Programming Languages and Computation

Week 1: Functional Programming in OCaml

Through the exercises in this lab sheet, you will revise functional programming *and* learn a new language, *OCaml*. Although you won't be examined on *OCaml*, the example interpreter that you will study is presented in this language, so it is useful to have a little proficiency.

1 Setup

You will need *git* and a working OCaml development environment. If you are using the lab machines, *git* is available by default and you can obtain a development environment by executing the following command in a terminal:

```
$ module load ocaml
```

If you want to obtain an OCaml development environment on your own machine, follow the all instructions at https://ocaml.org/docs/installing-ocaml. Ensure you complete all three parts of the setup: installing *opam*, initialising *opam* and installing the platform tools.

Choose a convenient directory to work in and clone the git repository containing some starter code for the revision problems:

```
$ git clone https://github.com/uob-coms20007/starter.git
$ cd starter
```

This will download the code and drop you into the root directory for the project. This is a good time to load up your favourite text editor with the contents of the current directory. Visual Studio Code with the *OCaml Platform* extension is an excellent choice (you will need to install this extension if you don't already have it):

```
$ code .
```

The project that you obtained from Github is setup as a library, so all the source code is in the subdirectory *lib*. In this case, there is only a single file, which contains all the examples from this problem sheet. For example, it contains a definition of the *square* function:

```
let square x = x * x
```

Toplevel Usage

You can check that you have everything correct by running some example code from the OCaml toplevel. The toplevel can be loaded by the OCaml build manager, *dune*, which will ensure that the code and any prerequesite libraries are loaded according to the requirements of your project (which are extremely minimal in this case).

```
$ dune utop
```

You will be presented with an OCaml read-eval-print-loop (REPL), similar to Haskell's *GHCI*, into which you can input OCaml expressions and have them evaluated. It already knows about the code in the project, which is contained in the module *My* (in OCaml, each source code file implicitly determines a module of the same name). So, you can invoke the square function by:

```
# My.square 3;;
- : int = 9
```

The double semicolon is used in the toplevel, but not in source code files, as a mechanism to signal that you have input a complete expression. If you press enter without the double semicolon, then the toplevel will assume you want to continue writing the expression on a second line. On the following line, the toplevel reports that the expression was of type *int* and has a value of 9.

One difference from GHCI is that there is no good mechanism to *reload* your code into the toplevel if you subsequently modify it, which you will be doing next. So, every time you have finished experimenting with the current version of your code, you should exit the toplevel, either by pressing *Ctrl-D*, or by inputting the command:

```
# #quit;;
```

2 Functional Programming

OCaml is an impure functional programming language. Here *impure* pertains to the questionable rectitude of the language, since it allows for all sorts of wicked behaviour such as *mutation of variables*, *throwing of exceptions* and, perhaps most unforgivable, *printing of output to the console*. This also means it is quite easy to get things done.

Otherwise, OCaml is very similar to Haskell: programs consist of a collection of modules, modules consist of a collection of function definitions, functions are applied to their arguments by juxtaposition (writing the input after the function without any parentheses).

2.1 Definitions

Each module is a sequence of definitions, each of which is signified by the keyword **let**. For example, to define the square function in OCaml one can write:

```
let square x = x * x
```

Like Haskell, everything is statically typed, and there is no need for us to say what the types *are*, because the compiler will infer them for us. Unlike Haskell, if we want to assert what we think the types are, which is good documentation, we write them inline as in:

```
let double (x : int) : int = 2 * x
```

This asserts that `double` takes an input `x` of type `int` and returns an `int`, and its definition is `2 * x`. The names `double` and `square` are then available throughout the module, and outside of it by the qualified names `My.double` and `My.square`.

In Haskell, it is common to use a *where* clause to give a name to subcomputations within a definition. Such names are in scope only within the definition itself. The same thing is achieved in OCaml using *let-in*. For example, to avoid long lines of code, we could compute `(double (x + 4) - (square (double (square x))) + (5 * (square (double x)))` by naming the pieces:

```
let long_computation (x:int) : int =
  let p = double (x + 4) in
  let q = square (double (square x)) in
  let r = 5 * (square (double x)) in
  p - q + r
```

This makes the performance of OCaml functions easy to understand because execution will follow the order as the code is written: first p is computed, then q is computed, then r is computed, and finally p − q + r is computed.

It is perfectly possible to locally define functions, e.g.

```
let long_computation' (x : int) : int =
  let sq_db y = square (double y) in
  let p = double (x + 4) in
  let q = sq_db (square x) in
  let r = 5 * (sq_db x) in
  p − q + r
```

The **let** constructions are only for defining simple abbreviations. To define a recursive function (one whose definition mentions the very function being defined), one must use **let rec**. For example, one can define a function to compute non-negative integer powers of 2 by:

```
let rec exp2 (n : int) : int =
  if n = 0 then 1 else 2 * exp2 (n − 1)
```

Notice how the function being defined calls itself in the body. Writing recursive functions is one of the most important skills in functional programming. Mostly writing a recursive function comes down to two tasks: (i) describe how to compute the output in each base case, and (ii) describe how to compute the output in each recursive case *given the result of computing the output on a smaller input*. Usually (i) is quite straightforward. Here, the base case is when $n = 0$ and in this case we know that 2^n should be 1. For (ii) we get to use the recursive call on a smaller input, exp2 (n−1), and our task is to work out how to compute exp2 n *assuming that the call* exp2 (n−1) was successful in producing the output we expect. So, describing the recursive case comes down to figuring out how to use exp2 (n−1) as part of a description of how to compute exp2 n for $n > 0$.

1. Define the factorial function as a toplevel name fac. It should take an integer as input and return an integer as output according to the mathematical function given below:

$$fac(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * fac(n-1) & \text{otherwise} \end{cases}$$

(Here there is nothing to figure out, because the mathematical function that we are implementing in OCaml is already a recursive definition, it is just about getting the OCaml syntax correct).

Solution

```
let rec fac (n : int) : int =
  if n = 0 then 1 else n * fac (n−1)
```

2. Define a recursive function sum_squares, which takes an integer and returns an integer. It should be defined so that sum_squares n returns the sum of the squares of the numbers from 1 to n whenever n is a positive integer. For example sum_squares 3 returns 14.

Solution

```
let rec sum_squares (n : int) : int =
  if n = 1 then square n else square n + sum_squares (n−1)
```

It would also be fine to write this as:

```
let rec sum_squares (n : int) : int =
  if n = 0 then 0 else square n + sum_squares (n-1)
```

This latter is a different function from the former, but only for `sum_squares 0`, and the problem statement only cares about the behaviour "whenever n is a positive integer", so this case doesn't matter.

3. Define a recursive function `sum_digits`, which takes an integer and returns an integer. It should be defined so that `sum_digits n` returns the sum of the digits that comprise n whenever n is a non-negative integer. For example, `sum_digits 4372` returns 16. The definition should have the following shape:

```
let rec sum_digits (n : int) : int =
  if n <= 9 then
    n
  else
    ...
```

Here the idea is to recurse on the number of digits in n, so for n of shape $d_1 d_2 \ldots d_{k-1} d_k$ (each d_i being one of its digits) the recursive call will be to sum the digits of $d_1 d_2 \ldots d_{k-1}$, which can be written in OCaml as `n 10` since integer division rounds down. So, describing the recursive case means figuring out how to combine the following ingredients:

- The input, `n`, which we can think of as having shape $d_1 d_2 \ldots d_{k-1} d_k$ (and we know that $k > 1$ since $n > 9$).
- The already computed sum of the digits $d_1 d_2 \ldots d_{k-1}$, which is written `sum_digits (n / 10)`.

The binary operator **mod** is useful, since `p` **mod** `q` returns the remainder after dividing `p` by `q`.

Solution

```
let rec sum_digits (n : int) : int =
  if n <= 9 then
    n
  else
    let d1 = n mod 10 in
    let rest = sum_digits (n / 10) in
    d1 + rest
```

2.2 Basic Types

Integers Integers support the usual operators `+`, `-`, `*`, `/`, **mod**, and relations such as `<`, `<=` and so on. Other useful functions are available in the `Int` module: https://ocaml.org/manual/5.3/api/Int.html.

Characters The type `char` is the type of characters, whose values are written in single quotes, as in `'c'` and `'4'`. Some basic functions on characters are provided in the `Char` library https://ocaml.org/manual/5.3/api/Char.html.

Strings The type string is the type of strings, whose values are written in double quotes, as in "string value" or "foo". Strings support the binary operator for concatenation, written ^. Unlike Haskell, strings are *not* just lists of characters, and the ith character of a string s may be accessed by the constant time indexing operation s.[i]. Some other basic functions on strings are provided in the String library https://ocaml.org/manual/5.3/api/String.html.

Booleans The type bool is the type of booleans, whose values are written **true** and **false**. The usual boolean operators are supported, and written: &&, ||, and not. A few other functions are available in the Bool library https://ocaml.org/manual/5.3/api/Bool.html.

Moreover, booleans allow for conditional branching using if-expressions:

```
let absolute (n : int) : int =
  if n >= 0 then n else -n
```

Like Haskell, **if** .. **then** .. **else** is an expression, so it evaluates to a value, namely the value of the expression in the then-branch when the guard is true, or the value of the expression in the else-branch when the guard is false.

Unit The type unit is the type containing a single value (). The single value (), also pronounced unit, contains no useful information, so it is used as the return value when we only want a function for its (side-) effect on the world. In this sense it is similar to *void* in C. We will see later that e.g. print_endline "hello", which prints the string "hello\n" to stdout, returns unit, because we only call this function for its effect.

Tuples The type t_1 * t_2 * ... * t_k is the type of tuples whose i^{th} component is of type t_i. For example, (1, "foo") : int * string. For the special case of pairs (i.e. tuples with two components), the standard library function fst extracts the value in the first component and snd the value in the second.

Functions The type t1 -> t2 is the type of functions that map inputs of type t1 to outputs of type t2. Typically function values are defined implicitly using a name f and the **let** construction:

```
let f (x : t1) : t2 = ...
```

However, they can also be written explicitly using a kind of lambda notation. For example, the square function can be written explicitly as **fun** x -> x * x and the double function by **fun** x -> 2 * x. Function types can be nested in the return position to describe *curried* functions, that is, functions that accept more than one argument. For example, the type int -> char -> string is the type of functions that take an integer arguments and then a character argument and finally returns a string. Such a function can be defined by e.g.

```
let f (x : int) (y : char) : string = ...
```

Equality All types except functions support equality testing, which is written = (equals) and <> (not equals). References are equal whenever the values they currently store are equal. Beware: there is also an operator ==, but this not the usual kind of equality (it tests whether two expressions evaluate to the same piece of memory), so avoid it.

Polymorphism Like Haskell, OCaml support polymorphic types. Type variables are written prefixed by an apostrophe (also known as *tick*) character. For example, the identity function which always returns its input has type 'a -> 'a, meaning that it can take an input of any type and will return an output of the very same type.

```
let identity (x : 'a) : 'a = x
```

4. Define a function is_punctuation of type char -> bool which returns **true** just if its input is '!' or '?' and **false** otherwise.

Solution

```
let is_punctuation (c : char) : bool =
  c = '!' || c = '?'
```

5. Define a function exclaim of type string -> string such that exclaim s returns a new string which is identical to s except that every full stop (period) character has been replaced with a bang (exclamation mark). For example the expression:

```
exclaim "She passed away in her sleep aged 92. A service will be held next Tuesday."
```

returns "She passed away in her sleep aged 92! A service will be held next Tuesday!". Look into the standard library function String.map.

Solution

```
let exclaim (s : string) : string =
  let update_char c =
    if c = '.' then '!' else c
  in
  String.map update_char s
```

6. Define a function stop of type string -> unit such that stop s returns () and, as a side effect, prints out "STOP\n" to standard output once for each full stop character in s. Look into the standard library functions String.iter and print_endline.

Solution

```
let stop (s : string) : unit =
  let print_or_not c =
    if c = '.' then print_endline "STOP" else ()
  in
  String.iter print_or_not s
```

2.3 Mutable Variables and Other Effects

Mutable variables (i.e. those whose value can be changed by destructive update) are provided by *references*. The value **ref** e is a reference (i.e. mutable variable) whose initial value is e. Its type is t **ref** where t is the type of e. For example, **ref** 3 is a reference of type int **ref**. *Integer operations cannot be performed on integer references*. However, integer operations can be performed on the value currently stored in the reference, which can be extracted using the dereferencing operator,

written ! in prefix position. For example, !(ref 3) is just 3. In general, when e is of type t ref, then !e is of type t. To update the value stored in the reference, the operator := is used. Usually, references are defined by some local name and then referred to within the scope.

When dealing with such morally corrupt expressions as assignment and dereferencing, sequencing becomes especially important. The semicolon ; describes the evaluation of two expressions in sequence. Avert your eyes from the following depraved approach to counting the number of periods in a string:

```
let num_period (s : string) : int =
  let num = ref 0 in
  let update c =
    if c = '.' then num := !num + 1 else ()
  in
  String.iter update s;
  !num
```

The semicolon is *not* a line terminator (like in C), notice that there is no semicolon at the end of the last line. Rather you should think of it as a way to glue two expressions together. Moreover, it only makes sense when the first expression returns unit. If the first expression really returns some interesting (non-unit) value, then you should evaluate the sequence using let .. in so that you can give that intermediate output a name – use _ if you don't care about it. For example, reading two lines from standard input, throwing away the first and echoing the second, should be written:

```
let read_second_line () : unit =
  let _ = read_line () in
  let s = read_line () in
  print_endline s
```

A generalisation of the semicolon is the while loop, which executes its body over and over again in sequence until the guard evaluates to false.

```
let echo () : unit =
  let b = ref true in
  while !b do
    let s = read_line () in
    print_endline s;
    if s = "quit" then b := false else ()
  done
```

Note that such a loop can always be rewritten as a simple recusive function:

```
let rec echo () : unit =
  let s = read_line () in
  print_endline s;
  if s = "quit" then () else echo ()
```

The standard library function read_line : unit -> string will throw an exception if it encounters the end-of-file (EOF) token (e.g. when the user presses Ctrl-D). This exception can be caught using a try ... with expression. For example, an alternative version of the echo program that waits for EOF:

```
let rec echo () : unit =
  try
    let s = read_line () in
    print_endline s;
    echo ()
  with
```

```
  | End_of_file -> ()
```

The **with** clause is actually a form of pattern matching, which will we look at shortly.

7. Define a function unmatched_parens of type string -> int so that unmatched_parens s is the number of unmatched left parentheses '(' in the input string s. For example unmatched_parens "(as(f(gh)asd" returns 2, and both of unmatched_parens "asdg ea" and umatched_parens "asd(ges(bgd)as)fe)" return 0.
There are two nice ways to compute this:

   - In a pure way, using the standard library functions String.fold_left or String.fold_right.
   - In an impure way, using a mutable variable and the standard library function String.iter.

   Choose whichever you prefer.

Solution

The pure way:

```
let unmatched_parens (s : string) : int =
  let update n c =
    if c = '(' then
      n+1
    else if c = ')' then
      n-1
    else
      n
  in
  let count = String.fold_left update 0 s in
  if count < 0 then 0 else count
```

and the impure way:

```
let unmatched_parens (s : string) : int =
  let n = ref 0 in
  let update c =
    if c = '(' then
      n := n + 1
    else if c = ')' then
      n := n - 1
    else
       ()
  in
    String.iter update;
    if !n < 0 then 0 else !n
```

8. Define a random number guessing program which may be invoked as a function rand_guesser of type unit -> unit. The program should first choose a random integer between 0 and 9, and then allow the user unlimited attempts to guess what it is. After each attempt, if the attempt was successful the program should report that the guess was correct, how many guesses they took and then terminate (return unit), otherwise it should report whether the target is higher or lower than than the guess and await another guess. The following will be useful:

- To generate a random number between 0 and 9, first initialise the random number generator with `Random.self_init ()` and then use `Random.int 9`, which returns the target number. Full details can be found at https://ocaml.org/manual/5.3/api/Random.html.
- A sequence of expressions `e1; e2; e3` inside the branch of an if-expression must be put in a block to avoid parsing ambiguities, this looks like:

```
if ... then
  begin
    e1;
    e2;
    e3
  end
else
  ...
```

- To obtain the integer encoded in a string, use `int_of_string : string -> int`. To represent an integer as a string, use `string_of_int : int -> string`.

Solution

```
let rand_guesser () : unit =
  Random.self_init ();
  let n = ref 0 in
  let b = ref true in
  let target = Random.int 10 in
  while !b do
    n := !n + 1;
    let g = read_line () in
    let gn = int_of_string g in
      if gn = target then
        begin
          print_endline "Well done!";
          print_endline ("You required " ^ string_of_int !n ^ " guesses.");
          b := false
        end
      else
        if gn < target then
          print_endline "Go higher!"
        else
          print_endline "Go lower!"
  done
```

2.4 Recursive Datatypes and Pattern Matching

A key part of functional programming is the definition of recursive datatypes from constructors. These types are called *algebraic datatypes* in Haskell, and *variant types* in OCaml. These are introduced by defining a new datatype name using the **type** keyword alongside an enumeration of the constructors for the type and their argument type.

```
type suit = Hearts | Diamonds | Clubs | Spades
type rank = Num of int | Jack | Queen | King | Ace
type card = rank * suit
type comparison = Higher | Lower | Equal
```

There are two things to note. (i) The type definition for card is just a synonym – it doesn't introduce a new datatype, only another name for an existing datatype – you can tell this because it doesn't introduce any new constructors. (ii) The Num constructor takes a single argument of type int, all the others don't take any arguments.

There are exactly four values of the type suit, namely Hearts, Diamonds, Clubs and Spades. There are (pow2 Sys.int_size) + 4 values of the type rank, namely Num n for each Sys.int_size-bit integer *n*, Jack, Queen, King and Ace; though, in practice, we only care about 13 of them.

The best way to manipulate values of variant types is through pattern matching, which is done using the **match** ... **in** ... construction. For example, to convert the suits to a string:

```
let string_of_suit (s : suit) : string =
  match s with
  | Hearts -> "\u{2661}"
  | Diamonds -> "\u{2662}"
  | Clubs -> "\u{2667}"
  | Spades -> "\u{2664}"
```

When we convert the ranks to strings, we get to bind the argument of Num to a new variable, n, that we can use on the right-hand side:

```
let string_of_rank (s : rank) : string =
  match s with
  | Num n -> string_of_int n
  | Jack -> "J"
  | Queen -> "Q"
  | King -> "K"
  | Ace -> "A"
```

We can form a pattern out of any kind of value, except function values. For example, to construct the string representation of a card, we can match on pairs and name their components:

```
let string_of_card (c : card) : string =
  match c with
  | (r, s) -> string_of_rank r ^ string_of_suit s
```

Patterns can be nested, and the wildcard _ used whenever we don't care to name some component.

```
let value_card (c : card) : int =
  match c with
  | (Ace, _)  -> 1
  | (Num n, _) -> n
  | (Jack, _)  -> 11
  | (Queen, _) -> 12
  | (King, _)  -> 13
```

9. Define a function compare_cards of type card -> card -> comparison such that compare_cards c d returns Higher just if c has a higher value than d, Lower just if c has a lower value than d and Equal otherwise.

Solution

```
let compare_cards (c : card) (d : card) : comparison =
  if value_card c > value_card d then
    Higher
  else if value_card c < value_card d then
    Lower
  else
    Equal
```

Finally, note that every constant, including string literals, can be used as patterns and cases of the match that have the same outcome can be combined. The following function can be used to parse user input.

```
let comparison_of_string (s : string) : comparison =
  match s with
  | "l" | "L" | "lo" | "Lo" -> Lower
  | "h" | "H" | "hi" | "Hi" -> Higher
  | _                       -> failwith "Input was not of the correct shape."
```

Here we see a "catch-all" case _ which will match anything not matched by the patterns above. In that case, we use the standard library function failwith to raise a Failure exception.

Lists The type t list is the type of lists whose elements are all of type t. It is conceptually a variant type with two constructors, [] (pronounced *nil*, or *the empty list*) and (::) (pronounced *cons*). For convenience, cons is written as a binary operator, x :: xs is the list with head element x and tail xs. Despite these syntactic conveniences, nil and cons are just variant type constructors and so we may pattern match on them. For example, a recursive function to sum a list of integers:

```
let rec sum (ys : int list) : int =
  match ys with
  | [] -> 0
  | x :: xs -> x + sum xs
```

Lists support the append operator xs @ ys, which glues list ys to the end of list xs. A few other useful functions on lists are provided by the standard library https://ocaml.org/manual/5.3/api/List.html. We can model the standard deck as a list of cards:

```
let ranks =
  [Ace, Num 2, Num 3, Num 4, Num 5, Num 6, Num 7, Num 8, Num 9, Jack, Queen, King]
let deck =
  let heart_cards = List.map (fun r -> (r, Hearts)) ranks in
  let club_cards = List.map (fun r -> (r, Clubs)) ranks in
  let spade_cards = List.map (fun r -> (r, Spades)) ranks in
  let diamond_cards = List.map (fun r -> (r, Diamonds)) ranks in
  heart_cards @ club_cards @ spade_cards @ diamond_cards
```

10. Define a function shuffle of type card list -> card list such that shuffle cs returns a random permu-

tation of cs. A strategy for doing this is as follows:

(i) Initialise the random number generator.

(ii) Write a function that takes a card c as input and returns a pair (n,c) where n is a random integer (in the range 0 .. 52).

(iii) Using List.map, map this function over the standard deck to obtain a list of all cards, each paired up with a random integer.

(iv) Sort this list according to the random numbers paired with each card, this can be achieved by the function List.sort compare : int * card list -> int * card list.

(v) Use List.map and snd to throw away the random numbers attached to each card in the sorted list, leaving only the deck of cards sorted in some random order.

Solution

```
let shuffle (cs : card list) : card list =
  Random.self_init ();
  let random_num c = (Random.int 52, c) in
  let rcs = List.map random_num cs in
  let sorted_rcs = List.sort compare rcs in
  List.map snd sorted_rcs
```

11. Define a function hi_lo_game of type unit -> unit. The function should first shuffle the deck. Then present to the user and discard the top card of the current deck. Then, until the deck is depleted, it should do the following:

    - Ask the user whether the next card will be higher or lower and await user input.
    - Draw the next card and present it to the user.
    - If the user was correct in their prediction, increment a count of correct consecutive guesses, otherwise terminate the game and announce the count of correct consecutive guesses.

    If the deck becomes empty, print an appropriate message and terminate the game.

Solution

```
let hi_lo_game () : unit =
  let score = ref 0 in
  let deck_state = ref (shuffle deck) in
  let prev_card = ref (List.hd !deck_state) in
  let rec game_loop () =
    match !deck_state with
    | [] -> print_endline "Deck depleted, well done!"
    | d :: ds ->
      print_endline "Higher (h) or lower (l)?";
```

```
      let s = read_line () in
      let exp = comparison_of_string s in
      let act = compare_cards !prev_card d in
      print_endline ("The next card was: " ^ string_of_card d ^ ".");
      if exp = act then
        begin
          prev_card := d;
          deck_state := ds;
          score := !score + 1;
          print_endline "Well guessed!";
          game_loop ()
        end
      else
        begin
          print_endline "Sorry, bad guess!";
          print_endline ("Your score was " ^ string_of_int !score ^ ".")
        end
in
print_endline ("The card is " ^ string_of_card !prev_card ^ ".");
deck_state := List.tl !deck_state;
game_loop ()
```