Functional Programming

Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 2

The LAMBDA CALCULUS

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

CONTENTS

- Target language and computational model
- Alonzo Church and the λ Calculus
- The λ Calculus and effective computability
- λ Calculus syntax – λ τerms, extended with constants, functions
- Rules for evaluation
- Terminology : “scope”, “bound” and “free”
- Normal forms and reduction strategies
- Examples

This lecture provides a basic introduction to the Lambda Calculus, including an explanation of its importance, genesis, definitions and some key terminology. The lecture will end with some examples.

Introductory background reading on the Lambda Calculus includes:

Foundations - a chapter from "Research Directions in Parallel Functional Programming ", by Hammond and Michaelson, (pub. Springer 1999), describing the lambda calculus (Section 2.3 only), and

The Lambda Calculus - Chapter 2 of "Implementation of Functional Programming Languages ", by Simon Peyton Jones (pub. Prentice Hall 1987)

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
Target language and computational model

Functional languages may be compiled into the λ Calculus, so it can serve as a target language

The λ Calculus is very simple: small syntax and few rules

Functions viewed as rules for generating an answer given a certain input, laying the foundation for computers and programming languages

A program is a function that returns a value and there are no "side effects"

Although the λ Calculus was initially conceived as being sequential, there are many non-sequential implementations

Target language and computational model

Functional languages may be compiled into the λ Calculus so it can serve as a target language (though there are other implementation routes, and the λ Calculus is then typically translated to a different run-time representation)

The λ Calculus is very simple — very small syntax and few rules, and provides the computational model for functional programming

The λ Calculus views functions as rules for generating an answer given a certain input — this is different to the "standard" notion of functions as point transformations between input and output values, and paved the way for a new perspective on calculations, "effectively calculable" and computations, laying the foundation for computers and programming languages

Notice that this approach views a program as a function that returns a value and there are no "side effects" such as file I/O or printing (though part of the output can for example be an instruction to an operating system to do these actions)

Although the λ Calculus was initially conceived as being sequential, there are many non-sequential implementations (e.g. much work was done in the 1980s to use functional languages - based on the λ Calculus - for parallel processing)

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
Alonzo Church and the λ Calculus

- Princeton University 1929-1967
- Mathematician, Logician and Philosopher
- The *Entscheidungsproblem* called for a decision procedure to decide whether a mathematical statement is Valid
- Needed a way to write any mathematical statement – a new "language" – the Lambda Calculus (or λ Calculus)
- First published in 1932 and 1933
- The first general-purpose programming language
- Used the λ **Calculus** to prove the *Entscheidungsproblem* was undecideable (1936) – "Church's Theorem"

Alonzo Church and the λ Calculus

Church was a Mathematician, Logician and Philosopher at Princeton University (1929-1967)

He was deeply interested in the "*Entscheidungsproblem*" – i.e. whether it is possible to have a procedure for deciding whether an **input statement** in mathematics is universally "Valid" or "Not Valid"

To answer the *Entscheidungsproblem*, first he had to design a way to write any mathematical statement. This was related to the search for a general theory of functions, and the search for a general foundation for logic and mathematics

So Church had to design a "language" for describing mathematical statements

This was the Lambda Calculus, first published in 1932 – it was not only a general way of writing mathematical statements, but also a mechanical way to **manipulate** those statements

This pre-dates the Turing Machine by four years, making the λ Calculus the first general-purpose programming system and programming language

In 1936, Church proved that the *Entscheidungsproblem* was in fact undecideable – this is "Church's Theorem"

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
The λ Calculus and effective computability

- Variants of the λ Calculus:
    - λ I Calculus (purely functional part) most successful theoretically
    - λ K Calculus is the most widely known and used
    - Can a function have a formal argument that does not appear "free" in the function body?
- Using the λ I Calculus Church defined "*effectively computable*" to mean "λ definability". Turing defined it to be "Turing computable" and Gödel and Herbrand defined it as general recursive functions
- Church-Turing thesis: all three are equivalent

The λ Calculus and effective computability

There are several variants of the λ Calculus, the most theoretically successful of which has been the purely functional part, sometimes referred to as the λI Calculus (published in 1941)

Though the λI Calculus was the most successful theoretically, in practice the variant that is most widely known and used is the λK Calculus. The difference between them is small – in the λI Calculus if a function has an argument variable then it must appear "free" in the function body, whereas in the λK Calculus it need not appear "free" in the function body. Later we will discuss what "free" means

Using the λI Calculus Church formalised the concept of "effectively computable" as being "λ definability". Turing developed a similar concept of "Turing computable", and in 1933 Gödel and Herbrand had developed a similar concept of general recursive functions

Church and Turing proved that the three concepts coincide. This "Church-Turing thesis" gives us confidence in our notions of what is, and what is not computable

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
λ K Calculus syntax – λ terms

- Formally **λ terms** are words over the following alphabet

  $v_0, v_1, \ldots$ *variables*
  λ *abstractor*
  (,) *parentheses*

  and the set of λ terms, called **Λ**, is defined as follows:

  $x \in \Lambda$ *variable*
  $M \in \Lambda \Longrightarrow (\lambda x\ M) \in \Lambda$ *abstraction*
  $M, N \in \Lambda \Longrightarrow (M\ N) \in \Lambda$ *application*

- Simple BNF syntax: e :: x | e e | λx e

- Alternative: e :: x | e e | λx.e

λK Calculus syntax – λ terms

Formally, terms in the λK Calculus are words formed from an alphabet that contains variables (v_0, v_1, …), the λ "abstractor", and parentheses

The set of all λ terms, called Λ, is defined as follows (where x is an arbitrary variable name):

- x is a member of the set Λ
- If M is a member of Λ then (λx M) is a member of Λ – we call this an "abstraction" and it signifies a function definition. For the λK Calculus the variable name "x" need not appear free inside M
- If M and N are both members of Λ then (M N) is a member of Λ – this signifies the application of the function abstraction M to the argument N

Using Backus Naur Form (BNF) a simple syntax of λ terms (sometimes called expressions) can be defined as follows: an expression "e" is either a variable name "x", or an expression "e" followed by another expression "e", or the abstractor λ followed by a variable name "x" and an expression "e"

Sometimes, in an abstraction, a dot is placed between the variable name "x" and the expression "e". Thus:

e :: x | e e | λx.e

Notice that function definitions (abstractions) do not have names, and there are no types

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
λ Calculus extended with constants

- Extensions to the simple BNF

  e ::= c | o | x | e e | λx. e | '(' e ')'

- We will not consider the typed λ Calculus

- For function abstractions with more than one argument:
  [see Currying (later lecture) and Schönfinkel, M. (1924) "Über die Bausteine der Mathematischen Logik", Math. Annalen 92, pp 305-316]

Either write: λx.(λy.e) or: λx. λy. e

In some books you might also see: λxy. e

Note that application associates to the left. Thus if M, N, P ∈ Λ then

M N P ≡ (M N) P

The type-free λK Calculus can compute anything that is computable. However, the minimal syntax is cumbersome

For practical purposes (for the examples we will give), we will now extend the BNF syntax with some "syntactic sugar":

- Constant values (such as 3, TRUE) will be denoted in the syntax by "c"
- Operators such as +, × will be denoted in the syntax by "o"

  Sometimes operators are included in the set of constants, but here we separate them for pedagogical reasons

  Initially, all operators are *prefix* (the operator appears to the left of its argument(s))
- Extra brackets for grouping (as appears in the formal alphabet for λ terms) are expressly included

The λ Calculus can be extended with types, but we will not consider the typed λ Calculus

For function abstractions with more than one argument we may write either:

λx.(λy.e) or: λx. λy. e

In some books and papers on the λ Calculus you may see function abstractions of more than one argument written as: λxy. e

Note that application associates to the left. Thus if M, N, P ∈ Λ then

M N P ≡ (M N) P

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
λ Calculus functions

- λ Calculus functions do NOT have names !
  - Functions can be defined but must be used immediately
- Function arguments DO have names
  - Those names can only be used inside the function body
- Functions can be arguments to other functions, and a function can return a function as its result (functions are *higher order*)
  - So a function can be named when it is passed as an argument to another function
  - and like any argument it can be used multiple times inside the other function

λ Calculus functions do NOT have names!

- Functions can be defined but must be used immediately (but see below)

Function arguments DO have names

- These names can only be used inside the function body (λI Calculus: one or more times, λK Calculus: zero or more times) [see later for "scope" of a λ x]

  From now on, when "λ Calculus" is written it means the λK Calculus unless specifically stated otherwise

Functions can be arguments to other functions, and a function can return a function as its result (functions are *higher order*)

- When a function is passed as an argument to another function it gets linked to the argument name
- In the term (λx. E) M, if M is a function it can be used inside E simply by using the name "x" and the name "x" can be used multiple times inside the function body "E"

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
Defining and applying λ Calculus functions

- The following is an anonymous function that adds 1 to *x*:

  λ*x* .((+ *x*)1)

- This is often simplified to the following:

  λ*x*.(+ *x* 1)

- We would prefer to simplify further:

  λ*x*.(*x* + 1)

  However, this would require a further extension to the syntax

- The following applies the above anonymous function to the constant integer 3:

  (λ*x*.(+ *x* 1)) 3

Defining and applying λ Calculus functions

The following λ term (using the λ Calculus extended with constants) defines an anonymous function that takes one argument (the argument is called *x*), adds 1 to *x,* and returns the sum as its result:

λ*x* .((+*x*)1)

This correspond to the λ term λx.E where E itself corresponds to the λ term M N where M is (+ x) and N is 1. Note that (+ x) also corresponds to the λ term M N with M being + and N being x

Because application is left-associative, this is often simplified to the following:

λ*x*.(+ *x* 1)

For human readability, we would prefer to simplify the expression further to this:

λ*x*.(*x* + 1)

However, this would require a further extension to the syntax to permit operators to appear in infix position – see the next slide

The following λ term applies the previously defined anonymous function to the constant integer 3:

(λ*x*.(+ *x* 1)) 3

This corresponds to the λ term for application: M N, where M is the function (λ*x*.(+ *x* 1)) and N is the argument 3

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING

λ Calculus syntax extended with infix operators

e ::= c | o | x | e e | e o e | λx. e | ‘(‘ e ‘)’

- We can now write the previous application in three different ways:

  (λ*x* .((+*x*)1)) 3

  (λ*x*.(+ *x* 1)) 3

  (λ*x*.(*x* + 1)) 3

Infix operators

To make it easier to write (and understand) example λ Calculus expressions we now further extend the λ Calculus syntax to include infix operators

In this BNF definition we have included the possibility “e o e”, meaning an expression followed by an operator followed by an expression – this is exactly the syntax we want for infix operators of two arguments

We can now write the previous application in three different but equivalent ways:

(λ*x* .((+*x*)1)) 3

(λ*x*.(+ *x* 1)) 3

(λ*x*.(*x* + 1)) 3

Notice how brackets are used to ensure that there is no ambiguity in the expression

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
Rules for evaluation

α Reduction

$\lambda x.E \longrightarrow \lambda y.E[y/x]$

β reduction

$(\lambda x.E)\ M \longrightarrow E[M/x]$

η reduction

$\lambda x.(E\ x) \longrightarrow E$ if x is not free in E

δ Rules (only for λ Calculus extended with constants)

Each operator (such as +, ×) has its own δ rule

Rules for evaluation

The λ Calculus also provides 4 rules for manipulating λ terms. This corresponds to "evaluation steps" for λ terms

α Reduction

- $\lambda x.E \longrightarrow \lambda y.E[y/x]$
- E[y/x] means "E with all free occurrences of x replaced by y"
- This permits the renaming of the formal arguments for a function

β reduction

- $(\lambda x.E)\ M \longrightarrow E[M/x]$
- A function applied to an argument "M" may be replaced by the body of the function ("E") wherein all free occurrences of "x" have been replaced by the argument "M"
- This is the primary mechanism for evaluating function application

η reduction

- $\lambda x.(E\ x) \longrightarrow E$ if x is not free in E
- This is an optimisation step – if there are no free occurrences of "x" in "E" then for any λ term "M" we have by β reduction:

  $(\lambda x.(E\ x))\ M \longrightarrow E\ M$
  and therefore $\lambda x.(E\ x) = E$

  [extensionality rule]

δ Rules (only λ Calculus with constants)

Each operator (such as +, ×) has its own δ rule: e.g. the δ rule for + replaces (3+4) with 7

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
Terminology

1. "*scope*": In λ x.E the "scope" of λ x is E and a variable x is "in the scope of" the λ x if it occurs free in E

   E.g. in λ x.(x + y) the scope of the λ x is (x + y), and the x in the term (x + y) is in the scope of the λ x

2. "*free*" and "*bound*": A variable x is "free" in M if x is not in the scope of a λ x; x occurs "bound" otherwise

   E.g. (i) in x (λ y.(x y)) x is free (twice) and y is bound

   (ii) in y (λ y. y) y is free once and bound once

3. When a function is applied to an argument a "binding" is created. E.g. in. (λ x. (x + 1)) 34 we say that x is bound to the value 34

Terminology

1. "scope"

In the term λx.E the "scope" of λx is E and a variable x is "in the scope of" the λ x if it occurs free in E

For example, in the term λx.(x + y) the scope of the λx is the term (x + y), and the x in the term (x + y) is in the scope of the λx

2. "free" and "bound"

A variable x appears "free" in a term M if x is not in the scope of a λx; x occurs "bound" otherwise

For example:

- in the term x (λy.(x y)) the variable x occurs free (twice) and y occurs bound
- In the term y(λy. y) the variable y occurs free once and occurs bound once

"Not bound" is a synonym for "free"

3. When a function abstraction is applied to an argument a "binding" is created.

For example, in the term (λx. (x + 1)) 34 we say that the binding for x is the value 34 – or, more briefly, that x is bound to the value 34

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
Defining the free variables

- FV(M) is the set of free variables of λ term M

  FV(x) = { x }

  FV(λx.M) = FV(M) – { x }

  FV(M N) = FV(M) ∪ FV(N)

- If FV(M) = Ø then we say that M is a combinator

Defining the free variables

Let FV(M) be the set of free variables of λ term M, defined on the basic definition of λ terms as follows:

FV(x) = { x }

FV(λx.M) = FV(M) – { x }

FV(M N) = FV(M) ∪ FV(N)

Note that if FV(M) = Ø then we say that M is a *combinator*

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
Defining substitution E[y/x]

- α reduction and β reduction rely on the notation E[N/x], meaning "E with all free occurrences of x replaced by N"
- Here is a formal definition of E[N/x] for any E

$x[N/x] \equiv N$

$y[N/x] \equiv y, \quad \text{if } x \not\equiv y$

$(\lambda y.M)[N/x] \equiv \lambda y.(M[N/x]), \text{ if } x \not\equiv y \text{ and } y \notin FV(N)$

$(M_1\ M_2)[N/x] \equiv (M_1[N/x])(M_2[N/x])$

Defining substitution E[y/x]

The definitions of α reduction and β reduction rely on the notation E[N/x], meaning "E with all free occurrences of x replaced by N"

Here is a formal definition on the basic definition of λ terms (i.e. that E can be x, M N, or λx.M)

$x[N/x] \equiv N$

$y[N/x] \equiv y, \quad \text{if } x \not\equiv y$

$(\lambda y.M)[N/x] \equiv \lambda y.(M[N/x]), \text{ if } x \not\equiv y \text{ and } y \notin FV(N)$

$(M_1\ M_2)[N/x] \equiv (M_1[N/x])(M_2[N/x])$

Note that in the third case if the constraints do not hold (i.e. either $x \equiv y$ or $y \in FV(N)$) then no replacement can occur

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
Normal Form and reduction strategies

- If a λ term M matches the left hand side of a reduction rule then M is called a "redex"
- To evaluate a λ term we successively apply reduction rules until neither the term nor any of its sub-terms is a β redex - this end result is called "normal form"

1. M "**is a" (or "is in")** normal form (or β-normal form) if M has no subterm (λx.R)S
2. M "**has"** a normal form if ∃ an N where M is convertible to N (using reduction rules) and N is a normal form

- A reduction strategy is a defined consistent choice of which redex to reduce next. For example "always reduce the leftmost redex next"

Normal Form and reduction strategies

If a λ term M matches the left hand side of a reduction rule (either α, β, η or δ) then M is called a "reducible expression" or "redex"

To evaluate a λ term we successively apply reduction rules (α, β, η or δ) until neither the term nor any of its sub-terms is a β redex - this end result is called "normal form"

For the λK Calculus extended with constants, we would extend the definition to say that normal form also requires that no δ redex must exist

Formally, for the unextended λK Calculus :

1. M "**is a" (or "is in")** normal form (or β-normal form) if M has no subterm (λx.R)S
2. M "**has"** a normal form if ∃ an N such that M is convertible to N (using reduction rules) and N is a normal form

A large λ term may contain many redexes and we must choose which to reduce first. A reduction strategy is (informally) a defined consistent choice of which redex to reduce next. For example "always reduce the leftmost redex next" (noting that following the reduction the whole λ term must be checked again to see which is the leftmost redex)

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
Normal Form and reduction strategies

- Church-Rosser theorem: all reduction orders lead to the same **unique** normal form **if they terminate**
- The leftmost-first reduction strategy ("Normal Order" reduction) is guaranteed to terminate if termination is possible
- Normal Order reduction evaluates a function argument only if used by the function. This corresponds to the programming notion of "call by reference"
- Applicative Order reduction entails evaluating a function argument before evaluating the result of applying the function to the argument – this corresponds to "call by value"

Does it matter what reduction strategy we use?

Fortunately the Church-Rosser theorem proves that all reduction orders (including a random choice of which redex to reduce next) lead to the same **unique** normal form if they terminate. Unfortunately some reduction orders may lead to non-termination even though a different reduction order might terminate

The leftmost-first reduction strategy (also known as Normal Order reduction) is guaranteed to terminate if termination is possible

Normal Order reduction is often referred to as "leftmost-outermost-first" to emphasise the fact that a function argument is not evaluated in advance, but only if and when used by the function body (this is important for the λK Calculus where for abstraction $\lambda x.M$ the variable "x" need not appear inside the function body M). This corresponds to the programming notion of "call by reference"

By contrast a strategy known as Applicative Order reduction entails evaluating a function argument before evaluating the result of applying the function to the argument – this corresponds to "call by value" and is necessary for many δ rules (e.g. + cannot add its arguments until they have been evaluated). It is sometimes referred to as "leftmost-innermost-first"

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
Examples

- Example 1
  (5+3) →
  8

- Example 2
  (λ*x* .(*x* + 3)) 5 →
  (5 + 3) →
  8

- Example 3 :
  (λ*y*.((λ*x*.(*x* + *y*)) 5)) 3 →
  (λ*x* .(*x* + 3)) 5 →
  (5 + 3)
  8

Examples

Example 1 – using our extended syntax, the λ term (5 + 3) δ-reduces to the number 8

Example 2 – the term (λ*x* .(*x* + 3)) 5 β-reduces to (5 + 3) which then δ-reduces to 8

Example 3 – the term (λ*y*.((λ*x*.(*x* + *y*)) 5)) 3 β-reduces to (λ*x* .(*x* + 3)) 5 which β-reduces to (5 + 3) which then δ-reduces to 8

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
Examples

- Example 4 :
  $(\lambda x.((\lambda x.(x + 3))\ x))\ 5 \rightarrow$
  $(\lambda y.((\lambda x.(x + 3))\ y))\ 5 \rightarrow$
  $(\lambda x\ .(x + 3))\ 5 \rightarrow$
  $(5 + 3) \rightarrow$
  8

- Example 5 :
  $(\lambda x\ .(x\ 5))\ (\lambda x\ .(x + 3)) \rightarrow$
  $((\lambda x\ .(x + 3))\ 5) \rightarrow$
  $(5 + 3) \rightarrow$
  8

Here are two more examples

Example 4 – the λ term
$(\lambda x.((\lambda x.(x + 3))\ x))\ 5$
contains two nested function abstractions, both using the variable name "x". We therefore first α-reduce the outermost abstraction, changing its variable name from "x" to "y". In its body, only the (single) free occurrence of the name "x" is changed, as follows:
$(\lambda y.((\lambda x.(x + 3))\ y))\ 5$ which β-reduces to
$(\lambda x\ .(x + 3))\ 5$ which β-reduces to
$(5 + 3)$ which δ-reduces to
8

Example 5 – this example applies a function to a function argument. The term
$(\lambda x\ .(x\ 5))\ (\lambda x\ .(x + 3))$ β-reduces to
$((\lambda x\ .(x + 3))\ 5)$
we can remove the outer brackets to give:
$(\lambda x\ .(x + 3))\ 5$ which β-reduces to
$(5 + 3)$ which δ-reduces to
8

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
Examples

- Example 6 :
  $(\lambda x.((x\ 5) + (x\ 4)))\ (\lambda x.(x + 3))$ →
  $((\lambda x.(x + 3))\ 5) + ((\lambda x.(x + 3))\ 4)$ →
  $(5 + 3) + ((\lambda x.(x + 3))\ 4)$ →
  $(5 + 3) + (4 + 3)$ →
  $8 + (4 + 3)$ →
  $8 + 7$ →
  15

Here is a final example

Example 6 – this example applies a function to a function argument and then uses the argument twice inside its function body:

$(\lambda x.((x\ 5) + (x\ 4)))\ (\lambda x.(x + 3))$
β-reduces to
$((\lambda x.(x + 3))\ 5) + ((\lambda x.(x + 3))\ 4)$
from which we choose to reduce the leftmost redex and therefore β-reduces to
$(5 + 3) + ((\lambda x.(x + 3))\ 4)$
and we next choose to reduce the β-redex to give
$(5 + 3) + (4 + 3)$
and choosing the leftmost δ-redex this δ-reduces to
$8 + (4 + 3)$
and because the δ-rule for + requires both arguments to be fully evaluated, we must next δ-reduce the (4+3) to give
$8 + 7$ which δ-reduces to
15

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

In summary, this lecture has given a basic introduction to the Lambda Calculus, including an explanation of its importance, genesis, definitions and some key terminology. The lecture ended with some examples.

SUMMARY

- Target language and computational model
- Alonzo Church and the λ Calculus
- The λ Calculus and effective computability
- λ Calculus syntax – λ τerms, extended with constants, functions
- Rules for evaluation
- Terminology : “scope”, “bound” and “free”
- Normal forms and reduction strategies
- Examples

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS