Functional Languages Notes

Riccardo Cappi

January 2024

0.1 Disclaimer

These are just my notes that I used to prepare for the exam. So, probably, there will be both spelling and conceptual errors. Feel free to contact me at riccardo.cappi@studenti.unipd.it if you find any errors. This is the github repo where you can find the latex files of the notes: `https://github.com/riccardocappi/Computer-Science-notes`

Unfortunately the notes for this course are **incomplete** :(

Contents

Chapter 1

Lec 09 - Higher-order polymorphic functions in F#

1.1 Higher-order functions

In functional languages, functions are actually values. This means that, like all other values of the language, they can be:

- memorized;
- passed as a parameters to a function;
- returned by a function;

An **higher-order function** is a function that takes one or more functions as parameters and/or returns a function as a result
Example of higher-order functions for lists:

map function:

```
let rec map (f, l) =
    match l with
    | [] -> []
    | x::xs -> f x :: map (f, xs)
```

This function has type ('a → 'b) * 'a list → 'b list. In fact, it takes as input a function from 'a to 'b and a list of elements of type 'a, and it returns a list of elements of type 'b. Basically this function transforms each element of a given list into another element according to a function **f**.

```
let string_to_ints = map( (fun s -> String.length s), ["Hello"; "World"])
```

The code above transforms each string of the list *["Hello";"World"]* into an int that is the size of the string (by using the function belonging to the String module of F# standard library String.length). The resulting list string_to_ints is the following:

```
val string_to_ints : int list = [5; 5]
```

Note that the previous syntax can be replaced by the following one:

```
let string_to_ints = map(String.length, ["Hello"; "World"])
```

And why is that? Let's reason about types.
Consider the lambda

$$\lambda x.fx$$

Let's say that x has type τ ($x : \tau$) and $f : \tau \to \sigma$. The lambda $\lambda \underbrace{x}_{\tau} . \underbrace{fx}_{\sigma}$ has the same type of f ($\tau \to \sigma$), so we can say that $\lambda x.fx \equiv f$.
Given a function f, adding a parameter and forwarding it to the function is called **η-expansion**

$$f$$

$$\lambda x.fx$$

1.1.1 Currying

How can we define functions with two parameters ? We have two different options:

- `let f1 = fun(x,y) -> x + y`
- `let f2 = fun x -> (fun y -> x + y)`

With the first method we are just defining a function that takes two integer numbers as parameters and produces the sum of the two. With the second one we are nesting lambdas. In fact, **f2** is a function that takes an integer number as input and *returns* another function that takes an integer as input and produces the sum of the two numbers. From the point of view of types:

```
val f1 : x:int * y:int -> int
val f2 : x:int -> y:int -> int
```

Defining functions as expressions formed by nested lambdas is called **currying**.
The syntax to call functions **f1** and **f2** defined above is the following

```
let call_uncurried = f1(3,4)
let call_curried = f2 3 4

val call_uncurried : int = 7
val call_curried : int = 7
```

As you can see, when we call **f2**, parameters are not between round brackets but they are separated by a space.
Currying is useful because it allows you to perform **partial evaluation**. Let's consider an example:

```
let add_something_to_3 = f2 3
let ten = add_something_to_3 7
let five = add_something_to_3 2
let seven = add_something_to_3 4

val add_something_to_3 : (int -> int)
val ten : int = 10
val five : int = 5
val seven : int = 7
```

With the first expression we are *partially* applying **f2**. Basically, **add_something_to_3** is a function that takes an integer as parameter and adds that number to 3.
Now let's go back to the **map** function mentioned before and let's change the syntax in order to define it with the currying syntax:

```
let rec map f l =
    match l with
    | [] -> []
    | x :: xs -> f x :: map f xs

val map : f:('a -> 'b) -> l:'a list -> 'b list
```

The only thing that is changed is how you write parameters (now separated by a space).

```
let map_string_length = map(String.length)
let map_int_squares = map(fun x-> x*x)
let map_int_times_2 = map(fun x-> x*2)

map_string_length(["Hello"; "World"])
//Result: val it : int list = [5; 5]

map_string_length(["Hello";"I'm";"Riccardo"])
//Result: val it : int list = [5; 3; 8]

map_int_squares([1;2;3;4;5])
//Result: val it : int list = [1; 4; 9; 16; 25]

map_int_times_2([5;4;3;2;1])
//Result: val it : int list = [10; 8; 6; 4; 2]
```

In the code above we defined the *behavior* of three functions (**map_string_length**, **map_int_squares**, **map_int_times_2**) by partially applying the function **map**.

1.1.2 Polymorphism and Currying

Let's consider the following expression:

```
let map_id = map id
```

where **id** is the identity function and it is **polymorphic** ('a $\rightarrow$ 'a). If you write this line of code, the F# compiler will give you this error: *Value restriction. The value 'map_id' has been inferred to have generic type val map_id : ('_a list $\rightarrow$ '_a list) Either make the arguments to 'map_id' explicit or, if you do not intend for it to be generic, add a type annotation.* This is because F# doesn't allow you to produce new polymorphic functions **from** polymorphic functions unless you explicitly **η-expand** it or put an explicit type annotation. F# enforces this restriction to avoid common programming errors when dealing with higher-order polymorphic functions.
The solution is to write **map_id** function as follows (**η-expansion**):

```
let map_id = fun x -> map id x
//Type: val map_id : x:'a list -> 'a list
```

That is the same of:

```
let map_id x = map id x
```

1.1.3 Other examples of higher-order functions

Filter function:

```
let rec filter f l =
    match l with
    | [] -> []
    |x :: xs -> if f x then x :: filter f xs else filter f xs

//Type: val filter : f:('a -> bool) -> l:'a list -> 'a list
```

This function filters a list of elements of type 'a, according to a predicate **f** (function that returns a boolean), and produces a new list of elements of type 'a composed by the elements that satisfy that predicate.

```
let ex1 = filter(fun x -> x > 25)[1 .. 30]

// Result: val ex1 : int list = [26; 27; 28; 29; 30]
```

Sum function:

```
let rec sum_ints l =
    match l with
    | [] -> 0
    | x :: xs -> x + sum_ints xs

let s1 = sum_ints [1;2;3]
//Result: val s1 : int = 6
```

This function performs the sum of all the elements of a list of integers. Obviously it is monomorphic, because it only works with integers lists, but we can make it polymorphic in the following way:

```
let rec sum (+) zero l =
    match l with
    | [] -> zero
    | x :: xs -> x + sum (+) zero xs

//Type: val sum : op_Addition:('a -> 'b -> 'b) -> zero:'b -> l:'a list -> 'b

let s2 = sum (+) 0.0 [1.0;2.22;3.65;4.45]
//Result: val s2 : float = 11.32
```

The parameters of this function are:

- A function named (+) that takes a 'a element and a 'b element and produces a 'b element. Note that, within the scope of the function, we use this operator in **infixed** form.
- The *zero-case* for our specific type (int $\rightarrow$ 0, floats $\rightarrow$ 0.0, ...).
- The list of elements that we want to sum.

Note that this function can be used for summing ints, floats, Strings but even booleans:

```
let s3 = sum (fun a b -> a || b) false [false; true; true; false]
//Result: val s3 : bool = true
```

With this expression we are performing **or** through the specified list of booleans. So actually our function doesn't perform only sum, but it performs anything we specify as the (+) parameter (as long as it is type-consistent).

Chapter 2

Lec 10 - Fold function

2.1 Iteration

In imperative languages (e.g. Java) it exists the **foreach** statement that permits you to iterate over a sequence.

```
for (MyType x: e){
    // do something
}
```

This Java code iterates over the sequence **e**, that must be a sub-type of $Iterable < MyType >$, and executes the block of code specified within the curly brackets.

In functional languages iteration works in a different way because there is no foreach statement. Let's define the following **iter** function in F#.

```
let rec iter f l=
    match l with
    |[] -> ()
    |x::xs -> f x; iter f xs

iter(fun n -> printf "%d\n" n)[1..20]
```

It applies a function **f** (in this case a lambda that prints an integer) over each element of a list **l** **ignoring** the result of the computation. Basically, the code above prints all the numbers in the list [1..20].

In F# if you don't bind the result it means that the expression has no result. But how can we represent *"don't having a result"* in a language that must always computes something? Let's look at the type of the function:

```
val iter : f:('a -> unit) -> l:'a list -> unit
```

The type inference expects that **f** takes something and returns nothing, and as you can see, the function **f** takes a parameter of type 'a (polymorphic) and returns a new type called **unit**. It is defined as follows:

```
type unit = ()
```

The type called *unit* has only one data-constructor with **name** () and it represents the notion of nothingness. Note that it is different from the keyword *void*, for example, in Java. In fact, void is not a type, you can't declare a variable of type void, and this is because in imperative languages you don't need to always compute something. In functional languages, instead, you can't omit a result. If we look at the printf function used in the previous code, it takes as parameters a format string, a number of additional arguments depending on the format string, and returns *unit*.

The iter function we defined returns *unit* when it reaches the end of the list. So, since the *unit* type has only one data-constructor, we can pattern-match the result of iter using the **let** keyword as follows:

```
let () = iter(fun n -> printf "%d\n" n)[1..20]
```

You can use the iter function, already defined in the F# standard library, to perform operations on lists that don't need a result.

2.1.1 Fold function

The **fold** function traverses a list applying a function **f** to each element and carrying over an **accumulator**[1] that is forwarded to the next element.

The fold function can be defined as **fold-left** or **fold-right**.

```
let rec foldR f z l =
    match l with
    | [] -> z
    | x::xs -> f (foldR f z xs) x

let rec foldL f z l =
    match l with
    | [] -> z
    | x :: xs -> foldL f (f z x) xs
```

Let's start with the fold-left. It is written in curried form and it takes 3 parameters: **f, z** and **l**. It traverses the list **l** using recursion and, at each recursion step, it updates the value of the accumulator **z** by applying the function **f** on the current element of the list (**x**) and forward it to the next step. This function can be seen as a polymorphic generalization of the foreach statement. The programmer can specify any **binary** function that processes each element and propagates an accumulator as the second argument.

```
let r1 = foldL (+) 0 [1..10]
```

The code above uses the fold-left function to sum the elements of a list of 10 numbers; starting from the initial state of the accumulator (0), it computes the sum between elements storing the result of the previous step in the accumulator that is forwarded to the next recursion step.

The fold-right version does the same thing, but in reverse order. It first recursively goes at the end of the list. Then, ascending from the recursion, it applies the function **f** in a reverse order.

Note that fold-left and fold-right can produce different results on the same list if the function **f** is not commutative.

[1]an accumulator is something that stores the result of the previous computation and is forwarded to the next one

```
let s1 = foldL (+) "" ["a"; "b"; "c"]
let s2 = foldR (+) "" ["a"; "b"; "c"]

//result
val s1 : string = "abc"
val s2 : string = "cba"
```

The (+) operator applied on a list of strings concatenates the elements, so the results are one the reverse of the other.

Chapter 3

Lec 11 - Higher-order functions using fold

3.1 Map and filter functions using fold-left

The fold function can be seen as a polymorphic generalization of the foreach statement. So, we can try to write the **map** function using fold-left.

The fold-left function is defined as follows:

```
let rec foldL f z l =
match l with
| [] -> z
| x :: xs -> foldL f (f z x) xs

//Type : val foldL : f:('a -> 'b -> 'a) -> z:'a -> l:'b list -> 'a
```

The map function creates a new list transforming each element of the input list into another element according to a function f.

An implementation of the map function using fold-left can be the follow:

```
let map f l = foldL (fun z x -> z @ [f x]) [] l
map (fun x -> x*x) [1..5]

//Result: int list = [1; 4; 9; 16; 25]

let map2 f l = foldL (fun z x -> f x :: z) [] l
map2 (fun x -> x*x)[1..5]

//Result: int list = [25; 16; 9; 4; 1]
```

The code above defines two implementations of the map function. The behavior of the two functions is to map each number to its respective square. The fold-left function consents to store the squares list in the accumulator and to forward it to the next recursive step. At the end, the accumulator will contain the mapped list to return. As you can see, the results are slightly different (one list is the reverse of the other). This is due to how the list to pass as accumulator is generated.

- In the first case, at each recursive step, the accumulator is forwarded according to the following expression:

  ```
  z @ [f x]
  ```

  The @ operator takes two lists and produces a new list, basically it *concatenates* two lists. So, in this case we can concatenate the result of the application of **f** on the current element **x** (using the temporary list [f x]) to the content of the accumulator. Basically, the accumulator is forwarded to the next step by "*adding*"[1] the mapped element [f x] at the **end** of the list contained in the accumulator at the previous step. By doing this, the resulting list will be in the same order as the input list.

  ```
  [] + [1] = [1]
  [1] + [4] = [1; 4]
  [1; 4] + [9] = [1; 4; 9]
  [1; 4; 9] + [16] = [1; 4; 9; 16]
  [1; 4; 9; 16] + [25] = [1; 4; 9; 16; 25]
  ```

- In the second function, instead, the accumulator is forwarded according to this expression:

  ```
  f x :: z
  ```

  The :: operator takes an **element** and a list and produces a list. It forwards the accumulator by creating a new list with the mapped element f x as head and the accumulator of the previous step as tail. So, since we are traversing the list *from left to right*, the resulting list will be in the opposite order as the input list.

  ```
  1 + [] = [1]
  4 + [1] = [4; 1]
  9 + [4; 1] = [9; 4; 1]
  16 + [9; 4; 1] = [16; 9; 4; 1]
  25 + [16; 9; 4; 1] = [25; 16; 9; 4; 1]
  ```

Note that we can't pattern-match on the @ operator because it is just a function and not a data constructor.

Implementing the map function using fold-right instead of fold-left would lead to the opposite result (the resulting list using :: operator will be in the *correct* order and the resulting list using @ will be reversed).

The same approach can be followed in order to write the **filter** function.

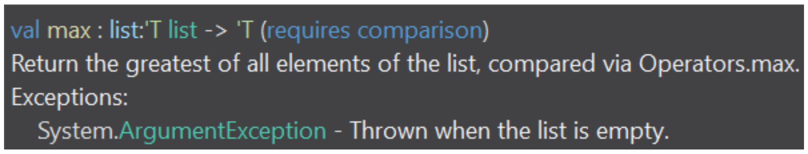
```
let filter f l = foldL (fun z x -> if f x then z @ [x] else z) [] l
filter (fun x -> x > 10) [1..20]

//Result: [11; 12; 13; 14; 15; 16; 17; 18; 19; 20]

let filter2 f l = foldL (fun z x -> if f x then x :: z else z) [] l
filter2 (fun x -> x > 10) [1..20]

//Result: [20; 19; 18; 17; 16; 15; 14; 13; 12; 11]
```

[1]Note that is not an actual adding operation since we are creating a new list, at each step, with the mapped element in last position

All the methods presented until now are provided in the List module of the F# standard library.

3.2 Max function

The **max** function defined in the List module of the F# standard library has the following type



```
val max : list:'T list -> 'T (requires comparison)
Return the greatest of all elements of the list, compared via Operators.max.
Exceptions:
   System.ArgumentException - Thrown when the list is empty.
```

The type ’T requires a **comparison** constraint. It means that the max function works correctly as long as the type ’T implements the < operator (less than). F# consents you to add constraints to types, but if we don’t want to use this advanced feature and we want to write the max function from scratch, we need to pass an additional function as parameter that performs the comparison. Also this kind of function is already defined in the List module and is called **maxBy**.

An implementation of the maxBy function can be the following:

```
let rec max_by cmp l =
   match l with
   | [] -> raise (Failure "message")
   | [x] -> x
   | x::xs -> let m = max_by cmp xs in if cmp x m then m else x

//Type: val max_by : cmp:('a -> 'a -> bool) -> l:'a list -> 'a

max_by (<) [1; -33; 55; 1]

//Result: val it : int = 55
```

If the list is empty it raises an exception. Otherwise, it recursively goes at the end of the list and select the last element as maximum. Ascending from the recursion it compares each element with the current maximum. If the current element **x** is greater than the maximum, this element becomes the maximum. At the end, the function returns the max value of the input list. Writing the function this way avoids adding an extra parameter in which to store the current maximum value.

The **in** keyword separates what you want to do after a binding. If you write the expression after the binding on a new line you don’t need this keyword.

3.2.1 MaxBy function using fold-left

MaxBy function can be defined also using the fold-left function by storing in the accumulator the current maximum value. The problem is how to set the initial value of the accumulator, because the initial max value is **unknown**. To express the concept of unknown value in F# we can use **option type**

```
type 'a option = None | Some of 'a
```

The type ’a can be either None or Something of type ’a[2].

[2] As usual, this type is already defined in the F# standard library

Now we can implement maxBy using fold-left as follows:

```
let max_by_fold cmp l=
   let f m x =
      match m with
      | None -> Some x
      | Some y -> if cmp x y then Some y else Some x
   foldL f None l

//Type: val max_by_fold : cmp:('a -> 'a -> bool) -> l:'a list -> 'a option

max_by_fold (<) [1; -33; 55; 100]

//Result: Some 100
```

This implementation works by defining a **nested** function **f** that performs the comparison between the element **x** and the current max value (taking into account the fact that the type of the accumulator is *'a option*). Then this function is passed to foldL with the initial state of the accumulator equals to None.

Chapter 4

Lec 12 - Higher-order functions on trees

4.1 Binary Trees

In F# a binary tree is defined as follows:

```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
```

As you can see, data are only in the leaves and nodes don't have any information attached.

If we want to add information also in the nodes, we can use the following definition:

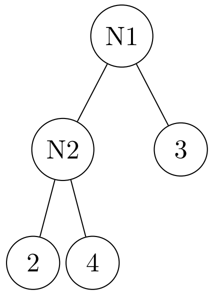
```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree * 'a
```

4.2 Map function

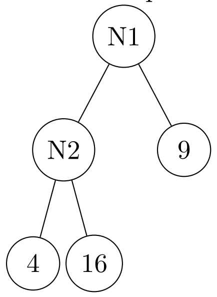
Let's define the map function for binary trees considering the definition with data only in the leaves.

```
let rec map_tree f t =
   match t with
      | Leaf x -> Leaf(f x)
      | Node (l,r) -> Node (map_tree f l, map_tree f r)

let tree = Node( Node( Leaf(2), Leaf(4)),Leaf(3))
let m = map_tree (fun x -> x*x) tree

// val map_tree : f:('a -> 'b) -> t:'a tree -> 'b tree
// val tree : int tree = Node (Node (Leaf 2, Leaf 4), Leaf 3)
// val m : int tree = Node (Node (Leaf 4, Leaf 16), Leaf 9)
```

The original tree is:



And the map function produces the following new tree:



Basically, each value is mapped to its respective square.

This function can be defined in many different ways:

```
let rec map_tree f t =
    match t with
    | Leaf x -> Leaf (f x)
    | Node (l, r) ->
        let m = map_tree f // val m: ('a tree -> 'b tree)
        Node(m l, m r)
```

In this case, instead of recursively call map_tree every time we reach a node, we bind *map_tree f* to m, using the **let** keyword, and then we use m to recursively build the new tree.

We can also η**-expand** this binding by adding a parameter t:

```
let rec map_tree f t =
    match t with
    | Leaf x -> Leaf (f x)
    | Node (l, r) ->
        let m t = map_tree f t //val m: ('a tree -> 'b tree)
        Node(m l, m r)
```

Note that the parameter t is a **new** identifier, and it is not the same parameter written in the definition of the function. We could have called it in any way. Redefining variables having the same name is called **shadowing**. It allows the programmer to *eliminate* the old name locally.

All these three definitions behave exactly the same. The definitions using the let binding are not so useful because they just add a function that is used only once. Also, the binding is performed every time we reach a Node, that is not very efficient. In order to solve this inefficiency, the compiler performs the optimization process called **Lambda uplifting**

4.2.1 Lambda uplifting

In order to understand Lambda uplifting, we need to define the map_tree function using a different curried syntax.

Writing the function in this way:

```
let rec map_tree f t
```

is a **syntactic sugar** of:

```
let rec map_tree = fun f -> fun t -> ...
```

So, we can **uplift** the binding of m as follows:

```
let rec map_tree =
    fun f ->
        let m t = map_tree f t
        in fun t ->
            match t with
            | Leaf x -> Leaf (f x)
            | Node (l, r) -> Node (m l, m r)
```

In this case, the binding is performed only the first time the function is called, and every recursive call will not do the binding again because the *internal* lambda has the function m in its scope.

This process doesn't have to be done by the programmer, the compiler automatically do it when we define local lambdas in order to perform the least number of bindings.

4.3 Sum function

Let's define an higher-order function that produces the sum of the elements of a given tree:

```
let rec sum_tree (+) t =
    match t with
    | Leaf x -> x
    | Node (l, r) -> sum_tree (+) l + sum_tree (+) r

//Type: val sum_tree : op_Addition:('a -> 'a -> 'a) -> t:'a tree -> 'a
```

Note that (+) is a parameter with a "*special*" name and it has the following type:

```
'a -> 'a -> 'a
```

It is an operator that performs the sum between the left sub-tree and the right sub-tree. This operator is defined in curried form and it takes 2 parameters of type 'a and produces a 'a element. Within the scope of the sum_tree function, we must use the (+) parameter in **infix** form rather than prefix form. This is because any sequence of non alpha-numeric characters is treated by F# as an infix name. The reason why we use the name (+) rather than other names is because we want to shadow the **global** + operator defined for integers. We can pass as this parameter each function that satisfies its types constraints.

For example, writing the following code:

```
let tree = Node( Node( Leaf(2), Leaf(4)),Leaf(3))

//Using the global + operator to perform the sum
let s1 = sum_tree (+) tree

// val s1 : int = 9
```

is the same of writing:

```
let tree = Node( Node( Leaf(2), Leaf(4)),Leaf(3))
let plus a b = a + b
//Using the custom function named 'plus' to perform the sum
let s1 = sum_tree plus tree

//val s1 : int = 9
```

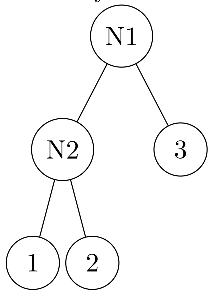
Chapter 5

Lec 13 - Higher-order functions for trees II

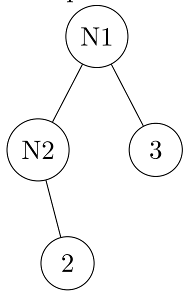
5.1 Filter function for trees

The **filter** function takes a predicate and a tree and produces a new tree with the elements that satisfy the predicate.

Let's say we have the following tree:



and as predicate the lambda $\lambda x.x > 1$. The filter function should produce the tree:



We defined our tree type as follows:

```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
```

As you can see, it doesn't support trees with only one child, but we can only represent trees with two children. So, in order to implement the filter function, we need to change the definition and make it more flexible. We can do it in different ways:

```
type 'a tree =
    | Leaf of 'a
    | Node of 'a tree * 'a tree
    | OneChild of 'a tree
```

The code above defines a union type with an additional data-constructor which represent a node with only one child. However, we wouldn't be able to distinguish between left-child and right-child. So, in order to obtain this distinction, we can specify two equivalent data-constructors with different names.

```
type 'a tree =
    | Leaf of 'a
    | Node of 'a tree * 'a tree
    | LeftChild of 'a tree
    | RightChild of 'a tree
```

We can get the same result by using **option** type and defining just 2 data-constructor:

```
type 'a tree =
    |Leaf of 'a option
    |Node of 'a tree * 'a tree
```

In this case, leaves may or may not contain an information. We express the concept of having only one child by defining binary trees that can have **empty** leaves.

Let's define the filter function according to this tree type definition:

```
let rec filter_tree p t =
    match t with
    | Leaf (Some x) -> if p x then Leaf (Some x) else Leaf None
    | Leaf None -> Leaf None
    | Node (l, r) -> Node (filter_tree p l, filter_tree p r)

let N = Node
let L x = Leaf (Some x)
let tree = N(N(L 1, L 2), L 3)
let t1 = filter_tree (fun x -> x > 1) tree

// val filter_tree : p:('a -> bool) -> t:'a tree -> 'a tree
// val t1 : int tree = Node (Node (Leaf None, Leaf (Some 2)), Leaf (Some 3))
```

As you can see, t1 is a tree with an empty leaf that represents the tree presented before.

5.1.1 Map and sum functions

Let's change the map_tree and sum_tree functions according to the new tree type definition:

```
let rec map_tree f =
    let R t = map_tree f t
    in fun t ->
        match t with
        | Leaf None -> Leaf None
        | Leaf (Some x) -> Leaf (Some(f x))
        | Node (l, r) -> Node (R l, R r)
```

```
let t2 = map_tree (fun x -> x>1) tree

(*val map_tree : f:('a -> 'b) -> ('a tree -> 'b tree)
  val t2 : bool tree =
    Node (Node (Leaf (Some false), Leaf (Some true)), Leaf (Some true))*)


let rec sum_tree (+) zero t =
    match t with
    | Leaf (Some x) -> x
    | Leaf None -> zero
    | Node (l, r) -> (sum_tree (+) zero l) + (sum_tree (+) zero r)
```

Note that we added the base case of sum (parameter 'zero') that is returned when the function reaches an empty leaf.

5.1.2 Union types and pattern-matching in Java

In order to implement the same thing in an object oriented programming language such as Java, we have to simulate union types using objects and inheritance.

Let's define the map_tree function in Java:

```
public abstract class Tree<T> {
    public abstract <S> Tree<S> map(Function<T, S> f);
}
public class Leaf<T> extends Tree<T> {
    @Nullable
    private T data;
    public Leaf(T data) {
        this.data = data;
    }
    @Override
    public <S> Tree<S> map(Function<T, S> f) {
        return new Leaf<S>(data != null ? f.apply(this.data) : null);
    }
}
public class Node<T> extends Tree<T> {
    private Tree<T> left, right;
    public Node(Tree<T> l, Tree<T> r) {
        this.leaf = l;
        this.right = r;
    }
    @Override
    public <S> Tree<S> map(Function<T, S> f) {
        return new Node<S>(left.map(f), right.map(f));
    }
}
```

Basically, the map_tree higher-order function becomes a method of the super-class *tree*, and we simulate pattern-matching by splitting each implementation of map_tree in the sub-classes (*Leaf* and *Node*). This solution is called **visitor pattern**. As you can see, the Java code is more complex and long than the F#

code.

5.2 Fold function for trees

```
let rec fold_tree f z t =
   match t with
   | Leaf (Some x) -> f z x
   | Leaf None -> z
   | Node (l,r) ->
      let z' = fold_tree f z l
      fold_tree f z' r
```

- When the function reaches a leaf that contains an information, it computes the new state of the accumulator by applying the function f. If the leaf is empty, it just returns the accumulator.
- When the function reaches a node, it forwards the accumulator computed on the left sub-tree to the right sub-tree.

Note that for lists there are two implementations of the fold function: fold-left and fold-right.

- **fold-left:** First computes the accumulator and then performs the recursion step.
- **fold-right:** Recursively goes to the end of the list and *updates* the accumulator in ascension.

Since our trees definition doesn't support information in the nodes, we can't distinguish between fold-left and fold-right.

Chapter 6

Lec 14 - Higher-order functions for trees III

6.1 Trees type definition

Until now we used binary trees that support data only in the leaves. Let's change the type definition in order to support information attached to nodes. We can do this in many different ways:

```
type 'a tree =
    | Node of 'a * 'a tree * 'a tree
    | Leaf of 'a
```

In this case, a node is composed by an element of type 'a (which is the information attached to it) and the two sub-trees. However, with this definition we wouldn't be able to represent dead branches. In fact, each node must always have two sub-trees that can be either another node or a leaf.

```
type 'a tree =
    | Node of 'a * 'a tree * 'a tree
    | Leaf of 'a option
```

With this implementation we can represent a dead branch defining a Leaf of None, but we could even define nodes with two empty leaves. So, this definition allows the programmer to write things that are not in their most simplified form.

In order to prevent this, we can do the following:

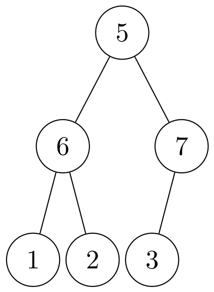
```
type 'a tree =
    | Node of 'a * 'a tree option * 'a tree option
```

Now the **Node** data-constructor is a triple composed by:

- An element of type 'a (information in the node)
- Two 'a tree option elements which represent the left and right sub-trees

When both the sub-trees are None, we are representing a leaf.

Let's write in F# the following tree using this definition:



```
let Leaf x = Some(Node(x, None, None))
let tree = Node(5,
           Some(Node(6, Leaf 1, Leaf 2)),
           Some(Node(7, Leaf 3, None)))
```

Note that **Leaf x** is a function, not a data-constructor. It is just a shorter way to define leaves. The difference between functions and data-constructors is that functions can be used to **create**, but not for deconstructing (they can't be used to pattern-match).

We can write trees even in a shorter form by defining the following function:

```
let SNode (x, t1, t2) = Some (Node (x, t1, t2))
let tree = Node (5,
          SNode (6, Leaf 1, Leaf 2),
          SNode (7, Leaf 3, None)
          )
```

How can we pattern-match this type definition ? We have to specify all possible combinations of node types.

```
let rec pretty_tree t =
   match t with
   | Node(x, None, None) -> sprintf "(. %O .)" x
   | Node(x, Some l, Some r) -> sprintf "(%s %O %s)" (pretty_tree l) x (pretty_tree r)
   | Node(x, Some l, None) -> sprintf "(%s %O %s)" (pretty_tree l) x "."
   | Node (x, None, Some r) -> sprintf "(%s %O %s)" "." x (pretty_tree r)

let st = pretty_tree tree

// val st : string = "(((. 1 .) 6 (. 2 .)) 5 ((. 3 .) 7 .))"
```

The function above is a **pretty_print** function for trees, which is a function that produces a string that represents the given tree. Instead of specify all the possibilities in the pattern-match, we can simplify things by defining an additional function.

```
let pretty_opt f o =
   match o with
   | None -> "."
   | Some x -> f x

let rec pretty_tree t =
   match t with
   | Node (x, lo, ro) ->
```

```
        let l = pretty_opt pretty_tree lo
        let r = pretty_opt pretty_tree ro
        sprintf "(%s %O %s)" l x r

let st = pretty_tree tree

// val st : string = "(((. 1 .) 6 (. 2 .)) 5 ((. 3 .) 7 .))"
```

6.2 Record types

In F# we can define the so called **record types** using the following syntax:

```
type 'a tree = {data : 'a;
                left : 'a tree option;
                right : 'a tree option}
```

Records are series of *Label : type* separated by a semicolon.

Chapter 7

Lec 15-16 Part I - Type inference in F#

7.1 Simple type inference algorithm

The type inference algorithm automatically understands the type of an expression in a formal language. Let's consider the following F# functions with their respective types:

```
let f a b c = if a then b c else a
//val f : a:bool -> b:('a -> bool) -> c:'a -> bool

let g x y z = (x, y, z)
//val g : x:'a -> y:'b -> z:'c -> 'a * 'b * 'c

let h x y z = z y x
//val h : x:'a -> y:'b -> z:('b -> 'a -> 'c) -> 'c
```

A simple type inference high-level algorithm could be the following:

1. Bind unknown types to the parameters ('a, 'b, 'c, ...). We will call these unknown types **type variables**.
2. **Substitute** each type variable with a concrete type (if possible) according to what is written in the body of the function. Once a concrete type (int, float, bool, ...) is fixed, it doesn't change anymore. If we need to substitute more that one concrete type it means there is an error in the function.

For example, if we change the definition of f as follows:

```
let f a b c = if a then b c else a+1
```

it produces an error because *a* was inferred as bool but is used as an int in the *else* branch of the *if-then-else* statement.

```
let f a b c = if a+1 > 3 then b c else a
```

Since the *if-then-else* statement must return the same type in both the branches, the type of the new definition of f is:

```
int -> ('a -> int) -> 'a -> int
```

- *a* has type int
- Also the function *b* produces an int

In the body of the function g there are no **constraints** that permit the type inference to substitute any concrete type, so it will remain a polymorphic function with respect to all the parameters.

Let's insert a constraint in the body of the function g.

```
let g x y z = (x-1, y, z)
```

With this change, the compiler understands that *x* must be an integer (due to the constraint x-1), while y and z remain unknown (polymorphic). Polymorphism arises from not knowing any constraint on a type.

Now we can define a more detailed pipeline of the type inference algorithm:

1. Bind unknown types to parameters (type variables)
2. Create a list of constraints according to the body of the function
3. Check the list of constraints:
   (a) Eliminate trivial constraints
   (b) If two or more constraints for the same type variable are found, produce an error
4. Substitute to each type variable a concrete type, according to the constraints list, to find the final type.

Let's make an example:

```
let j x = if x then x else x
```

1. We start by binding unknown types:

   ```
   j: 'a -> 'b
   ```

2. The *if-then-else* statement requires two constraints:
   - The guard of the *if* must be a bool
   - The two branches must produce the same type

   So the list of constraints becomes:

   ```
   'a -> bool
   'a -> 'a
   'b -> 'a
   ```

   By eliminating trivial constraints the list becomes:

   ```
   'a -> bool
   'b -> 'a
   ```

   Then, we simplify the constraints list:

```
'a -> bool
'b -> bool
```

3. Since there are not multiple constraints for the same type variable, we can perform the substitution. The type of j becomes:

   ```
   j: bool -> bool
   ```

   The substitution can be seen as a function from type variables to types:

$$\Theta : \alpha \to \tau$$

Note that if we had written the following expression:

```
let j x = if x then x else 2
```

the constraints list would have been:

```
'a -> bool
'a -> int
'b -> int
```

So, the type inference algorithm would have produced an error since there are multiple constraints for the type variable 'a.

Chapter 8

Lec 15-16 Part II - Type checking rule

8.1 Inference logic rule

$$\frac{\Gamma, (x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

The formula above, in the world of logic, is called **inference logic rule**. Basically, it is an implication from "*up*" to "*down*". It means that the formula above the horizontal line implies the formula below.

In order to understand this rule, we need to define its terms.

8.1.1 Expressions

e is an **expression** defined through a **BNF** grammar:

$\langle e \rangle ::= \langle x \rangle$
  $|$ $\langle \lambda x.e \rangle$
  $|$ $\langle e\ e \rangle$
  $|$ $\langle Let\ x = e\ in\ e \rangle$

An expression e can either be:

- A variable name
- A lambda
- An application
- A let binding

Let's consider the following expression in F#:

```
let m = (1+2+3) 4
```

The right part of the let binding is correct from the point of view of the syntax (it's an application), but is wrong from the point of view of types. So, a compiler can be seen as a pipeline:

1. First checks the syntax
2. Then understands types (type inference)

8.1.2 Types

A type τ is defined as a syntactic entity through a BNF grammar:

$\langle\tau\rangle ::= \langle c\rangle$
$\quad | \quad \langle\tau \to \tau\ \rangle$
$\quad | \quad \alpha$

Where c is a constant type name (e.g. int, float, bool, ...) and α is a **generic** type variable.

8.1.3 The environment Γ

Γ is the environment (scope) and is defined as follows:

$\langle\Gamma\rangle ::= \emptyset$
$\quad | \quad \langle\Gamma, (x : \tau)\ \rangle$

Let's consider the following F# code:

```
let f a b c = if a then b c else a

let myfun x y=
    let f z = f x y x
    f 3
```

Note that when we bind *f* inside *myfun*, since it is not specified the **rec** keyword, we are referring to the definition of *f* written above *myfun*. So, in order to analyze the type of the nested function *f* defined inside *myfun*, the compiler must have memorized the type of the function *f* defined above *myfun*. In F# the let binding adds an entry to the scope, but how can we represent this information ?

The simplest way for representing the scope is a sequence of pairs, where each pair is composed by an identifier and its type. Γ can be seen as a function from the set of identifiers X to types:

$$\Gamma : X \to \tau$$

8.1.4 Understanding the formula

Let's go back to the logic rule mentioned before:

$$\frac{\Gamma, (x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

- $\vdash$ means "*deduce*"
- $:$ means "*has type*"

so, a literal translation of $\Gamma \vdash e : \tau$ would be: "*In a scope* Γ*, we deduce that the expression* e *has type* τ".

The meaning of the whole formula is the following: Given a lambda $\lambda x.e$ in an environment Γ, if we extend the scope Γ by binding to the parameter x the type τ_1 and we deduce that the body e has type τ_2, then the lambda must have type $\tau_1 \to \tau_2$. Basically, it says that the domain of the lambda is the type of the parameter and the codomain is the type of the body.

This is a **type checking** rule because it says what must be true for an expression in order to be valid from the point of view of types, but it doesn't show how to infer types. Type inference shows how to automatically understands types, rather than just assuming them.

Let's make an example in F#:

```
let r = fun x -> x 23
```

The right part of the let binding is a lambda that has an application as body. The type inference understands that:

- the parameter x has type int $\rightarrow$ 'a
- the expression e, that is an application, has type 'a

Then, by applying the type checking rule, the whole lambda must have type (int $\rightarrow$ 'a) $\rightarrow$ 'a

It is useful to express the type checking algorithm through a series of logical implications. This is because we want to describe the algorithm in such a way that we can derive some mathematical properties. The most important property that we can derive from this definition is **soundness**.

Soundness means that given a program, if the types are correct then the program works (in the sense that run without crashing). Note that, in functional languages, every program is an expression made of multiple let bindings.

8.1.5 Type checking rule for the other expression terms

We defined the type-checking rule only for the lambda-term of expressions. Let's write the same rule with respect to application, variable name and let binding terms (according to the definition of expression we gave in section 8.1.1).

- **Variable name:**

  $$\frac{\begin{array}{c} x \in domain(\Gamma) \\ \Gamma(x) = \tau \end{array}}{\Gamma \vdash x : \tau}$$

  A variable x has type τ in the scope Γ if it is defined in the scope and if Γ applied to x produces τ (Γ is a function as we said previously).

- **Application:**

  $$\frac{\begin{array}{c} \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \\ \Gamma \vdash e_2 : \tau 1 \end{array}}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

- **Let binding:**

  $$\frac{\begin{array}{c} \Gamma \vdash e_1 : \tau_1 \\ \Gamma, (x : \tau_1) \vdash e_2 : \tau_2 \end{array}}{\Gamma \vdash let\, x = e_1\, in\, e_2 : \tau_2}$$

  Let's make an example in F#:

  ```
  let a=3 in let b = true in a
  ```

In this example $e_1 = 3$ and $e_2 =$ *let b = true in a* . The rule that type checks the binding *let a = 3* is $\Gamma \vdash e_1 : \tau_1$. In our example $\tau_1 = int \rightarrow e_1 : int$. Then, we bind *(a:int)* and we add this information to the original scope. By recursively repeating the type-checking for e_2 in the extended scope, we'll eventually have the environment Γ composed by: $\{(a : int), (b : int)\}$ and the variable a will be type-checked following the rule mentioned before.

Chapter 9

Lec 16 - Evaluation rule

9.1 Addition term

Before talking about the evaluation rules, let's add the + operator to our expressions grammar:

$\langle e\rangle ::= \langle x\rangle$
$\quad|\quad \langle \lambda x.e\rangle$
$\quad|\quad \langle e\ e\rangle$
$\quad|\quad \langle Let\ x = e\ in\ e\rangle$
$\quad|\quad \langle e + e\rangle$

The type-checking rule for the addition (only between integers) is the following:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : int\\ \Gamma \vdash e_2 : int\end{array}}{\Gamma \vdash e_1 + e_2 : int}$$

9.2 Evaluation rules

In the C language when a program is compiled the compiler checks syntax and types and produces an executable. Then, when we **run** the program, the executable is **evaluated**. Conversely, in python the program is just evaluated, without compiling it.

Let's talk about evaluation of our simple form of expressions.

In the world of evaluation the scope is defined as follows:

$\langle \Delta\rangle ::= \emptyset$
$\quad|\quad \Delta,(\mathrm{x} \rightarrow \mathrm{v})$

The scope Δ can either be:

- empty
- extended with a binding for a variable x which computes a **value** v

Γ binds variables to types, while Δ binds variables to values.

The evaluation rule for the + term previously defined is the following:

$$\frac{\begin{array}{c}\Delta \vdash e_1 \rightarrow v_1 \\ \Delta \vdash e_2 \rightarrow v_2 \quad v = v_1 + v_2\end{array}}{\Delta \vdash e_1 + e_2 \rightarrow v}$$

In a scope Δ, the expression $e_1 + e_2$ computes v if, in the same scope Δ, e_1 computes v_1, e_2 computes v_2 and $v = v_1 + v_2$ where the + operator is the one defined among integers. So, we first evaluate e_1, then e_2 and the result is the sum of the two. Note that v_1 and v_2 must be numbers.

9.2.1 Evaluation of lambdas

If we write a lambda in the F# interactive window it will be evaluated as follows:

```
fun x -> x + 1;;
val it : x:int -> int
```

The lambda can't be evaluated until it is called by passing an argument for the parameter x. So, in order to define an evaluation rule for lambdas, we need some kind of memory representation that stores the lambda **blocked** with the scope where it is defined (lexical scoping). This structure is called **closure**. Then, the evaluation rule is:

$$\frac{\emptyset}{\Delta \vdash \lambda x.e \rightarrow < \lambda x.e; \Delta >}$$

where $< \lambda x.e; \Delta >$ is a pair composed by the lambda and the original scope where the lambda is defined (closure).

9.2.2 Values definition

A value v is defined by the following BNF grammar:

$\langle v \rangle$ ::= lit
  | $\langle$ $\lambda x.e$; Δ $\rangle$

v can either be:

- a literal (4, true, false, 8.9, ...)
- a closure: it consists of a lambda expression and the scope where it is defined.

9.2.3 Evaluation rules of the other expression terms

- **Variables identifiers:**

$$\frac{\begin{array}{c}x \in domain(\Delta) \\ \Delta(x) = v\end{array}}{\Delta \vdash x \rightarrow v}$$

- **Let binding:**

$$\frac{\begin{array}{c}\Delta \vdash e_1 \rightarrow v_1 \\ \Delta, (x \rightarrow v_1) \vdash e_2 \rightarrow v_2\end{array}}{\Delta \vdash let\, x = e_1\, in\, e_2 \rightarrow v_2}$$

Let's consider the following F# code:

```
let a = 4 in a
//val it : int = 4
```

In this case $e_1 = 4$ and it is evaluated to 4. Then, the variable a is associated to the evaluated value ('4') and it is added to the scope $\Delta, (a \to 4)$. Finally, e_2 is evaluated: since $e_2 = a$ and $a \to 4$, the whole expression produces 4.

- **Application:**

$$\frac{\begin{array}{c}\Delta \vdash e_1 \to < \lambda x.e_0; \Delta_0 > \\ \Delta \vdash e_2 \to v_2 \\ \Delta_0, (x \to v_2) \vdash e_0 \to v_0\end{array}}{\Delta \vdash e_1\, e_2 \to v_0}$$

  - the left part (e_1) of an application must produce a closure
  - the right part (e_2) can produce anything

This is β-reduction, that is, how applications works at run-time in functional languages.

The whole application produces the evaluation of the body e_0 of the lambda defined in the original environment Δ_0, adding a binding for the parameter x to the value that e_2 computes.

```
let k = 1

let f = fun x -> x + k

let n = f 2
//val n : int = 3
```

When we call f:

- The evaluation of f produces a closure with the scope having $k \to 1$
- $e_2 = 2 \to 2$
- x is bound to 2 $(\Delta, (x \to 2))$. So, the application produces 3