Introduction to OCaml Programming

OCaml: a higher-order *functional* programming language.

Again, an online browser based interpreter is available.
    `https://try.ocamlpro.com/`

The paradigm? *Functional*
That is, functions are a "generously supported" data type, and operations with functions, such as defining functions and invoking functions are implemented efficiently.

In fact, functions are "first-class" citizens. They can be passed as arguments to other functions, and returned as results of functions. They can be *incognito* — that is, we can define and use functions without naming them (just as we can use values such as the integer 3 without giving it a name).

Where do we start? **Sets** <—> **Types**
(Later we will refine this to "Propositions = Types")

The Unit Type, corresponding to the set 1

The unit type has a single value: `()`

```
(* the type unit *)
();;
- : unit = ()
```

The type bool, corresponding to the set 2

The type of booleans has two canonical values: `true, false`

```
(* the type bool *)

true;;
- : bool = true
false;;
- : bool = false
```

(Note: One can think of $\mathbf{2} \cong \mathbf{1} + \mathbf{1}$ for some notion of a set-theoretic operator +)

OCaml has a built-in library of standard functions on the type `bool`.

Negation on the booleans is the unary operator `not` _
(not `!`, which means something completely different in OCaml).

```
not true;;
- : bool = false
not false;;
- : bool = true
```

Conjunction is the infix binary operator _ `&&` _

```
true && false;;
- : bool = false
true && true;;
- : bool = true
```

Disjunction is the infix binary operator `_ || _`

```
false || false;;
- : bool = false
false || true;;
- : bool = false
```

Negation, Conjunction and Disjunction are *functions*

```
not;;
- : bool -> bool = <fun>
(&&);;
- : bool -> bool -> bool = <fun>
(||);;
- : bool -> bool -> bool = <fun>
```

Experiment: What happens if one does not put the parenthesis around the infix operators?

The type `int`, corresponding to the set $\mathbb{Z}$

We write the canonical members of type `int` in the usual way using Indo-Arabic *numerals*. Unary negation, addition, subtraction and division are written in the traditional notation. Remainder is computed by the infix operator `mod`.

```
(* the type int *)
0 ;;
- : int = 0
1 ;;
- : int = 1
0+1 ;;
- : int = 1
1*1 ;;
- : int = 1
0*1 ;;
- : int = 0
-3 + 1 ;;
- : int = -2
- - 3 ;;
- : int = 3
-3 * -3 ;;
- : int = 9
- 3 mod 2 ;;
- : int = -1
- 3 / 2 ;;
- : int = -1
-3 mod 2 ;;
```

```
- : int = -1
3 / -2 ;;
- : int = -1
3 mod -2 ;;
- : int = 1
-3 / -2 ;;
- : int = 1
-3 mod -2 ;;
- : int = -1
```

The types of strings and characters

```
"Hello, world!" ;;
- : string = "Hello, world!"
't' ;;
- : char = 't'
'\n' ;;
- : char = '\n'
"Hello" ^ " World!\n" ;;
- : string = "Hello World!\n"
```

Defining *new* types

OCaml allows us to *define* our own types. We use the keyword `type` to indicate that we are defining a type. This type can either be (i) an alternative name for a pre-existing type or "type expression" built out of existing types; (ii) a "brand new" type with its own *constructors*. A type definition begins with the keyword `type` followed by the name of the type which we wish to define (any identifier starting with a lower-case letter, and which is not a keyword or a reserved word may be used), then the equality sign `=` and on the right-hand side of which we list the different constructors, each of which has to begin with a Capital letter.

An example of an alternative name for a pre-existing type is:

```
type age = int ;;
type age = int
```

An example of an alternative name for a type expression built using pre-existing types is:

```
type identity = string * age ;;
type identity = string * age
```

(Note `*` is used for cartesian product.)

An example of defining a new type (called `myBool`) is:

```
type myBool = T | F;;
type myBool = T | F
T;;
- : myBool = T
F;;
- : myBool = F
```

Note that `myBool` is a completely new type, distinct from any type that OCaml has in its library. The particular type consists of two distinct values, which are given by the constructors `T` and `F`. As noted above, constructors in OCaml must begin with a Capital letter. The constructors are separated by a single vertical bar `|`.

(Note: The constructors can be variously seen as function/constant symbols that evaluate to themselves, or as colours or tags on the unit value `()` which is suppressed. An alternative definition of the type could have been:

```
type myBool' = T' of unit | F' of unit ;;
type myBool' = T' of unit | F' of unit
T'() ;;
- : myBool' = T' ()
F'() ;;
- : myBool' = F' ()
```

Let us now define functions from the type **`myBool.`** The keyword to indicate a definition of a value or function is **`let`**. A function definition begins with this keyword **`let`** which is followed by the *name* of the function being defined (this can be any identifier beginning with a small letter, , and which is not a keyword or a reserved word) followed by 1 or more *parameters* to the function. Then we have the equality sign **=** , and to the right of which we have the *body* of the function — an expression that defines the result of the function.

```
let myBool2bool b = match b with
    T -> true
  | F -> false
;;
val myBool2bool : myBool -> bool = <fun>
```

The function (unimaginatively) called **`myBool2bool`** takes one parameter, namely **`b`**. It is defined using *pattern matching* i.e., case analysis with respect to the constructors **`T`** and **`F`**. The "**`match`** _ **`with`**" construct in OCaml lets us perform *case analysis* on the specified expression (**`b`** in this case) by *pattern-matching*. The cases are separated by a single vertical bar |. Each case is presented (in a clause form) as a pattern followed by an arrow **`->`** to the right of which is an expression telling us what value should be returned in that case. There are two possible cases in this function, indicated on the left side of the arrows: When the expression **`b`** evaluates to **`T`** , then the OCaml bool value **`true`** is returned, whereas when **`b`** evaluates to **`F`**, then the value **`false`** is returned.

There is nothing very interesting here, except that you have to note that we are trying to be scrupulous about separating "syntax" from "semantics", where we are using the standard OCaml type **`bool`** and its values **`true, false`** and its operators as the mathematical booleans (the semantics) whereas the new data type **`myBool`** is "syntax".

Note that we have nowhere indicated the type of the parameter, of the function, or of the return values. OCaml is remarkable in that the interpreter is able to infer that the parameter **`b`** is of our newly defined type **`myBool`**, and the possible results must be of type **`bool`** (OCaml's **`bool`** type) , and so the function **`myBool2bool`** being defined is of type **`myBool -> bool`**. (Note that the interpreter treated the function as a *value*).

How did it manage to do so? There is a type-checking engine with OCaml that inspects the different cases, and solves constraints (somewhat like how a simultaneous equation solver works), and works out the type of the parameter from the case analysis, and the type of the result from the type of the expression on the right side of the patterns. Of course, if there are inconsistencies, it reports a type error.

Note also that the OCaml interpreter does not try to print out the canonical representation of this function, and demurely reports its value to be a function (indicated by). This is because it would be wrong to presume that there is a canonical representation of functions (called "*intensions*") since a single *extensional* form of a function can be coded with multiple *intensional* forms.

We can check that our program works as expected:

```
     myBool2bool T;;
     - : bool = true
     myBool2bool F;;
     - : bool = false
```

Let us define a function **`bool2myBool`** in the reverse direction (this is an example of setting up a bijection between types **`bool`** and **`myBool`**.)

```
     let bool2myBool b = match b with
         true -> T
       | false -> F
     ;;
     val bool2myBool : bool -> myBool = <fun>
```

In this function, the case analysis on the parameter is over the OCaml built-in **`bool`** values **`true`** and **`false`** which appear to the left of the case analysis arrow.

Let us now define a toy **`myBool`** calculator, by defining "syntactic algorithms" encoding the truth tables, but within the user-defined type **`myBool`**.

```
     let myNot b = match b with
         T -> F
       | F -> T
     ;;
     val myNot : myBool -> myBool = <fun>

     let myAnd b1 b2 = match b1 with
        T -> b2
      | F -> F
     ;;
     val myAnd : myBool -> myBool -> myBool = <fun>
```

We perform case analysis on the first argument **`b1`**, and if it matches the **`myBool`** constructor **`T`**, returns whatever results from evaluating the second argument **`b2`**. Otherwise the result is **`F`**.

Note that **`myAnd`** takes two parameters **`b1`** and **`b2`**. These are given one-by-one (in that order) separated by spaces — and not as a pair **`(b1, b2)`**. This style is called a "Curry-ed" style of passing arguments (after the logician Haskell Curry). It is a standard style used in mathematics and higher-order functional programming.

Note how the type of the defined function is reported: it is a *function* that takes a single `myBool` input `b1` and returns a nameless function which is of the type `myBool -> myBool` (this unnamed function takes a single input `b2` of type `myBool` and returns an answer of type `myBool`). In other words, the types with arrows should be parenthesised with *right associativity:* `myBool -> (myBool -> myBool)`

Again OCaml is able to *infer* the type of the function from the case analysis and the type constraints collected without (painful) type annotation by us.

Get used to writing functions in Curry-ed form. Curry-ing gives us some flexibility in partial application of a function, when only the first few arguments are available.

Exercise: Similarly define a function `myOr` -- a disjunction operation on a pair of `myBool` inputs that returns a `myBool` value.

Let us now reason about the *correctness* of the function `myNot`. Programs are formal objects about which we can reason, using equational reasoning — expanding definitions in either direction and using case analysis.

Prove *forall* `b: myBool, myBool2bool (myNot b) = not (myBool2bool b)`
Proof: Case analysis on `b`

Case `b = T:`
```
   myBool2bool (myNot T)
 = myBool2bool F // defn of myNot
 = false // defn of myBool2bool
 = not true // defn of not
 = not (myBool2bool T) //  defn of myBool2bool in reverse
```
Case `b = F:`
```
   myBool2bool (myNot F)
 = myBool2bool T // defn of myNot
 = true // defn of myBool2bool
 = not false // defn of not
 = not (myBoool2bool F) // defn of myBool2bool in reverse
```

Exercise: Check this statement above is true for `b = T` and `b = F`

Exercise: State and prove the correctness of function `myAnd`.
Check your statement by running the programs on all cases.

Exercise: State and prove the correctness of your function `myOr`.
Check your program and its correctness by running the programs on all cases.