OCaml Cheatsheet

Note: this document comes from a previous course (by Sylvain Schmitz).
The explanations of the OCaml syntax in this sheet are by no means intended to be complete or even sufficient; check http://mirror.ocamlcore.org/ocaml-tutorial.org/ for further information.

1 The let Keyword, Functions

Defining constants and functions in Caml is done through the let keyword.

- let x = 2;; defines a constant x with value 2.
- let double a = 2 * a;; defines a function double that returns the double of its parameter. Another possible syntax is let double = fun a -> 2 * a;;.
- let x = ⟨value⟩ in ⟨expr⟩;; binds x to value, but only in expr.

2 The OCaml interpreter

The command ocaml (without parameters) launches the Caml interpreter. You can type Caml instructions inside and immediately get their result—it is highly recommended to use the ledit command with ocaml as argument to make interacting with this interpreter bearable.

For instance, you can type:

# let x = 2 ;;
val x : int = 2
# let double = fun a -> 2 * a ;;
val double : int -> int = ⟨fun⟩

The shell returns information about the newly defined object in the form val ⟨name⟩ : ⟨type⟩ = ⟨value⟩. Function types are of the form ⟨type_param⟩ → ⟨type_result⟩.

You may want to import a source file in the interpreter. You can do so by using the command #use "file.ml" ;;. You can also directly run a file outside the interpreter by: ocaml file.ml. If you are using an external module, you might need to provide the ocaml command with a path, e.g. ocaml -I +lambil to add the LablTK library.

3 Polymorphism, Pairs

Let us look at the identity function:

# let id = fun x -> x
val id : 'a -> 'a = ⟨fun⟩
This is a polymorphic function. It can take any type as input, and returns a value of the same type.

Given x and y, a pair can be built by using the notation (x, y). If x is of type 'a and y of type 'b, then (x, y) is of product type 'a * 'b.

**Exercise 1.** Write a function `dup` of type `'a → ('a * 'a)` that, given a value, returns a pair with the input value as both first and second elements.

## 4 Curryfication

Let us redefine addition:

```ocaml
# let addition x y = x + y ;;
val addition : int → int → int = fun
```

Defined like this, addition is a function of type `int → (int → int)`, that is, a function that takes an integer and returns a function that takes an integer and returns an integer'. This is called *curryfication*. This is the preferred way of defining functions with more than one parameter (instead of using pairs).

**Exercise 2.** Write a function `curry` of type `(('a * 'b) → 'c) → 'a → 'b → 'c`, that, given a function that uses a pair to encode parameters, returns a currified function.

**Exercise 3.** Write a function `decurry` of type `('a → 'b → 'c) → ('a * 'b) → 'c` that performs the opposite operation.

**Exercise 4.** The function `List.map` of type `('a → 'b) → 'a list → 'b list` takes a function and a list and returns the list obtained by applying the function to each element of the initial list. Write a function `add1` that takes a list and adds 1 to each element of this list.

## 5 Recursion

If you want to define a recursive function, then the `rec` keyword must be added:

```ocaml
# let f x = if x = 0 then 0 else (1 + f (x - 1));;
Error : Unbound value f
```

```ocaml
# let rec f x = if x=0 then 0 else (1 + f (x - 1));;
val f : int → int = fun
```

**Exercise 5.** Write a function `fibonacci` such that `fibonacci n` returns the n-th term of the Fibonacci sequence.

**Exercise 6.** Test your Fibonacci function with `n = 5`, then `n = 400`. If this does not terminate within a reasonable time, improve your function!
6 Sum Types, Pattern Matching

A sum type can be defined by the following syntax:

```
# type sumtype = FirstCase [of type] | 
> SecondCase [of type] ... | 
> LastCase [of type]
```

This can be recursive. For example, Caml lists could be defined by:

```
# type 'a list = Nil | Cons of 'a * 'a list
```

Then, you could build a list by using the syntax `Cons(1, Cons(2, Nil))`.

**Exercise 7.** Define a type that can represent propositional formulæ with the Not, And, and Or connectives and a Var constructor for variable names (using strings for names).

The interest of sum types lies with pattern matching. You can use the following constructions to filter x based on its type:

```
# match (x) with 
> | FirstCase [(v1)] \to ... 
> | ... 
> | LastCase [(vn)] \to ...
```

An extremely useful construction is function \(x\), which is syntactic sugar for \(\text{fun } x \to \text{match } x \text{ with}\).

**Exercise 8.** A propositional formula is in negative normal form if the negation operator is only applied to propositions.

Define a function `nnf` that turns a propositional formula into an equivalent formula in negative normal form.

**Exercise 9.** A literal is either a variable or the negation of a variable. A propositional formula is in conjunctive normal form if it is of the form:

\[
\bigwedge_i \bigvee_j \ell_{i,j}
\]

(where \(\ell_{i,j}\) are literals)

Define a function `cnf` that turns a propositional formula into an equivalent expression in conjunctive normal form.

**Exercise 10.** A list of variable assignements can be given in the following form (this uses the Caml notation for lists):

```
# let assoc = [("x", true); ("y", false); ("z", true)];
val assoc : (string * bool) list = (val)
```

Define a function `eval` of type `(string * bool) list \to propform \to bool` that evaluates a propositional formula given a variable assignement. You might want to use the function `List.assoc`; check its documentation using `man List`.

**Exercise 11.** Define a function `fv` that, given a propositional formula, returns the list of variables that appear inside, without duplicates.
Exercise 12. Define a function compile that, given a propositional formula, prints on the
standard output a Caml function that has the behaviour of the formula. For example,
you should be able to have something like:

```caml
# compile (And(Or(Var "x", Var "y"), Or(Var "x", Var "z")));;
fun x y z → ((x) || (y)) && ((x) || (z))
- : unit
```

7 References

Up to now, we only used constants, and never mutable variables. So, how can we modify
a variable value? A possible answer could be that you should never use mutable variables
in Caml, but their use can sometimes be justified. To define a mutable variable (called
a reference), do:

```caml
# let x = ref(0);;
val x : int ref = {contents = 0}
```

You can assign and retrieve the value of a reference by using := and !:

```caml
# x := 1;;
# !x;;
- int = 1
```

Exercise 13. Translate the following functions into Caml by trying to stick as close as
possible to C style.

```caml
int factorielle (int n)
{
  int y = 1, res = 1;
  while (y <= n)
  {
    res *= y;
    y++;
  }
  return res;
}

int findmax(int[] array, int length)
{
  int max, i = 1;
  assert (length > 0);
  max = array[0];
  while (i < length)
  {
    if (array[i] > max)
      max = array[i];
  }
  return max;
}
```
Exercise 14. Rewrite these functions in Caml style (use a list instead of an array for findmax).