

The λ -calculus is a notation for functions. It is extremely economical but at first sight perhaps somewhat cryptic, which stems from its origins in mathematical logic. Expressions in the λ -calculus are written in strict *prefix* form, that is, there are no infix or postfix operators (such as $+$, $-$, etc.). Furthermore, function

and argument are simply written next to each other, without brackets around the argument. So where the mathematician and the computer programmer would write “ $\sin(x)$ ”, in the λ -calculus we simply write “ $\sin x$ ”. If a function takes more than one argument, then these are simply lined up after the function. Thus “ $x + 3$ ” becomes “ $+ x 3$ ”, and “ x^2 ” becomes “ $* x x$ ”. Brackets are employed only to enforce a special grouping. For example, where we would normally write “ $\sin(x) + 4$ ”, the λ -calculus formulation is “ $+ (\sin x) 4$ ”.

88. Functions in the λ -calculus

If an expression contains a variable — say, x — then one can form the function by considering the relationship between concrete values for x and the resulting value of the expression. In mathematics, function formation is sometimes written as an equation, $f(x) = 3x$, sometimes as a mapping $x \mapsto 3x$. In the λ -calculus a special notation is available which dispenses with the need to give a name to the function (as in $f(x) = 3x$) and which easily scales up to more complicated function definitions. In the given example we would re-write the expression “ $3x$ ” into “ $* 3 x$ ” and then turn it into a function by preceding it with “ $\lambda x.$ ”. We get: “ $\lambda x. * 3 x$ ”. The Greek letter λ (“lambda”) has a role similar to the keyword “function” in some programming languages. It alerts the reader that the variable which follows is not part of an expression but the *formal parameter* of the function declaration. We say that we *abstract over* the variable x and call $\lambda x.$ the *binder* and x the *bound variable*. The dot after the formal parameter introduces the function body also called the *scope* of the λ abstraction. The scope extends to the right side of the dot until it is ended by a closing bracket.

Let’s look more closely at the similarity with programming languages, for example in Java:

```
public static int f(int x ) { return( 3 * x ); }
      |                |   |                |
       $\lambda$            x   .                * 3 x
```

In a functional programming language the same definition is generally shorter. For example in Lisp or Scheme (a Lisp dialect):

```
(lambda (x) (* 3 x))
```

89. Application of λ -functions

In mathematics the application of a function is often denoted by inserting the concrete value for the variable in question. So if $f(x) = 3x$ then $f(4) = 3 \cdot 4$. This notation, however, implies that we know what the function name f actually stands for when we read $f(4)$. If the function is written in λ -notation it can itself be used in an expression. For example, the application of the function from above to the value 4 is written as $(\lambda x. * 3 x) 4$. Thus, application is simply juxtaposition. Observe that the brackets around the function are crucial, to make clear where the definition of the function ends. If we wrote $\lambda x. * 3 x 4$ then 4 would become part of the function body and we would get the function which assigns to x the value $3 * x * 4$ (assuming that $*$ is interpreted as a 3-ary function; otherwise the λ -term is nonsensical, see below). So again, brackets are used for delineating parts of a λ -term, they do not have an intrinsic meaning of their own. We will see below how we actually compute the value of the function application.

90. Reintroducing Names

Functions in the λ -calculus are nameless. When we want to reuse them, we have to restate them, which can become quite lengthy. Thus, although it is not strictly necessary, it will be convenient to introduce abbreviations for λ -terms. We name functions using a special equality symbol. So if we abbreviate our function term to F :

$$F \stackrel{\text{def}}{=} \lambda x. * 3 x$$

then we can write $F 4$ instead of $(\lambda x. * 3 x) 4$.

With this our description of the λ -calculus as a notational device is almost complete; there is just one more case to consider. Suppose the body of a function consists of another function, as here

$$N \stackrel{\text{def}}{=} \lambda y. (\lambda x. * y x)$$

If we apply this function to the value 3 then we get back $\lambda x.* 3 x$, in other words, N is a function, which when applied to a number, returns another function (i.e., $N 3$ behaves like F). However, we could also consider it as a function of *two* arguments, where we get a number back if we supply N with *two* numerical arguments ($N 3 4$ should evaluate to 12). Both views are legitimate and perfectly consistent with each other. If we want to stress the first interpretation we may write the term with brackets as above, if we want to see it as a function of two arguments then we can leave out the brackets:

$$\lambda y.\lambda x.* y x$$

or, as we will lazily do sometimes, even omit the second lambda:

$$\lambda y x.* y x$$

but note that this is really just an abbreviation of the official term.

Likewise, in the application of N to arguments 3 and 4 we can use brackets to stress that 3 is to be used first: $(N 3) 4$ or we can suggest simultaneous application: $N 3 4$. Whatever our intuition about N , the result will be the same (namely, 12).

91. Grammar of the λ -calculus

Function formation and function application are all that there is. They can be mixed freely and used as often as desired or needed, which is another way of saying that λ -terms are constructed according to the grammar

$$M ::= c \mid x \mid M M \mid \lambda x.M$$

Here the placeholder c represents any *constant* we might wish to use in a λ -term, such as numbers 1, 2, 3,... or arithmetic operators $+$, $*$, etc. Sometime constants are also called global variables, since they might be bound by a λ binder somewhere outside the term in question. A term without constants is called *pure*. Similarly, the letter x represents any of infinitely many possible variables.

The grammar is ambiguous; the term $\lambda x.x y$ could be parsed as

1. $\lambda x.(x y)$, or
2. $(\lambda x.x) y$,

respectively. However we want to allow only the first interpretation to be possible. Using additional non-terminals and productions the conventional interpretation can be enforced:

$$\begin{aligned} \langle \text{term} \rangle & ::= \langle \text{atom} \rangle \mid \langle \text{app} \rangle \mid \langle \text{fun} \rangle \\ \langle \text{atom} \rangle & ::= \langle \text{head-atom} \rangle \mid (\langle \text{app} \rangle) \\ \langle \text{head-atom} \rangle & ::= x \mid c \mid (\langle \text{fun} \rangle) \\ \langle \text{app} \rangle & ::= \langle \text{head-atom} \rangle \langle \text{atom} \rangle \mid \langle \text{app} \rangle \langle \text{atom} \rangle \\ \langle \text{fun} \rangle & ::= \lambda x. \langle \text{term} \rangle \end{aligned}$$

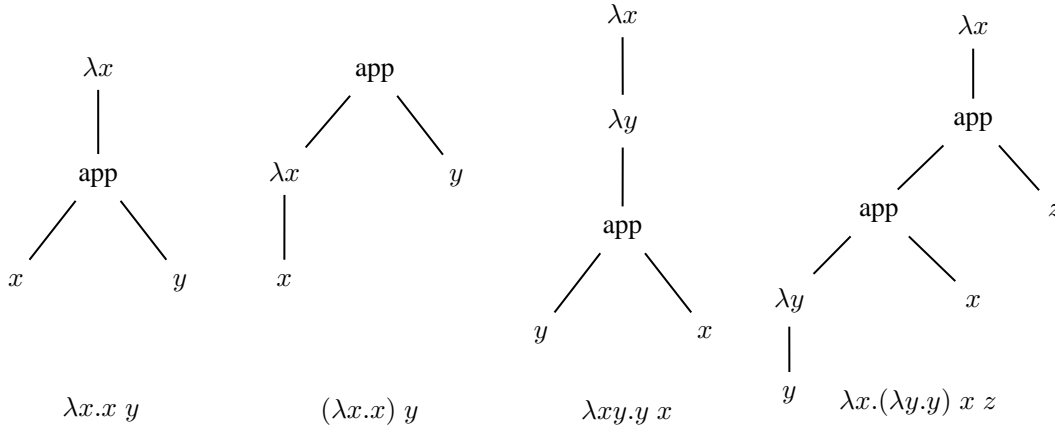
but only a compiler would be interested in so much detail.

For us it is important to know, that we need the `app` non-terminal in order to express λ -term's as derivations in the above grammar. It will always be used in a binary form to indicate applications of the form " $M M$ ".

92. Abstract Syntax Trees

Derivations of λ -terms in the above grammar are called abstract syntax trees. They omit non-terminals as far as possible, except for `app`.

Example:



The right-most abstract syntax tree is constructed by exploiting the fact that we can transform the functions in the term $\lambda x. (\lambda y. y) x z$ into functions with only one argument, i.e. $\lambda x. ((\lambda y. y) x) z$ (remember that `app` corresponds to $M M$). This transformation is called *Currying*.

93. α -Renaming

α renaming formally captures the idea that it is irrelevant how one names variables in a function. In a λ -term it does not really matter how we name the bound variables. In fact, we can rename a bound variable by replacing it in the binder and consequently also renaming all its occurrences in the scope. This technique is called α -renaming or α -conversion.

Example: We rename $(\lambda x. y x x)$ to $(\lambda z. y z z)$.

We say that two λ -terms are α -equivalent if one can be gained from the other by α renaming. For pure λ -terms α -equivalence can be very easily seen by comparing their abstract syntax trees.

Example: Obviously $(\lambda x. y x x)$ and $(\lambda z. y z z)$ are α -equivalent. However, they are not α -equivalent to $(\lambda z. r z z)$, since we r and y are constants or global variables and it is not known if they are equivalent. α -renaming seems trivial, but has to be applied cautiously. Firstly we are only allowed to rename only those variable occurrences that are bound to the same abstraction. For instance, renaming the first x in $\lambda x. \lambda x. x$ to y will result in $\lambda y. \lambda x. x$ but not in $\lambda y. \lambda x. y$. ($\lambda x. \lambda x. x$ might look odd, but think of it as a recursive call to the same function and in each recursion the function uses the variable x .) Secondly, we have to be careful that renamed variables are not “captured” by some binder they do not belong to. For example we cannot rename $\lambda x. \lambda y. x$ to $\lambda y. \lambda y. y$.

94. β -Reduction

λ -terms on their own would be a bit boring if we didn’t know how to *compute* with them as well. There is only one rule of computation, called *reduction* (or β -reduction, as it is known by the aficionados), and it concerns the replacement of a formal parameter by an actual one. It can only occur if a functional term has been applied to some other term.

Example:

$$\begin{aligned}
 (\lambda x. * 3 x) 4 &\longrightarrow_{\beta} * 3 4 \\
 (\lambda y. y 5)(\lambda x. * 3 x) &\longrightarrow_{\beta} (\lambda x. * 3 x) 5 \longrightarrow_{\beta} * 3 5
 \end{aligned}$$

We see that reduction is nothing other than the textual replacement of a formal parameter in the body of a function by the actual parameter supplied. Always carry out reductions from left to right and consume as many arguments as the binder contains.

Example:

$$\begin{aligned}
 (\lambda xy. x y) (\lambda z. z) (\lambda r. r r) c &\longrightarrow_{\beta} (\lambda y. (\lambda z. z) y) (\lambda r. r r) c \longrightarrow_{\beta} ((\lambda z. z) (\lambda r. r r)) c \longrightarrow_{\beta} \\
 &\longrightarrow_{\beta} (\lambda r. r r) c \longrightarrow_{\beta} c c
 \end{aligned}$$

One would expect a term after a number of reductions to reach a form where no further reductions are possible. Surprisingly, this is not always the case. The following is the smallest counterexample:

$$\Omega \stackrel{\text{def}}{=} (\lambda x. x x)(\lambda x. x x)$$

The term Ω always reduces to itself. If a sequence of reductions has come to an end where no further reductions are possible, we say that the term has been reduced to *normal form*. As Ω illustrates, not every term has a normal form.

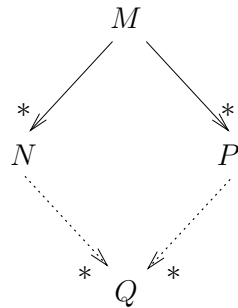
95. Confluence

It may be that a λ -term offers many opportunities for reduction at the same time. In order for the whole calculus to make sense, it is necessary that the result of a computation is independent from the order of reduction. We would like to express this property for all terms, not just for those which have a normal form. This is indeed possible.

Example: In the previous example we could have reduced the term $(\lambda y.(\lambda z.z) y) (\lambda r.r r) c$ differently by first reducing inside the first function and would have still arrived at the same result.

This fact is stated in the following theorem:

Theorem 30 (Church-Rosser) *If a term M can be reduced (in several steps) to terms N and P , then there exists a term Q to which both N and P can be reduced (in several steps). As a picture:*

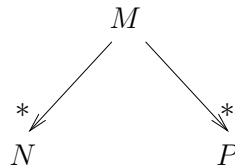


*(The little * next to the arrows indicates several instead of just a single reduction. “Several” can also mean “none at all”.)*

For obvious graphical reasons, the property expressed in the Theorem of Church and Rosser is also called *confluence*. We say that β -reduction is *confluent*. The following is now an easy consequence:

Corollary 31 *Every λ -term has at most one normal form.*

Proof. For the sake of contradiction, assume that there are normal forms N and P to which a certain term M reduces:



By the theorem of Church and Rosser there is a term Q to which both N and P can be reduced. However, N and P are assumed to be in normal form, so they don't allow for any further reductions. The only possible interpretation is that $N = P = Q$. █

Exercise Sheet 10

Quickies (I suggest that you try to do these before the Exercise Class. Hand in your solutions to the tutor at the end of the class.)

52. In which ways does **Java**'s way of defining functions (i.e. methods) differ from that of the λ -calculus? 1

Classroom exercises (Hand in to your tutor at the end of the exercise class.)

53. Translate the following **Java** expressions into λ -calculus notation:

- (a) `length(x+y)+z`
- (b) `cos(2+y)`
- (c)

```
public static int quot(double x, double n)
    { return (int)(x/n); }
```

2

54. Consider the λ -term $(\lambda f x.f(fx)) (\lambda y.* y 2) 5$.

- (a) Draw its abstract syntax tree.
- (b) β -reduce the term as far as possible. 1+1

Homework exercises (Submit via the correct pigeon hole before next Monday, 9am.)

55. Consider the λ -term $(\lambda xyz.g (x z) (y z)) (\lambda r.f r 1) (\lambda s.f s 2) 2$.

- (a) Draw its abstract syntax tree.
- (b) β -reduce the term as far as possible. 1+1

56. Consider the following lambda terms:

$$\begin{aligned} X &= \lambda x.((\lambda z.z x)((\lambda r.\lambda s.s r) y f)) \\ Y &= \lambda x.((\lambda z.\lambda u.\lambda v.u v z)x f y) \end{aligned}$$

- (a) Draw the abstract syntax trees for X and Y .
- (b) Use β -reduction to show the equality $X =_{\beta} Y$. 1+2

Stretchers (Problems in this section go beyond what we expect of you in the May exam. Please submit your solution through the special pigeon hole dedicated for these exercises. The deadline is the same as for the other homework.)

- S12. Suppose a symbol of the λ -calculus alphabet is always 5mm wide. Write down a pure λ -term (i.e., without constants) with length less than 20cm having a normal form with length at least $10^{10^{10}}$ light-years. (A light-year is about 10^{13} kilometers.) 10

Models of Computation Glossary 10

Abstraction	A λ -term that begins with a λ binder.	99
Alpha-Equivalent	Two λ -term are α -equivalent if one can be gained from the other by α renaming.	101
Alpha-Renaming	The process of renaming bound variable in a λ -term.	101
Application	A λ -term in which a function is applied to arguments (which may again be functions).	99
Beta-Reduction	A rewrite rule that computes the function application in a λ -term.	101
Binder	In a λ -term the binder is the prefix containing the lambda and the variables it abstracts over. The end of the binder is normally signalled by a dot.	99
Bound Variable	The variable in a lambda term over which the lambda expression abstracts. I.e. in the subsequent expression this variable can replace during β -reductions of the binder.	99
Church-Rosser	A different notion for confluence.	102
Confluence	A concept describing that a λ -term can be β -reduced to the same term in different ways.	102
Currying	Fully bracketing a λ -term by recursively applying each function to exactly one argument.	101
Formal Parameter	The bound variable in a lambda term.	99
Lambda Calculus	A model of computation that favours a functional view on computations and programming languages.	98
Lambda Term	A term in the lambda calculus.	98
Normal Form	A λ -term that cannot be further reduced by β -reduction steps. Not every λ -term can be rewritten to a normal form.	101
Prefix Notation	Notational convention in which functions are always written before all of its arguments.	98
Pure Term	A λ -term is called pure if it does not contain any constants.	100
Scope	The function body of a λ -term over which the preceding lambda binder ranges.	99