Chapter 2

λ-Calculus

In traditional imperative programming (like C, Java, Python), we have a clear distinction between *programs*, that are sequences of instructions that are executed sequentially, and *data*, that are values given in input, stored in memory, manipulated during computation and returned as output. Programs and data are distinct and are kept separated. Programs are not modified during computation. (It is in theory possible to do it, since programs are stored in memory like any other data. However, it is difficult and dangerous.)

Functional Programming is a different paradigm of computation in which there is no distinction between programs and data. Both are represented by terms/expressions belonging to the same language. Computation consists in the reduction of terms to normal form. That includes terms that represent functions, that is, the programs themselves.

The pure realization of this idea is the λ-calculus. It is a pure theory of functions with only one kind of objects: λ-terms. They represent both data structures and programs.

The main idea is the definition of functions by *abstraction*. For example, we may define a function f on numbers by saying that $f(x) = x^2 + 3$. By this we mean that any argument to the function, represented by the variable x, is squared and added to 3. The use of variables is different from imperative programming: x is just a place-holder to denote any possible value, while in imperative programming variables represent memory locations containing values that can be modified.

We can specify the function f alternatively with the *mapping* notation:

$$x \xmapsto{f} x^2 + 3.$$

This is written in λ-*notation* as: $f = \lambda x.x^2 + 3$. (In the functional programming language Haskell, it is `\x -> x^2+3`.)

While abstraction is the operation to define a new function, computing it on a specific argument is called *application*. We indicate it simply by juxtaposition:

$$f\,5 = (\lambda x.x^2 + 3)\,5 \leadsto 5^2 + 3 \leadsto^* 28.$$

As the example shows, the application of a λ-abstraction to an argument is computed by replacing the abstraction variable with the argument. This is called *β-reduction* and it is the basic computation step of λ-calculus.

The λ-notation is convenient to define functions, but you may think that the actual computation work is done by the operations used in the body of the abstraction: squaring and adding 3. However, the λ-calculus is *a theory of pure functions*: terms are constructed using only abstraction an application, there are no other basic operations. At first, this looks like a rather useless system: no numbers, no arithmetic operations, no data structures, no programming primitives. The surprising fact is that we don't really need them. We don't need numbers (5, 3) and we don't need operations ($-^2$, $+$). They all can be defined as purely functional constructions, built using only abstraction and application!

Syntax of λ-Calculus

The language of λ-calculus is extremely simple, we start with variables and construct terms using only abstraction and computation. It's BNF definition is as follows (assume x, y, z range over a given infinite set of variable names).

$$\begin{array}{rll} t ::= & x \mid y \mid z \mid \cdots & \text{variable names} \\ \mid & \lambda x.t & \text{abstraction} \\ \mid & t\,t & \text{application.} \end{array}$$

The β-reduction relation on term is defined, for every pair of terms t_1 and t_2 as:

$$(\lambda x.t_1)\,t_2 \rightsquigarrow_\beta t_1[x := t_2].$$

The left-hand side means: in t_1 substitute all occurrences of variable x with the term t_2. Substitution is actually quite tricky and its precise definition is a bit more complex that replacing every occurrences of x with t_2. One has to be careful to manage variable occurrences properly.

We need some intermediate concept. The first is α-equivalence and it says that, since the variable in an abstraction is just a place holder for an argument, the names of abstracted variables does not matter. For example, the simplest function we can define is the identity $\lambda x.x$ which takes an argument x and returns it unchanged. Clearly, if we use a different variable name, $\lambda y.y$, we get exactly the same function. We say that the two terms (and any using different variables) are α-equivalent:

$$\lambda x.x =_\alpha \lambda y.y =_\alpha \lambda z.z =_\alpha \cdots.$$

We are free to change the name of the abstracted variable any way we like. However, we have to be careful to avoid *variable capture*. If the body of the abstraction contains other variables than the abstracted one, we can't change the name to those:

$$\lambda x.y\,(x\,z) =_\alpha \lambda w.y\,(w\,z) \neq_\alpha \lambda y.y\,(y\,z) \neq_\alpha \lambda z.y\,(z\,z).$$

The reason for this restriction is that changing the name of the abstracted variable from x to either y or z in this example would capture the occurrence of that variable which was *free* in the original term (not bound by a λ-abstraction).

Formally, we define set $\mathsf{FV}(t)$ of the variables that occur free in the term t, by recursion on the structure of t:

$$\begin{array}{l}\mathsf{FV}(x) = \{x\} \\ \mathsf{FV}(\lambda x.t) = \mathsf{FV}(t) \setminus \{x\} \\ \mathsf{FV}(t_1\,t_2) = \mathsf{FV}(t_1) \cup \mathsf{FV}(t_2).\end{array}$$

Another way in which a variable can be incorrectly captured is when we perform a substitution that puts a term under an abstraction that may bind some of its variables:

$$(\lambda x.\lambda y.x\,y)\,(y\,z) \rightsquigarrow_\beta (\lambda y.x\,y)[x := (y\,z)] \neq \lambda y.(y\,z)\,y.$$

If we replace x with $(y\,z)$ in this way, the occurrence of the variable y (which was free before performing the β-reduction) become bound. This is incorrect. We should rename the abstraction variable before performing the substitution:

$$(\lambda y.x\,y)[x := (y\,z)] =_\alpha (\lambda w.x\,w)[x := (y\,z)] = \lambda w.(y\,z)\,w.$$

To avoid problems with variable capture, we adopt the *Barendregt variable convention*: before performing substitution (or any other operation on terms) change the names of the abstracted variables so they are different from the free variables and from each other.

With this convention, we can give a precise definition of substitution by recursion on the structure of terms:

$$\begin{array}{rll}x[x := t_2] &= t_2 & \\ y[x := t_2] &= y & \text{if } y \neq x \\ (\lambda y.t_1)[x := t_2] &= \lambda y.t_1[x := t_2] & \\ (t_0\,t_1)[x := t_2] &= t_0[x := t_2]\,t_1[x := t_2]. &\end{array}$$

In the third case, the variable convention ensures that the variable y and the bound variables in t_2 have already been renamed so they avoid captures. In more traditional formulations, one would add the requirements: "provided that $y \neq x$ and y doesn't occur free in t_2".

A part from some complication about substitution, the λ-calculus is extremely simple. It seems at first surprising that we can actually do any serious computation with it at all. But it turns out that all computable functions can be represented by λ-terms. We see some simple function in this section and we will discover how to represent data structures in the next.

For convenience, we use some conventions that allow us to save on parentheses.

- λ-abstraction associates to the right, so we write $\lambda x.\lambda y.x$ for $\lambda x.(\lambda y.x)$;
- Application associates to the left, so we write $(t_1\,t_2\,t_3)$ for $((t_1\,t_2)\,t_3)$;

- We can use a single λ symbol followed by several variables to mean consecutive abstractions, so we write $\lambda x\,y.x$ for $\lambda x.\lambda y.x$.

Here are three very simple functions implemented as λ-terms:

- The identity function $\lambda x.x$. When applied to an argument it simply returns it unchanged:
$$(\lambda x.x)\,t \leadsto x[x := t] = t.$$

- The first projection function $\lambda x.\lambda y.x$. When applied to two arguments, it returns the first:
$$(\lambda x.\lambda y.x)\,t_1\,t_2 \leadsto (\lambda y.x)[x := t_1]\,t_2 = (\lambda y.t_1)\,t_2 \leadsto t_1[y := t_2] = t_1.$$
Remember, in reading this reduction sequence, that we are adopting the variable convention, so the variable y doesn't occur free in t_1.

- The second projection function $\lambda x.\lambda y.x$. When applied to two arguments, it returns the second:
$$(\lambda x.\lambda y.y)\,t_1\,t_2 \leadsto (\lambda y.y)[x := t_1]\,t_2 = (\lambda y.y)\,t_2 \leadsto y[y := t_2] = t_2.$$
We can also say that the second projection is the function that, when applied to an argument t_1, returns the identity function $\lambda y.y$.

Church Numerals

So far we have seen only some very basic functions that only return some of their arguments unchanged. How can we define more interesting computations? And first of all, how can we represent values and data structures? It is in fact possible to represent any kind of data buy some λ-term.

Let's start by representing natural numbers. Their encodings in λ-calculus are called *Church Numerals*:

$$\begin{array}{l}\overline{0} := \lambda f.\lambda x.x\\ \overline{1} := \lambda f.\lambda x.f\,x\\ \overline{2} := \lambda f.\lambda x.f\,(f\,x)\\ \overline{3} := \lambda f.\lambda x.f\,(f\,(f\,x))\\ \cdots.\end{array}$$

A numeral $\overline{n}$ is a function that takes two arguments, denoted by the variables f and x, and applies f sequentially n times to x.

What is important is that we assign to every number a distinct λ-term in a uniform way. We must choose our representation so it is easy to represent arithmetic operations. The idea of Church numerals is nicely conceptual: numbers are objects that we use to count things, so we can define them as the counters of repeated application of a function.

Let's see if this representation is convenient from the programming point of view: can we define basic operations on it?

Let's start with the successor function, that increases a number by one:

$$\mathsf{succ} := \lambda n.\lambda f.\lambda x.f\,(n\,f\,x).$$

Let's test if it works on an example: if we apply it to $\overline{2}$ we should get $\overline{3}$:

$$\begin{array}{ll}\mathsf{succ}\,\overline{2} & = (\lambda n.\lambda f.\lambda x.f\,(n\,f\,x))\,\overline{2}\\ & \rightsquigarrow \lambda f.\lambda x.f\,(\overline{2}\,f\,x) = \lambda f.\lambda x.f\,((\lambda f.\lambda x.f\,(f\,x))\,f\,x)\\ & \rightsquigarrow \lambda f.\lambda x.f\,((\lambda x.f\,(f\,x))[f := f]\,x) = \lambda f.\lambda x.f\,((\lambda x.f\,(f\,x))\,x)\\ & \rightsquigarrow \lambda f.\lambda x.f\,((f\,(f\,x))[x := x]) = \lambda f.\lambda x.f\,(f\,(f\,x)) = \overline{3}.\end{array}$$

We have explicitly marked the substitutions in this reduction sequence: they are both trivial, substituting f with itself and x with itself. From now on, we'll do the substitutions on the fly, without marking them.

Other arithmetic operations can be defined by simple terms:

$$\begin{array}{l}\mathsf{plus} := \lambda m.\lambda n.\lambda f.\lambda x.m\,f\,(n\,f\,x)\\ \mathsf{mult} := \lambda m.\lambda n.\lambda f.m\,(n\,f)\\ \mathsf{exp} := \lambda m.\lambda n.n\,m.\end{array}$$

Verify by yourself that these terms correctly implement the addition, multiplication and exponentiation functions. For example:

$$\begin{array}{l}\mathsf{plus}\,\overline{2}\,\overline{3} \rightsquigarrow^* \overline{5}\\ \mathsf{mult}\,\overline{2}\,\overline{3} \rightsquigarrow^* \overline{6}\\ \mathsf{exp}\,\overline{2}\,\overline{3} \rightsquigarrow^* \overline{8}\\ \mathsf{exp}\,\overline{2}\,\overline{3} \rightsquigarrow^* \overline{9}.\end{array}$$

Surprisingly, two other basic functions are much more difficult to define: the predecessor and the (cut-off) subtraction functions. Try two define two terms pred and minus such that:

$$\begin{array}{l}\mathsf{pred}\,\overline{3} \rightsquigarrow^* \overline{2}\\ \mathsf{pred}\,\overline{0} \rightsquigarrow^* \overline{0}\\ \mathsf{minus}\,\overline{5}\,\overline{2} \rightsquigarrow^* \overline{3}\\ \mathsf{minus}\,\overline{2}\,\overline{5} \rightsquigarrow^* \overline{0}.\end{array}$$

Other Data Structures

Other data types can be encoded in the λ-calculus.

Booleans For truth values we may choose the first and second projects that we defined earlier:

$$\begin{array}{l}\mathsf{true} := \lambda x.\lambda y.x\\ \mathsf{false} := \lambda x.\lambda y.y.\end{array}$$

We must show how to compute the logical operators. For example, conjunction can be defined as follows:

$$\mathsf{and} := \lambda a.\lambda b.a\,b\,\mathsf{false}.$$

Let's verify that it give the correct results when applied to Boolean values:

$$\begin{aligned}
\mathsf{and}\,\mathsf{true}\,\mathsf{true} &= (\lambda a.\lambda b.a\,b\,\mathsf{false})\,\mathsf{true}\,\mathsf{true}\\
&\rightsquigarrow^* \mathsf{true}\,\mathsf{true}\,\mathsf{false} = (\lambda x.\lambda y.x)\,\mathsf{true}\,\mathsf{false} \rightsquigarrow^* \mathsf{true}\\
\mathsf{and}\,\mathsf{true}\,\mathsf{false} &= (\lambda a.\lambda b.a\,b\,\mathsf{false})\,\mathsf{true}\,\mathsf{false}\\
&\rightsquigarrow^* \mathsf{true}\,\mathsf{false}\,\mathsf{false} = (\lambda x.\lambda y.x)\,\mathsf{false}\,\mathsf{false} \rightsquigarrow^* \mathsf{false}\\
\mathsf{and}\,\mathsf{false}\,t &= (\lambda a.\lambda b.a\,b\,\mathsf{false})\,\mathsf{false}\,t\\
&\rightsquigarrow^* \mathsf{false}\,t\,\mathsf{false} = (\lambda x.\lambda y.y)\,t\,\mathsf{false} \rightsquigarrow^* \mathsf{false}
\end{aligned}$$

All the other logical operators could be defined if we had a conditional construct $\mathsf{if} - \mathsf{then} - \mathsf{else} -$. In fact this can be defined very easily:

$$\mathsf{if} := \lambda b.\lambda u.\lambda v.b\,u\,v.$$

Let's verify that it has the correct computational behaviour:

$$\begin{aligned}
\mathsf{if}\,\mathsf{true}\,t_1\,t_2 &= (\lambda b.\lambda u.\lambda v.b\,u\,v)\,\mathsf{true}\,t_1\,t_2\\
&\rightsquigarrow^* \mathsf{true}\,t_1\,t_2 = (\lambda x.\lambda y.x)\,t_1\,t_2 \rightsquigarrow^* t_1\\
\mathsf{if}\,\mathsf{false}\,t_1\,t_2 &= (\lambda b.\lambda u.\lambda v.b\,u\,v)\,\mathsf{false}\,t_1\,t_2\\
&\rightsquigarrow^* \mathsf{false}\,t_1\,t_2 = (\lambda x.\lambda y.y)\,t_1\,t_2 \rightsquigarrow^* t_2.
\end{aligned}$$

Then we can define all logical connectives as conditionals, for example:

$$\mathsf{and} := \lambda a.\lambda b.\mathsf{if}\,a\,b\,\mathsf{false}, \quad \mathsf{or} := \lambda a.\lambda b.\mathsf{if}\,a\,\mathsf{true}\,b, \quad \mathsf{not} := \lambda a.\mathsf{if}\,a\,\mathsf{false}\,\mathsf{true}.$$

Tuples Pairs of λ-terms can be encoded by a single term: If t_1 and t_2 are terms, we define the encoding of the pair as

$$\langle t_1, t_2\rangle := \lambda x.x\,t_1\,t_2.$$

First and second projections are obtained by applying a pair to the familiar projections (or truth values) that we have already seen:

$$\begin{aligned}
\mathsf{fst}\,p &= p\,(\lambda x.\lambda y.x)\\
\mathsf{snd}\,p &= p\,(\lambda x.\lambda y.y)
\end{aligned}$$

We can verify that they have the correct reduction behaviour:

$$\begin{aligned}
\mathsf{fst}\,\langle t_1,t_2\rangle &= \langle t_1,t_2\rangle\,(\lambda x.\lambda y.x) = (\lambda x.x\,t_1\,t_2)\,(\lambda x.\lambda y.x) \rightsquigarrow (\lambda x.\lambda y.x)\,t_1\,t_2 \rightsquigarrow^* t_1,\\
\mathsf{snd}\,\langle t_1,t_2\rangle &= \langle t_1,t_2\rangle\,(\lambda x.\lambda y.y) = (\lambda x.x\,t_1\,t_2)\,(\lambda x.\lambda y.y) \rightsquigarrow (\lambda x.\lambda y.y)\,t_1\,t_2 \rightsquigarrow^* t_2.
\end{aligned}$$

Triples and longer tuples may be encoded as repeated pairs, for example $\langle t_1, t_2, t_3\rangle := \langle t_1, \langle t_2, t_3\rangle\rangle$, or directly using the same idea as for pairs: $\langle t_1, t_2, t_3\rangle := \lambda x.x\,t_1\,t_2\,t_3$.

Lists **TO DO** [They can be represented as repeated tuples. But we need to have a nil element to represent the empty list. This can be done by putting a Boolean in front to signal whether the list is empty or continues.]

Binary Trees **TO DO** [Similarly: tupling the label and the subtrees, with a Boolean to signal the leaves.]

Syntax Trees

As for any structured language, there is a representation of λ-terms as abstract syntax trees. We have binary nodes representing application, unary nodes representing abstraction and leaves representing variable occurrences. For example, the Church numeral $\overline{2} = \lambda f.\lambda x.f\,(f\,x)$ is represented by this tree:

$$\begin{tikzcd}
& \lambda f \arrow[d] & & \\
& \lambda x \arrow[d] & & \\
& \mathsf{app} \arrow[dl] \arrow[dr] & & \\
f & & \mathsf{app} \arrow[dl] \arrow[dr] & \\
& f & & x
\end{tikzcd}$$

A more efficient representation uses *term graphs*: bound variable occurrences point to the abstraction node for that variable and we allow *sharing* by letting different edges point at the same subgraph:

$$\begin{tikzcd}
\overline{2} = & \lambda f \arrow[d] & \\
& \lambda x \arrow[d] & \\
& \mathsf{app} \arrow[uu, bend left=60] \arrow[dr] & \\
& & \mathsf{app} \arrow[uuul, bend left=80] \arrow[uul, bend right=40]
\end{tikzcd}
\qquad
\begin{tikzcd}
\lambda x.\lambda y.(y\,x)\,(y\,x) = & \lambda x \arrow[d] \\
& \lambda y \arrow[d] \\
& \mathsf{app} \arrow[d, bend right] \arrow[d, bend left] \\
& \mathsf{app} \arrow[uu, bend left=60] \arrow[uuu, bend right=60]
\end{tikzcd}$$

In this formalism, we don't need to explicitly write the variable names on the abstraction nodes, since they are not necessary to determine which variable occurrences are bound by them. This is convenient, because we don't need to worry about α-equivalence any more:

$$\overline{2} = \begin{tikzcd}
\lambda \arrow[d] & \\
\lambda \arrow[d] & \\
\mathsf{app} \arrow[u, bend left=60] \arrow[uu, bend left=70] \arrow[dr] & \\
& \mathsf{app} \arrow[uuul, bend left=70] \arrow[uul, bend right=40]
\end{tikzcd}
\qquad
\lambda x.\lambda y.(y\,x)\,(y\,x) = \begin{tikzcd}
\lambda \arrow[d] \\
\lambda \arrow[d] \\
\mathsf{app} \arrow[d, bend right] \arrow[d, bend left] \\
\mathsf{app} \arrow[uu, bend left=70] \arrow[uuu, bend right=70]
\end{tikzcd}$$

Representing terms in this way is more efficient in several ways. When we make some reduction rules inside a subterm, we have to do it just ones, while if every occurrence were represented by a separate subtree, we would have to reduce each one separately. One has to be careful anyway: sometimes terms are modified in one way in one branch but not on others. In that case the subgraph has to be duplicated.

Confluence

A λ-term may contain several redexes. We have the choice of which one to reduce first. When we make one step of β-reduction, some of the redexes that were there in the beginning may disappear, some may be duplicated into many copies, some new ones may be created. Although we say that β-reduction is a "simplification" of the term, in the sense that we eliminate a pair of consecutive abstraction and application by immediately performing the associated substitution, the resulting reduced term is not always simpler. It may actually be much longer and complicated.

Therefore it is not obvious that, if we choose different redexes to simplify, we will eventually get the same result. It is also not clear whether the reduction of a term will eventually terminate.

The first property is anyway true. But there are indeed terms whose reduction does not terminate.

Theorem 7 (Confluence) *Given any λ-term t, if t_1 and t_2 are two reducts of it, that is $t \rightsquigarrow^* t_1$ and $t \rightsquigarrow^* t_2$; then there exists a common reduct t_3 such that $t_1 \rightsquigarrow^* t_3$ and $t_2 \rightsquigarrow^* t_3$.*

TO DO [Proof]

Definition 8 *A* normal form *is a λ-term that doesn't contain any redexes. A term* weakly normalizes *if there is a sequence of reduction steps that ends in a normal form. A term* strongly normalizes *if any sequence of reduction steps eventually ends in a normal form.*

There exist terms that do not normalize. The most famous one is a very short expression that reduces to itself:

$$\begin{aligned} \omega \quad &:= (\lambda x.x\,x)\,(\lambda x.x\,x) \\ &\rightsquigarrow (x\,x)[x := \lambda x.x\,x] = (\lambda x.x\,x)\,(\lambda x.x\,x) = \omega \\ &\rightsquigarrow \omega \rightsquigarrow \cdots. \end{aligned}$$

There are also terms that grow without bound when we reduce them, for example:

$$\begin{aligned} (\lambda x.x\,x\,x)\,(\lambda x.x\,x\,x) &\rightsquigarrow (\lambda x.x\,x\,x)\,(\lambda x.x\,x\,x)\,(\lambda x.x\,x\,x) \\ &\rightsquigarrow (\lambda x.x\,x\,x)\,(\lambda x.x\,x\,x)\,(\lambda x.x\,x\,x)\,(\lambda x.x\,x\,x) \\ &\rightsquigarrow \cdots. \end{aligned}$$

Here is an example of a term that weakly normalizes but doesn't strongly normalize:

$$(\lambda x.\lambda y.x)\,(\lambda z.z)\,\omega.$$

This terms applies the first projection function to to arguments. If we immediately reduce the application of the projection, the argument ω disappear and we are left with the identity function, which is a normal form:

$$(\lambda x.\lambda y.x)\,(\lambda z.z)\,\omega \rightsquigarrow \lambda z.z.$$

However, if we try to reduce the redex inside the second argument, we don't make any progress and we could continue reducing it forever.