# Chapter 2

# Basic Concepts in Functional Programming

> "An ideal language allows us to express easily what is useful for the programming task and at the same time makes it difficult to write what leads to incomprehensible or incorrect programs."
>
> - Nico Habermann

OCaml is a *statically typed functional* programming language. What does this mean? - In functional programming languages we write programs using *expressions*, in particular functions. These *functions compute a result* by manipulating and analyzing values recursively. OCaml as many ML-like languages (such as SML, Haskell, F#, etc.) facilitates defining recursive data structures and programming functions via *pattern matching* to analyze them. This often leads to elegant, compact programs where a functional program computes a result (i.e. a boolean, an integer, a list, etc.).

Functional programming is also often associated with effect-free programming, i.e. programmers do not explicitly allocate and update memory, there are no exceptions programmers could use to divert the control flow. Functional languages that do not support effects are often called pure. Haskell is such a language. OCaml on the other hand supports both exception handling as well as state-full computation and hence is an impure functional language. It is hence ideally suited to learn functional programming and at the same time gain a deeper understanding of effectful programming, a style that is common and familiar to programmers using imperative or procedural languages such as C for example. In procedural or imperative languages we typically update some state by assigning values to variables or fields, and we observe the result as an effect, i.e. we print the result on standard output or we read and lookup the state and value of a given variable. Using OCaml we will be able

to contrast pure functional programming vs effectful programming where we store a value in memory and observe the effect of computations.

What does statically typed mean? - Types approximate the runtime behaviour of an expression statically, i.e. before running and executing the program. We can also say "Types classify expressions by their values". Basic types are integers, booleans, floating point numbers, strings or characters. But we can in fact also reason about functions by relating the input and output of a function. And we can then check whether functions are used with the correct types of input and reason about the correct use of the results they compute. Static type checking is a surprisingly effective technique. It is quick (i.e. for most programs it is polynomial). It can be re-run every time the program changes. It gives precise error messages where the problem might lie and how it might be fixed.

Type checking is conservative. It examines the code statically purely based on its syntactic structure checking whether the given program is meaningful, i.e. its evaluation does something sensible. There are programs which the type checker rejects, although nothing would go wrong during evaluation. But more importantly, if the type checker succeeds, then we are guaranteed that the execution of the program does not lead to a core dump or crash.

Statically typed functional languages like OCaml (but also Java, ML, Haskell, Scala for example) are often called type-safe, i.e. they guarantee that if a program is well-typed, then its execution does not go wrong, i.e. either it produces a value, or it aborts gracefully by raising a runtime exception, or it continues to always steps to a well-defined state. This is a very strong guarantee. As a consequence, typed functional programs do not simply crash because of trying to access a null pointer, trying to access a field in an array that is out of bound or trying to write over and beyond a given buffer (buffer overrun). In fact, the heartbleed bug in 2014 is a classic example of a type error and could have been easily avoided.

Let's begin slowly. We begin using OCaml's interactive toplevel, aka a read-eval-print loop, that we can use to play around. To start simply type `ocaml` in a terminal:

```
[bpientka][taiko][~]$ ocaml
        OCaml version 4.02.1
#
```

We can now type expressions followed by `;;` to evaluate them.

## 2.1 Expressions, Names, Values, and Types

The most basic expressions are numbers, strings, and booleans.

```
# 3;;
- : int = 3
# 2;;
- : int = 2
# 3 + 2 ;;
- : int = 5
#
```

OCaml says that `3` is an integer and it evaluates to `3`. More interestingly, we can ask what the value of the expression `3 + 2` is and OCaml will return `5` and tell us that its type is `int`.

As you can see, the format of the toplevel output is

```
<name> : <type> = <value>
```

If we do not bind the value resulting from evaluating an expression to a name, OCaml will simply write _ instead. We often want to introduce a name to be able to subsequently refer to it. We come back to the issue of binding a little later.

We can not only compute with integers, but also for example with floating point numbers.

```
# 3.14;;
- : float = 3.14
# 5.0 /. 2.0;;
- : float = 2.5
#  3.2 +. 4.5;;
- : float = 7.7
#
```

Note that we have a different set of operators to compute with floating point numbers. All arithmetic operators have as a postfix a dot. You might wonder whether it is possible to simply say `5  3.2+`. The answer is no as you can see below.

```
# 5 + 3.2;;
Characters 4-7:
  5 + 3.2;;
      ^^^
Error: This expression has type float but an expression was expected of type
         int
#
```

Arithmetic operators are not *overloaded* in OCaml. Overloading operators requires that during runtime we must decide what function (or method) to choose base on the type of the arguments. This requires that we keep types around during runtime and it can be expensive. OCaml does not keep any types around during runtime - this is unlike languages such as Java. In OCaml (as in many functional languages) types are purely used for statically reasoning about the code and optimizing the compilation of the code.

The example illustrate another benefit of types: precise error messages. The type checker tells us that the problem seems to be in the right argument which is passed to the operator for addition.

OCaml has other standard base types such as strings, characters, or booleans.

```
# "comp302";;
- : string = "comp302"
# 'a';;
- : char = 'a'
# true;;
- : bool = true
# false;;
- : bool = false
# true || false ;;
- : bool = true
# true && false;;
- : bool = false
#
```

Here we have used boolean operators `||` (for disjunction) and `&&` (for conjunction). We can also use if-expressions.

```
# if 0 = 0 then 1.0 else 2.2;;
- : float = 1.
#
```

As we mentioned type checking is conservative. Hence there are programs that would produce a value, but the type checker rejects them. This happens for example in the program below:

```
Characters 23-28:
  if 0 = 0 then 1.0 else "AAA";;
                         ^^^^^
Error: This expression has type string but an expression was expected of type
         float
#
```

Clearly the guard `0 = 0` is always true and we would never reach the string `''AAA''`. We can also say the second branch is dead code. While this is obvious when our guard is of the form `0 = 0`, in general this is hard to detect statically. In principle, the guard could be a very complicated call to a function which always happens to produce true for a given input. The type checker makes no assumption about the actual value a guard has, but only verifies that the guard would evaluate to a boolean. Hence the type checker does not know what branch will be taken during runtime and it must check that both branches produce the same kind of value. Why? - Because the if-expression may be part of a larger expression. Reasoning about the type of expressions is compositional.

Read Expression → Type Check → Evaluate

```
# (if 0 = 0 then 1.0 else 2.2) +. 3.3;;
```

```
- : float = 4.3
#
```

As mentioned earlier, if the type checker accepts the program, executing it either yields a value or the program is aborted gracefully, if it reaches a state that is identified as a run-time error. An example of such a run-time error is division by zero.

```
# 3 / 0;;
Exception: Division_by_zero.
```

Attempting to divide `3` by `0` will pass the type checker, because the type checker simply checks whether the two arguments given to the division operator are integers. This is the case. During runtime we note that we are dividing by zero, and the evaluator raises a built-in exception `Division-by-zero`.

## 2.2 Variables, Bindings, and Functions

A central concept in a programming language are variables and the scope of variables. In OCaml we can declare a variable at the top-level (i.e. a global declaration) using a let-expression of the form `let <name> = <expression>` as follows:

```
# let pi = 3.14;;
val pi : float = 3.14
#
```

Here `pi` is the name of the variable and we bind to the name the value `3.14`. Note, because OCaml is a call-by-value language, we bind *values* to variable names - not expressions. For example as a result of evaluating the following expression, we bind the variable name `x` to the value `9`.

```
# let x = 3 * 3;;
val x : int = 9
#
```

A variable binding is not an assignment. We simply establish a relation between a name and a value. There is no state being allocated and this association or relation cannot be updated or changed. Once the relationship is done it is fixed. We can only *overshadow* a given binding, not update it.

```
# let x = 42;;
val x : int = 42
#
```

For example by establishing again a binding between the variable name `x` and the value `42`, we now have two bindings for `x` on the binding stack. The last binding we pushed onto that stack bind `x` to `42`. When we look up the value of a variable, we

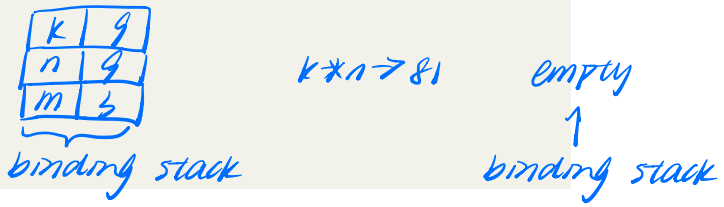
look it up in the stack and we pick the one that was establish most recently, i.e. the binding that is at the top. The earlier binding is *overshadowed*.

A garbage collector might decide to remove the earlier binding for efficiency reasons, it determines there is no other code that still uses it.

We can also introduce scoped variable bindings or local bindings as illustrated in the following example.

```
# let m = 3 in
  let n = m * m in
  let k = m * m in
    k*n
;;

        - : int = 81
#
```




We use a let-expression that has the following structure:

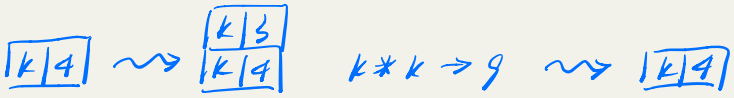
`let <name> = <expression 1> in <expression 2>.`

The `<name>` can be used in the body, i.e. `<expression 2>`, to refer to the value of `<expression 1>`, i.e. we bind the value of `<expression 1>` to the variable `<name>` and continue evaluating `<expression 2>` using this new binding. The binding of variable `<name>` to the value of `<expression 1>` ends after the let-expression has finished evaluating and the binding is removed from the binding stack.

We also say the scope of the variable `<name>` ends after `<expression 2>`.

How do global and local bindings interact? - Local bindings are only temporary. Hence, they may overshadow temporarily a given global binding. Here is an example:

```
# let k = 4;;
val k : int = 4
# let k = 3 in k * k ;;
- : int = 9
# k;;
- : int = 4
#
```




When we evaluate the body `k * k` in the second let-expression, we have two bindings for the name `k`: the first one bound `k` to `4`; the second one and the most recent one that is on top of the binding stack says `x` is bound to `3`. When we evaluate `k * k`, we look up the most recent binding for `k` and obtain `3`. Hence we return the value `9`. When we then ask what is the value of `k` we obtain `4`, as the local binding between `k` and `3` was removed from the binding stack. This clearly illustrates that variables bindings are not updated - once they are made they persist.

In functional language such as OCaml we cannot only names to values such as numbers or strings, but functions are also values. For example, we may define a

function `area` which when given a floating point number as input computes the area of a circle. The expression below makes explicit that `area` is a name and it is bound to a function.

```
let area : float -> float = function r -> pi *. r *. r;;
let area : float -> float = fun r -> 3.14 *. r *. r;;
```

In OCaml, there are two ways to represent functions explicitly using the keyword `function r -> ...` and `fun r -> ...`. We will get back to the differences later.

For convenience, we often write the input as part of the right hand side when defining functions.

```
let area (r:float) = pi *. r *. r;;
```

All the given definitions are equivalent. All of them have function type `float -> float`.

We call functions by simply passing an argument of the appropriate type to the function.

```
let a2 = area (2.0);;       (* result a2 = 12.56 *)
```

Let's re-define `pi`.

```
let (pi : float) = 6.0;;       (* now pi = 6.0 *)
```

Recall that we only establish bindings and we now add to the binding stack the binding of `pi` to `6.0`. We do not update the previous binding of `pi`. We simply *over-shadow* the binding. If we refer to `pi` now, we will get `6.0`. However, functions (in fact any previous binding) only see the past, but they do not see the future. Hence, when we re-execute

```
let a3 = area (2.0);;       (* result a3 = 12.56 *)
```

we still get the same result as previously! To obtain a function `area` which indeed uses the new definition of `pi = 6.0`, we also need to re-define and overshadow the definition `area`. For examle, we can overshadow `pi` with a more accurate definition of $\pi$ and and redeclare area as follows:

```
let pi = 4. *. atan 1.;;

let area (r:float) = pi *. r *. r;;

let a4  = area (2.0);;       (* result a4 =  12.5663706144 *)
                             (* it will use the new pi and the new area def.
   *)
```

Last, we discuss recursive functions. In OCaml recursive functions are declared using the key word `let rec`. We can then implement directly the definition of factorial

# COMP 302: Programming Languages and Paradigms

Prof. B. Pientka, McGill University

McGill

## Week 1 (Part 3): Functions and Recursion

### Writing functions

Recall that functions are first-class values.

**Example 1** Three different ways to write functions.

```
let area : float -> float = function r -> 3.14 *. r *. r


let area : float -> float = fun r -> 3.14 *. r *. r


let area r =   3.14 *. r *. r
```

**Example 2** Different types – Different inputs

```


(* add : int * int -> int *)
let add (x,y) = x + y


(* add : int -> int -> int *)
let add x y = x + y
```

same time

order

product

let add = fun x -> fun y -> x+y

first input

second input

$$\text{fact } n = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{fact } (n-1) & \text{if } n > 0 \\ \text{undefined} & \text{if } n < 0 \end{cases}$$

via the following recursive program in OCaml. To characterize the undefined case, we define an exception `Domain` which we raise in the case where `n < 0`.

```
exception Domain;;

let rec fact n =
  if n < 0 then raise Domain
  else if n = 0 then 1
  else n * fact(n-1);;
```

This program is not quite satisfactory, as we want to check the condition that `n >= 0` only once – the program above checks the condition in each iteration. A better solution is to use a *local* function that is only called on inputs that are guaranteed to be greater or equal to `0`.

```
exception Domain;;

let rec fact n =
  if n < 0 then raise Domain
  else if n = 0 then 1
  else n * fact(n-1)
```

We can improve upon this implementation further by making it *tail-recursive*. A function is said to be *tail-recursive*, if there is nothing to do except return the final value. Since the execution of the function is done, saving its stack frame (i.e. where we remember the work we still in general need to do), is redundant. This allows us to write recursive definitions without worrying about space efficiency. In fact, it is a deep and surprising fact that every function in a functional programming language can be made tail-recursive! In the case of factorial, we can write it the local function `f` tail-recursively by adding an extra argument `acc` to accumulate the result. Once we reach `n=0` we simply return the accumulated result. In the recursive case, we build up the result using the accumulator. When we call the local function `f` we initialize the accumulator appropriately with `1`.

```
let fact_tr n =
  let rec f n acc =
    if n = 0 then acc else f (n-1) (n * acc)
  in
    f n 1
```

The use of a local function `f` ensures that any usedr of the function `fact` or `fact_tr` is safe and we will always check that the input is greater or equal to `0`. This ensures that there is no possible abuse by programmers and avoids possible errors, when a

programmer might use `f` instead of `fact` or `fact_tr`. This is the first simple instance of information hiding – this is an important software engineering principle we will see also later in these notes.

## 2.3 Data Types and Pattern Matching

### 2.3.1 Introduction to Non-Recursive and Recursive Data Types

Often it is useful to define a collection of elements or objects and not encode all data we are working with using our base types of integers, floats or strings. For example, we might want to model a simple card game. As a first step, we want to define the suit and rank present in the game. In OCaml, we define a finite, unordered collection of elements using a (non-recursive) data type definition. Here we define a new type `suit` that contains the elements `Clubs`, `Spades`, `Hearts`, and `Diamonds`.

```
type suit = Clubs | Spades | Hearts | Diamonds
```

We also say `Clubs`, `Spades`, `Hearts`, and `Diamonds` *inhabit* the type `suit`. Often, we simply say `Clubs`, `Spades`, `Hearts`, and `Diamonds` *are* of type `suit`. When we talk about the input or outputs of a given program, we also might simply say "given a `suit`" meaning that we are given an element of type `suit`.

When we define a collection of elements the order does not matter. Further, OCaml requires that each element begins with a capital letter.

Elements of a given type are defined by *constructors*, i.e. constants that allow us to construct elements of a given type. In our previous definition of our type `suit` we have four constructors, namely `Clubs`, `Spades`, `Hearts`, and `Diamonds`. In fact, these are simply user-defined constants that have type `suit`.

How do we write programs about elements of type `suit`? - The answer is *pattern matching* using a match-expression.

```
match <expression> with
| <pattern> -> <expression>
| <pattern> -> <expression>
 ...
| <pattern> -> <expression>
```

We call the expression that we analyze the *scrutinee*. Patterns characterize the shape of values the scrutinee might have by considering the type of the scrutinee. Let's make this more precise by looking at an example. We want to write a function `dom` that takes in a pair `s1` and `s2` of `suit`. If `s` beats of is equal to suit `s2` we return true – otherwise we return false. We will use the following ordering on suits:

`Spades` > `Hearts` > `Diamonds` > `Clubs`