Functional Programming (OCaml)

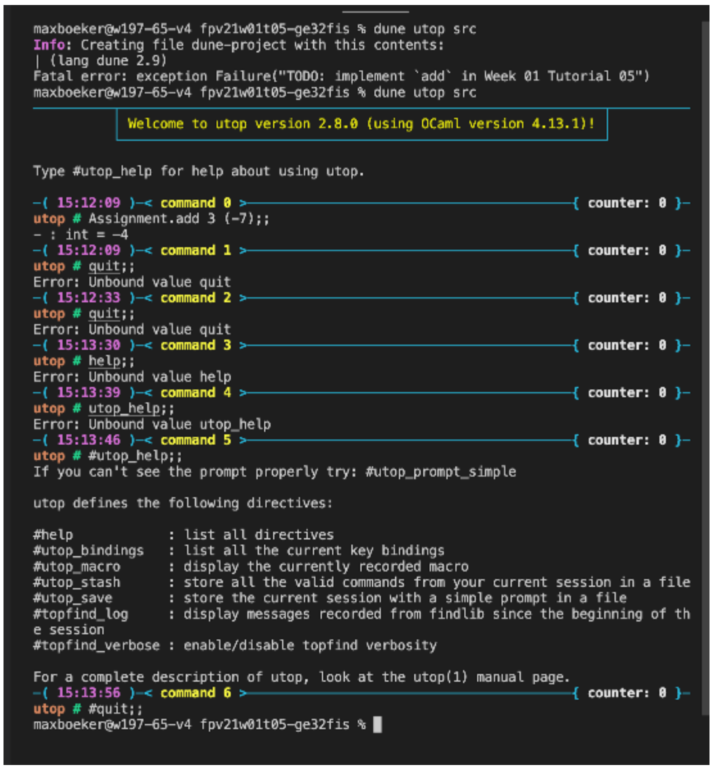
Funktionale Programmierung (Technische Universität München)

Functional Programming (OCaml)

- **Key Advantages of Functional Programming**
  - programs are strictly created through functions which are expressions that simply map inputs and outputs rather than sequences of code that update the state of an application (as do programming languages such as Python)
  - pure functions are better as its return value is the same for the same arguments (as they assure the outside program is not altered, interference from external environments), thus pure functions are also easier to test, concurrency is more easily kept safe, immutable variables lead to fewer side-effects
  - FP leads to fewer bugs because of its correctness and assurance for the correct type
  - functional code tends to have its state isolated, making it easier to comprehend
- **Why OCaml?**
  - extremly efficient implementation leads to very fast execution, as fast as C
  - OCaml is a strongly-typed language which means it detects type conflicts at compile-time rather than at run-time, and thus, reduce number of type related bugs significantly; plus is polymorphically typed meaning that listss can be mainpulated regardless of types that list contains; no type declarations
  - OCaml is predictable becaus of its simple compiler, runtime and general execution model, thus in practice fastest functional language; due to its strict type system global type inference is possible in almost all cases
  - programming languages affect reliability, security, and efficiency of the code one writes as well as how easy it is to read, refactor, and extend; also changes how u think, and influence the way u design software → OCaml facilitates these important characteristics of good software
  - OCaml is an elegant combination of a set of language features that have been developed over last 40 years ⇒ All of this makes OCaml a great choice for programmers who want to step up to a better programming language, and at the same time get practical work done.
    - *Garbage collection* for automatic memory management, now a feature of almost every modern, high-level language.
    - *First-class functions* that can be passed around like ordinary values, as seen in JavaScript, Common Lisp, and C#.
    - *Static type-checking* to increase performance and reduce the number of runtime errors, as found in Java and C#.
    - *Parametric polymorphism*, which enables the construction of abstractions that work across different data types, similar to generics in Java and C# and templates in C++.
    - Good support for *immutable programming*, *i.e.*, programming without making destructive updates to data structures. This is present in traditional functional languages like Scheme, and is also found in distributed, big-data frameworks like Hadoop.
    - *Type inference*, so you don't need to annotate every single variable in your program with its type. Instead, types are inferred based on how a value is used. Available in a limited form in C# with implicitly typed local variables, and in C++11 with its `auto` keyword.
    - *Algebraic data types* and *pattern matching* to define and manipulate complex data structures. Available in Scala and F#.

OCaml Introduction

- OCaml = programming language
- OPAM = source-based package manager
- OCaml's repl = read-eval-print-loop allowing interaction with OCaml system → compiles OCaml into executable
- **UTOP** provides improved interactive OCaml toplevel environment to work with OCaml
- first utop session including quitting: build and run code with `dune utop y` use `.` to open all in utop



```
maxboeker@w197-65-v4 fpv21w01t05-ge32fis % dune utop src
Info: Creating file dune-project with this contents:
| (lang dune 2.9)
Fatal error: exception Failure("TODO: implement `add` in Week 01 Tutorial 05")
maxboeker@w197-65-v4 fpv21w01t05-ge32fis % dune utop src
                Welcome to utop version 2.8.0 (using OCaml version 4.13.1)!

Type #utop_help for help about using utop.

-( 15:12:09 )-< command 0 >------------------------------------{ counter: 0 }-
utop # Assignment.add 3 (-7);;
- : int = -4
-( 15:12:09 )-< command 1 >------------------------------------{ counter: 0 }-
utop # quit;;
Error: Unbound value quit
-( 15:12:33 )-< command 2 >------------------------------------{ counter: 0 }-
utop # quit;;
Error: Unbound value quit
-( 15:13:30 )-< command 3 >------------------------------------{ counter: 0 }-
utop # help;;
Error: Unbound value help
-( 15:13:39 )-< command 4 >------------------------------------{ counter: 0 }-
utop # utop_help;;
Error: Unbound value utop_help
-( 15:13:46 )-< command 5 >------------------------------------{ counter: 0 }-
utop # #utop_help;;
If you can't see the prompt properly try: #utop_prompt_simple

utop defines the following directives:

#help            : list all directives
#utop_bindings   : list all the current key bindings
#utop_macro      : display the currently recorded macro
#utop_stash      : store all the valid commands from your current session in a file
#utop_save       : store the current session with a simple prompt in a file
#topfind_log     : display messages recorded from findlib since the beginning of th
e session
#topfind_verbose : enable/disable topfind verbosity

For a complete description of utop, look at the utop(1) manual page.
-( 15:13:56 )-< command 6 >------------------------------------{ counter: 0 }-
utop # #quit;;
maxboeker@w197-65-v4 fpv21w01t05-ge32fis % 
```

  - How to exit OCaml interpreter? `Ctrl + D` or `#quit;;`
- Expressions: indicate input with # and cause evaluation of given input with ;; , e.g. `# 3+4;; - : int = 7` (OCaml indicates resulting type based on given operation)
  - output indicated by `- : <type> <value>`
  - successful declaration of variable, function etc. by `<data object> <name> : <type> <value>`
- Pre-defined Constants and Operators

Type	Constants: examples	Operators
int	0 3 -7	+ - * / mod
float	-3.0 7.0	+. -. *. /.
bool	true false	not || &&
string	"hello"	^
char	'a' 'b'	

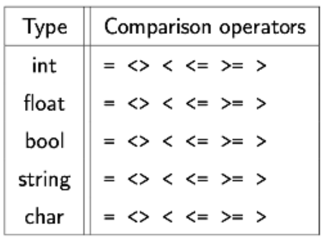
```
(* Computing a to the power of b, i.e. a^b *)
# 2. ** 3.;;
- : float = 8.

(* defining multiplication function mul in two equal ways *)
# let mul1 a b = a*b;;
val mul1 : int -> int -> int = <fun>

# let mul2 = ( * );;
val mul2 : int -> int -> int = <fun>
```

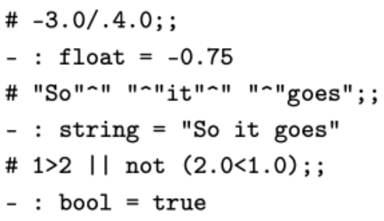
  - unknown / wildcard value / regardless of its value or type : `_`
  - Note the difference between the wildcard `_` and `unit` type

```
# fun _ -> 0;; (* polymorphic datat type *)
- : 'a -> int = <fun>
# fun () -> 0;; (* unknown value / wildcard, value is not bound and irrelevant *)
- : unit -> int = <fun>
```



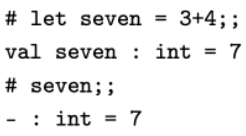
Type	Comparison operators
int	= <> < <= >= >
float	= <> < <= >= >
bool	= <> < <= >= >
string	= <> < <= >= >
char	= <> < <= >= >

<> = ≠



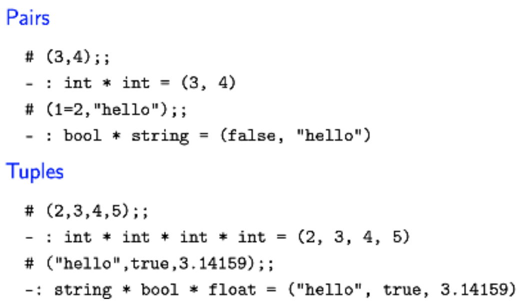
```
# -3.0/.4.0;;
- : float = -0.75
# "So"^" "^"it"^" "^"goes";;
- : string = "So it goes"
# 1>2 || not (2.0<1.0);;
- : bool = true
```

  - Not that operator precedence: less operator binds stronger than logical disjunction, not operator binds stronger than less operator → important for brackets
- declare variables as a constant (for ever!)



```
# let seven = 3+4;;
val seven : int = 7
# seven;;
- : int = 7
```

  - declaring another variable with the same name does not assign the new value to the already existing variable, but creates a new variable with the same name; old variable hidden/lost
- complex datatypes



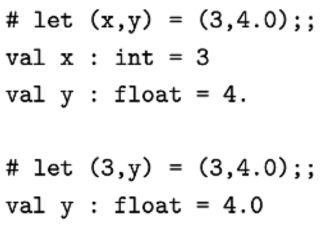
Pairs

```
# (3,4);;
- : int * int = (3, 4)
# (1=2,"hello");;
- : bool * string = (false, "hello")
```

Tuples

```
# (2,3,4,5);;
- : int * int * int * int = (2, 3, 4, 5)
# ("hello",true,3.14159);;
-: string * bool * float = ("hello", true, 3.14159)
```

- simultaneous definition of variables



```
# let (x,y) = (3,4.0);;
val x : int = 3
val y : float = 4.

# let (3,y) = (3,4.0);;
val y : float = 4.0
```

- Records similar to dictionaries in python

```
# type person = {given:string; sur:string; age:int};;
type person = { given : string; sur : string; age : int; }
# let paul = { given="Paul"; sur="Meier"; age=24 };;
val paul : person = {given = "Paul"; sur = "Meier"; age = 24}
# let hans = { sur="kohl"; age=23; given="hans"};;
val hans : person = {given = "hans"; sur = "kohl"; age = 23}
# let hansi = {age=23; sur="kohl"; given="hans"}
val hansi : person = {given = "hans"; sur = "kohl"; age = 23}
# hans=hansi;;
- : bool = true
```

- order irrelevant, declaration by `type` command, names start with lowercase letter
- accessing record components:

```
(* via selection of components, i.e. dot notation *)
# paul.given;;
- : string = "Paul"

(* with pattern matching *)
# let {given=x;sur=y;age=z} = paul;;
val x : string = "Paul"
val y : string = "Meier"
val z : int = 24

(*and if we are not interested in everything*)
# let {given=x} = paul;;
val x : string = "Paul"
```

- Case Distinction: `match` and `if`

```
match n (*same as switch case in C*)
  with 0 -> "null" (*do something for 0*)
      | 1 -> "one" (*do something for 1*)
      | _ -> "uncountable!" (*do something for the rest*)

match e (* can also be written as if e then e1 else e2*)
  with true  -> e1
      | false -> e2

(* use when to check for conditions in match pattern *)
let rec remove k ms = match ms with
  | [] -> []
  | (k',_)::ms when k' = k -> remove k ms
  | m:ms -> m::remove k ms
```

  - Watch out for redundant and incomplete matches: case distinction must make sense

- Lists constructed by means of `[]` and `::` , can be arbitrarily long

```
# let mt = [];;
val mt : 'a list = []
# let l1 = 1::mt;;
val l1 : int list = [1]
# let l = [1;2;3];;
val l : int list = [1; 2; 3]
# let l = 1::2::3::[];;
val l : int list = [1; 2; 3]
```

  - Caveat (i.e. condition): elements must have same type
  - lists in OCaml are stacks, i.e. elements can only be added in the front; list cant be traversed and appended as in Java or C using match cases
  - `tau list` : list with elements of type `tau`
  - **'a is a polymorphic type variable, i.e. a variable that can be of any type**
  - traversing through a list with a case distinction

```
# match l
    with []    -> -1
        | x::xs -> x;;
-: int = 1
```

- lists can be manipulated by adding values in front or at back

```
(* adding infront *)
x :: list

(* adding at back by concatenating existing list with new list element *)
list @ [x]
```

- **Functions** (are 1st class citizens, i.e. treated like any other variable)
  - after function name follow the parameters; function name = variable whose value is a function; a function is a value → first class citizen, i.e. a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable → higher-order functions

```
# let double x = 2*x;;
val double : int -> int = <fun>
# (double 3, double (double 1));;
- : int * int = (6,4)

(*alternative way of defining a function*)
# let double = fun x -> 2*x;;
val double : int -> int = <fun>
```

  - Caveat: function can access values of variables defined before the function was defined; a function is a value

```
# let factor = 2;;
val factor : int = 2
# let double x = factor*x;;
val double : int -> int = <fun>
# let factor = 4;;
val factor : int = 4
# double 3;; (*function call*)
- : int = 6
```

  - function can be applied to no, all, or only some of its arguments
  - unary functions are functions with only one input (although technically all functions in OCaml are unary), e.g. `Stdlib.( ~- )` is the negation of a given input
  - **function that calls itself is recursive**
    - no loops in functional languages, such as OCaml, **the only way of creating loops is creating recursive functions**
    - create recursive functions in OCaml including keyword `rec`

```
(* factorial function defined as a recursive function *)
# let rec fac n = if n<2 then 1 else n * fac (n-1);;
val fac : int -> int = <fun>

(* fibonacci function defined as recursive function *)
# let rec fib = fun x -> if x <= 1 then 1
                          else fib (x-1) + fib (x-2);;
val fib : int -> int = <fun>

(* mutually recursive function *)
# let rec even n = if n=0 then "even" else odd (n-1)
      and odd n = if n=0 then "odd" else even (n-1);;
val even : int -> string = <fun>
val odd : int -> string = <fun>
```

  - when dealing with lists we need to do case distinction to traverse it and distinct between empty and none empty lists
    - `function` takes one argument and immediately does pattern matching, patterns define the input

- list of parameters are declared as: `fun <name> parameter1 parameter2 parameter3 = <do something>`

```
# let rec len = fun l -> match l
                          with [] -> 0
                          | x::xs -> 1 + len xs;;
val len : 'a list -> int = <fun>
# len [1;2;3];;
- : int = 3

(* pattern matching *)
# let rec len = function [] -> 0
                  | x::xs -> 1 + len xs;;
val len : 'a list -> int = <fun>
# len [1;2;3];;
- : int = 3

(* combining two lists together through
case distinction for several arguments*)
# let rec app l y = match l
                     with [] -> y
                     | x::xs -> x :: app xs y;;
val app : 'a list -> 'a list -> 'a list = <fun>
# app [1;2] [3;4];;
- : int list = [1; 2; 3; 4]
(* can also be written in a unIry function *)
# let rec app = function [] -> fun y -> y
                    | x::xs -> fun y -> x::app xs y;;
val app : 'a list -> 'a list -> 'a list = <fun>
# app [1;2] [3;4];;
- : int list = [1; 2; 3; 4]
```

- Note that pattern matching functions don't necessarily need an arg as pattern already defined at least one arg, also note that one may not require " | " first pattern

```
let inspect = function
  | Node (v, l, r) -> Some (v, l, r)
  | Empty -> None

(* is the same as *)

let inspect = function
  Node (v, l, r) -> Some (v, l, r)
  | Empty -> None
```

- local definitions introduced by `let` :

  `let pattern = expression in other-expression`

```
# let     x = 5
  in let sq = x*x
  in     sq+sq;;
- : int = 50
# let facit n = let rec (* computation of n! *)
      iter m yet = if m>n then yet
                   else iter (m+1) (m*yet)
      in iter 2 1;;
val facit : int -> int = <fun>

(* Note that local definitions of functions allow us to define
functions in the following way *)
(* normal function *) let f x y = x+y;;
(* equally we can define a function g(x) that is a function y which
returns x+y *) let g x = fun y -> x+y;;
```

- `let ... in` allows to define multiple functions or variables usable in another function; note that in order to use a definition it must be defined beforehand

- **Defining a function in different ways**: note that a function can be defined in the following three ways while all three statements result in the same function

```
# let f1 y = fun x -> x+y;;
val f1 : int -> int -> int = <fun>
# f1 2 2;;
- : int = 4

# let f2 x y = x+y;;
```

```
val f2 : int -> int -> int = <fun>
# f2 2 2;;
- : int = 4

# let f3 = fun x -> fun y -> x+y;;
val f3 : int -> int -> int = <fun>
# f3 2 2;;
- : int = 4
```

- **Pattern matching and recursive functions**
  - How to read `h :: t` ? bounding head (the first element of the given list) in `h` and tail (whatever the rest of the given list, i.e. all elements except the first element) in `t` ; Thus `h::t` accesses the first and remaining elements of a given list.

```
let rec sum lst =
  match lst with
  | [] -> 0
  | h :: t -> h + sum t

(* in order to see the calls a function makes,
use the directive "#trace functionname",
untrace it with "#untrace functionname" *)
utop # sum [1;2;3;4;5;6];;
sum <-- [1; 2; 3; 4; 5; 6]
sum <-- [2; 3; 4; 5; 6]
sum <-- [3; 4; 5; 6]
sum <-- [4; 5; 6]
sum <-- [5; 6]
sum <-- [6]
sum <-- []
sum --> 0
sum --> 6
sum --> 11
sum --> 15
sum --> 18
sum --> 20
sum --> 21
- : int = 21
```

  - One can also simply copy / paste a piece of code in a utop environment to use it immediately.
- **User-defined datatypes**
  - **Enumeration Types** `type` : order appearance determines which variable is superior to the other making constructors comparable; note constructors are either nullery or unery which means they either contain no or one argument

```
(* example enumeration type of playing cards *)
# type color = Diamonds | Hearts | Gras | Clubs;;
type color = Diamonds | Hearts | Gras | Clubs
(* based on definition of color Diamonds > Hearts *)
# type value = Seven | Eight | Nine | Jack | Queen | King | Ten | Ace;;
type value = Seven | Eight | Nine | Jack | Queen | King | Ten | Ace
# Clubs;;
- : color = Clubs
# let gras_jack = (Gras,Jack);;
val gras_jack : color * value = (Gras,Jack)

(* pattern matching example function of enumeration types taking
pair as input *)
# let is_trump = function
               | (Hearts,_) -> true
               | (_,Jack)  -> true
               | (_,Queen)   -> true
               | (_,_)      -> false
val is_trump : color * value -> bool = <fun>
```

    - typing errors recognizable as alternatives (called constructors) separated by | are uniquely assigned to a type; constructors start with capital letter, e.g. Hearts
    - **Sum Types**: allow enumeration types to have arguments in their constructors

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

- `Option` is a module collecting useful functions and values for `type option`
- `Some` enables one to create an option for some values, e.g. `Some (1, "abc");; -: (int * string) option = Some (1, "abc")` or `Some 10;;` returns `-: int option = Some 10`
- `None` an optional value
- option types is polymorphic (can be defined for any type `'a`) and useful for defining partial functions when there is no result

```
(* get the value for key a, if the list is empty no value for
the given key can be returned so we return None; however,
if key is found we want to return the value, but the type of the
returned variable must be the same type so we make it optional
with Some z*)
let rec get_value a l = match l with | [] -> None
      | (b,z)::rest -> if a=b then Some z else get_value a rest
```

◦ **Recursive Datatypes,** i.e. nested data types that can be evaluated recursively by unpacking expression layer by layer

```
type sequence = End | Next of (int * sequence)

# Next (1, Next (2, End));; (* sequence of 2 *)
- : sequence = Next (1, Next (2, End))

(* could also be polymorphic where the type is 'a sequence *)
type ’a sequence = End | Next of (’a * ’a sequence)

# Next (1, Next (2, End));;
- : int sequence = Next (1, Next (2, End))

(* using recursive datatypes in recursive functions *)
# let rec nth n s = match (n,s) with | (_,End) -> None
                      | (0,Next (x,_)) -> Some x
                      | (n,Next (_, rest)) -> nth (n-1) rest;;
val nth : int -> int sequence -> int option = <fun>

# nth 4 (Next (1, Next (2, End)));;
- : int = None
# nth 2 (Next (1, Next(2, Next (5, Next (17, End)))));;
- : int = Some 5

(* another example *)
# let rec down = function
                     0 -> End
                   | n -> Next (n, down (n-1));;
val down : int -> int sequence = <fun>

# down 3;;
- : int sequence = Next (3, Next (2, Next (1, End)));;
# down (-1);;
Stack overflow during evaluation (looping recursion?).
```

- **Last Calls:** is a call to another function which returns a value and this value then represents the result of the function from which the other one was called, there are no further computations required

```
let f x = x + 5
let g y = let z = 7
          in if y > 5 then f (-y) (* -> last call as result is result of g*)
             else z + f y (* not a last call as addition in fun g with
                            z still missing *)
```

◦ last calls do not let stack spaces, needed to execute a function, grow resulting in very efficient code

◦ recursive functions are called **tail recursive** if all calls to it are last: tail-recursive functions can be executed as efficiently as “loops”, intermediate results are handed from one recursive call to the next, stopping rule computes result

```
(* Examples of functions with only last calls,
i.e. tail-recursive functions *)
let rec fac1 = function
       (1, acc) -> acc
     | (n, acc) -> fac1 (n - 1, n * acc);; (* fac1 is tail rescursive because the recursive call will first execute (n - 1, n
                                              and then as last execution step calls fac1 with the evaluated arguments *)
```

```
let rec loop x = if x < 2 then x
                 else if x mod 2 = 0 then loop (x / 2)
                 else loop (3 * x + 1);;
```

- usually recursive functions require the same number of stack spaces as it recursively creates different function calls
- tail recursive functions with only last calls can use the same stack space with every new recursive function call, thus is very efficient

```
(* reversing a list with app function and tail-recursive function *)
let rec app = function [] -> fun y -> y
                    | x::xs -> fun y -> x::app xs y;;
let rec rev list = match list with [] -> []
                    | x::xs -> app (rev xs) [x]
(* => quadratic runtime! *)

let rev list = let rec r a l = match l with [] -> a
                 | x::xs -> r (x::a) xs
                 in r [] list
(* linear runtime! by introducing auxilia function which adds
list elements of l to a in reverse order *)
```

- **Higher-order Functions - Currying** is the form of writing a function in an easier form: instead of writing the function `f (a, b)` one can write `f a b` here `f` is applied to `a` which returns a function which is then applied to `b` to fully evaluate the function `f a b` , `f` is thus considered to be a higher order function, the application of `f` to a single argument `a` is called **partial** as another argument must be provided to evaluate the entire body of the function `f`
  - in general high order functions are functions that receive functions as inputs or give functions as output
- **List Functions**:

```
let rec map f = function [] -> []
        | x::xs -> f x :: map f xs

(* fold_left has 3 arguments: function f accumulator a and list,
which is not a specific paramter, but reflected through the matching
function*)
let rec fold_left f a = function [] -> a
        | x::xs -> fold_left f (f a x) xs

(* not tail recursive! *)
let rec fold_right f = function [] -> fun b -> b
        | x::xs -> fun b -> f x (fold_right f xs b)

(* find a value in a list that satisfies the condition f *)
let rec find_opt f = function [] -> None
        | x::xs -> if f x then Some x
                   else find_opt f xs
```

  - `fold_left f a l`: applies `f` to every element of given list `l` and stores result in `a` from left to right

```
utop # List.fold_left (fun temp_list x -> x::temp_list) [] [0;1;2];;
- : int list = [2; 1; 0]
```

  - these functions specify the recursion according to the list structure, therefore called recursion schemes or functionals and are independent of the element type of the list, that must only be known to the function f
- **Polymorphic Functions** operate on equally structured data of any type

```
(* above mentioned functionals have following types, they can be
instantiated by any type, but each occurrence with the same type *)
(* map gets a function & list as arguments and returns a list *)
map : (’a -> ’b) -> ’a list -> ’b list
fold_left : (’a -> ’b -> ’a) -> ’a -> ’b list -> ’a
fold_right : (’a -> ’b -> ’b) -> ’a list -> ’b -> ’b
find_opt : (’a -> bool) -> ’a list -> ’a option

(* for example calling map with string_of_int will instantiate 'a
with int resulting in int list and 'b with string resulting in
string list *)
# string_of_int;;
```

```
val : int -> string = <fun>
# map string_of_int;;
- : int list -> string list = <fun>

(* more examples *)
let compose f g x = f (g x)
let twice f x = f (f x)
let rec iter f g x = if g x then x else iter f g (f x);;

val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
val iter : ('a -> 'a) -> ('a -> bool) -> 'a -> 'a = <fun>
```

- if a functional is applied to a polymorphic function, the result may again be polymorphic

- **Polymorphic Datatypes**: user-defined datatypes may be polymorphic as well

```
(* tree storing data in the leafs, data of any type therefore 'a ;
every node has two leafs ; tree is type constructor as possible to create
a new type from another type *)
type 'a tree = Leaf of 'a
             | Node of ('a tree * 'a tree)
```

  - in order to use polymorphic datatypes we should use polymorphic functions to make the most of it

```
let rec  size = function (* get size of a tree *)
       Leaf _     -> 1
       | Node(t,t') -> size t + size t'

let rec flatten = function (* insert all tree elements into a list *)
       Leaf x     -> [x]
       | Node(t,t') -> flatten t @ flatten t' (* take left subtree,
insert elements to the list, then append (using build-in function @)
that list to the resulting list of inserting all elements from
right subtree *)

(* above flatten implementation is close to quadratic runtime as
append run time is as long as lefthandside list, thus rewriting it to
using accumulating parameter and adding leafs of every node to the list
of right elements *)
let flatten1 t = let rec doit = function
                   (Leaf x, xs) -> x :: xs
             | (Node(t,t'), xs) -> let xs = doit (t',xs)
                                   in doit (t,xs)
       in doit (t,[])
```

  - operator `@` concatenates two lists

    ```
    # [1;2] @ [3;4]
    - : int list = [1;2;3;4]
    ```

- **Application: Queues**
  - requirements: enqueue, dequeue, is_empty, queue_of_list, list_of_queue
  - implementation: represent queue through two lists, the 1st list is in the right order for extraction and the 2nd in the reverse order for insertion, such that making enqueuing very easy by simply putting an element in front of the 2nd list, dequeuing is easy by taking the first element of the first list, if the first list is empty no elements can be dequeued therefore we have to reverse the second list and use it then to dequeue elements

```
type 'a queue = Queue of 'a list * 'a list
let is_empty = function
               Queue ([],[]) -> true
               | _           -> false
let queue_of_list list = Queue (list,[])
let list_of_queue = function
                Queue (first,[])     -> first
                | Queue (first,last) -> first @ List.rev last
let enqueue x (Queue (first,last)) =
               Queue (first, x::last)
let dequeue = function
    Queue ([],last) -> (match List.rev last
                with [] -> (None, Queue ([],[]))
```

```
                | x::xs -> (Some x, Queue (xs,[])))
    | Queue (x::xs,last) -> (Some x, Queue (xs,last))
```

- **Anonymous Functions:** functions are data which can be used without naming

```
(* example *)
# fun x y z -> x+y+z;;
- : int -> int -> int -> int = <fun>
```

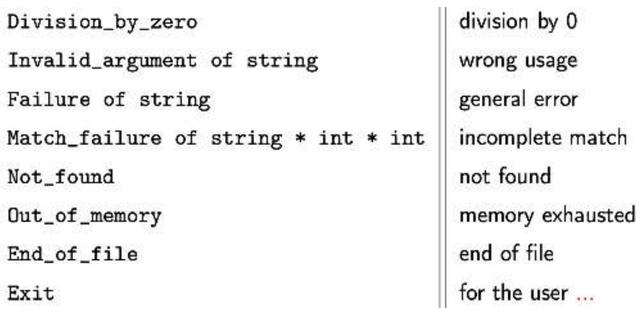
  - this cant be used for recursive functions as we need a name to reference to the same function

- **Exceptions:**
  - in OCaml we have no objects which are normally used to handle exceptions (e.g. Java), in OCaml we only have functions and values, in OCaml we raise exceptions

```
(* Example Build-In Exceptions *)
# 1 / 0;;
Exception: Division_by_zero.
# List.tl (List.tl [1]);;
Exception: Failure "tl".
# Char.chr 300;;
Exception: Invalid_argument "Char.chr".

# match 1+1 with 0 -> "null";;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: 1
Exception: Match_failure ("", 2, -9).
```

  - other pre-defined exceptions; exceptions are first-class citizen of datatype `exn`



Division_by_zero	division by 0
Invalid_argument of string	wrong usage
Failure of string	general error
Match_failure of string * int * int	incomplete match
Not_found	not found
Out_of_memory	memory exhausted
End_of_file	end of file
Exit	for the user ...

  - exceptions can also be created

```
# exception Hell;;
exception Hell
# Hell;;
- : exn = Hell

# Division_by_zero;;
- : exn = Division_by_zero

(* creating a failure *)
# Failure "complete nonsense!";;
- : exn = Failure "complete nonsense!"

(* Creating exceptions with arguments *)
# exception Hell of string;;
exception Hell of string
# Hell "damn!";;
- : exn = Hell "damn!"
```

  - exceptions can be raised and handled

```
# let divide (n,m) = try Some (n / m)
      with Division_by_zero -> None;;

# divide (10,3);;
- : int option = Some 3
```

```
# divide (10,0);;
- : int option = None
```

- `try` blocks define return value, when exception is raised `with` block is executed and matched to appropriate exception, thus value of `with` block must be the same as return value of `try` block

  ```
  let rec member x l = try if x = List.hd l then true
                           else member x (List.tl l)
      with Failure _ -> false

  # member 2 [1;2;3];;
  - : bool = true
  # member 4 [1;2;3];;
  - : bool = false

  (* handling multiple exceptions *)
  try <exp>
  with <pat1> -> <exp1> | ... | <patN> -> <expN>
  ```

  - note that `List.hd` and `List.tl` allow to access head and tail of a list without pattern matching

- one can trigger exceptions through keyword `raise` , exceptions may occur at any sub-expression arbitrarily nested

  ```
  # let f (x,y) = x / (y-1);;
  # let g (x,y) = try let n = try f (x,y)
                              with Division_by_zero ->
                                   raise (Failure "Division by zero")
                      in string_of_int (n*n)
                  with Failure str -> "Error: "^str;;
  # g (6,1);;
  - : string = "Error: Division by zero"
  # g (6,3);;
  - : string = "9"
  ```

- **Textual Input and Output:** reading input and writing output violates purely functional programming; operations are realized by means of side-effects, i.e. means of functions whose return value is irrelevant e.g. `unit`, ordering of evaluation matters
  - `unit` is the empty value, one can think of it as the 0-tuple or `()` , it contains no information or no value which is particular useful when an expression doesn't return anything or it is irrelevant what it returns, because all functions in OCaml must take an argument and return a value

  ```
  (* Printing string has return type unit,
  note print_string creates side-effect *)
  (* val print_string : string -> unit *)
  # print_string "Hello World!\n";;
  Hello World!
  - : unit = ()

  (* read_line takes no parameter thus unit and returns previous value
  on console: *)
  (* val read_line : unit -> string *)
  # read_line ();;
  Hello World!
  - : string = "Hello World!"

  (* reading from files: files must be opened and closed,
  otherwise if no more lines to read exception
  End_of_file raised *)
  (* val open_in : strong -> in_channel *)
  # let infile = open_in "test";;
  val infile : in_channel = <abstr>

  (* val input_line : in_channel -> string *)
  # input_line infile;;
  - : string = "The file’s single line ...";;
  # input_line infile;;
  Exception: End_of_file

  (* val close_in : in_channel -> unit *)
  # close_in infile;;
  - : unit = () (* we get unit as no actual value is provided i.e. empty *)
  ```

  - `stdin : in_channel` is the standard input as channel.

  - `input_char : in_channel -> char` returns the next character of the channel.
  - `in_channel_length : in_channel -> int` returns the total length of the channel.

```
(* output to files is analogous *)
(* val open_out : string -> out_channel *)
# let outfile = open_out "test";;
val outfile : out_channel = <abstr>

(* val output_string : out_channel -> string -> unit *)
# output_string outfile "Hello ";;
- : unit = ()
# output_string outfile "World!\n";;
- : unit = ()
```

- **Sequences:** sequencing using the sequence operator (binary operator) `;`

```
# print_string "Hello";
  print_string " ";
  print_string "world!\n";;
Hello world!
- : unit = ()

(* printing all elements of a list with build-in function List.iter *)
#  let rec iter f = function
       []    -> ()
     | x::[] -> f x
     | x::xs -> f x; iter f xs;;
val iter : ('a -> unit) -> 'a list -> unit = <fun>
```

  - big-step operation for sequencing operator `;` with phi = e_0 terminates and e_1 ⇒ v

$$\frac{\phi}{e_0; e_1 \Rightarrow v}$$

- `Printf.fprintf (open_out "foo") "bar"` creates side-effects, writes "bar" into file "foo", but cannot create exception `End_of_file` as this exception can only be raised when writing into file