# Foundations of Computer Science

*Supervision 3*

Dima Szamozvancev and Michael Lee

## 1 Functions as values

**Prerequisites**: Lecture 8

1. Consider the following polymorphic functions. Infer the types of `sw`, `co` and `cr` (without asking OCaml) and the give the definitions of `id`, `ap` and `ucr` based on their types. What do these functions do and what are their uses?

```
let sw f x y = f y x
let co g f x = g (f x)
let cr f a b = f (a,b)
val id  : 'a -> 'a = <fun>
val ap  : ('a -> 'b) -> 'a -> 'b = <fun>
val ucr : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

2. *Ordered types* are OCaml types `T` with a comparison operator `< : T -> T -> bool` such that `a < b` returns `true` if `a : T` is "smaller than" `b : T`. Many OCaml types – such as `string` and `int` – can be ordered and compared with the `<` operator in the obvious way. We often want to combine two such orderings to get comparison operators for compound types such as `string * int`. Two ways of doing this are *pairwise ordering*, which compares elements of the pair individually:

$$(x, y) <_p (x', y') \iff x < x' \land y < y'$$

and *lexicographic ordering*, which orders by the first elements, and if they are equal, by the second element (for an arbitrary number of elements we get the familiar word ordering used in dictionaries):

$$(x, y) <_\ell (x', y') \iff x < x' \lor (x = x' \land y < y')$$

   a) Write OCaml functions implementing pairwise and lexicographic ordering for the type of pairs `string * int`.

   b) Hardcoding the comparison operator `<` makes these functions a bit inflexible: for example, we cannot order a list of pairs in increasing order on the first element, but decreasing order on the second. We can make the functions more abstract by taking the comparison operators as higher-order arguments, and using them instead of `<`. Write two higher-order OCaml functions to perform pairwise and lexicographic comparison of values of type `'a * 'b`, where the comparison operators for types `'a` and `'b` are passed as arguments.

   c) Explain how you would use your functions in the previous part, and the higher-order sorting function `insort`, to sort a list of type `(string * (int * string)) list` according to

the following specification:

$$(s_1,(m,s_2)) < \left(s_1',(n,s_2')\right) \iff s_1 \leq s_1' \ \wedge \ \left(m > n \ \vee \ (m = n \ \wedge \ s_2 < s_2')\right)$$

3. Without using `map`, write a function `map2` such that `map2 f` is equivalent to the composition `map (map f)`. The obvious solution requires declaring two recursive functions. Try to get away with one by exploiting nested pattern-matching.

4. The built-in type `option`, shown below, can be viewed as a type of lists having at most one element. (It is typically used as an alternative to exceptions.) Declare an analogue of the function `map` for type `option`.

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

5. Consider the following program

```
let f () =
   let x = 1 in
   let g () = x in
   g

let h = f ()

let x = 2 in h ()
```

   a) What should the result of this program be?

   b) One way to implement variables is by looking up their values on the stack. Explain, with reference to the program above, why higher order functions need to be treated specially for this approach to work. *Hint:* is `x = 1` still on the stack when `h ()` is evaluated?

6. Recall the making change function of Lecture 4:

```
let rec change till amt = match till, amt with
  | _, 0  -> [ [] ]
  | [], _ -> []
  | c::till, amt ->
     if amt < c then change till amt else
       let rec allc = function
         | [] -> []
         | (cs::css) -> (c::cs) :: allc css
       in allc (change (c::till) (amt-c))
          @ change till amt
```

The function `allc` applies the function "cons a `c`" to every element of a list. Eliminate it by declaring a curried cons function and applying `map`.

## 2 Sequences and laziness

**Prerequisites**: Lecture 9

1. Consider the list function `concat`, which concatenates a list of lists to form a single list. Can it be generalised to concatenate a sequence of sequences? What can go wrong?

```
let rec concat = function
  | []    -> []
  | l::ls -> l @ concat ls
```

2. What is the time complexity of appending two lazy lists? What is the time complexity of getting the head of a lazy list? How do these complexities compare with their eager versions? When might it be desirable to use a lazy list over an eager list?

3. Memoisation involves storing the results of computation, so as to avoid recomputation. For example, whenever I try and compute

```
f x
```

   I first look up if I have already performed the computation, and if I have, I just look up the result. Otherwise, I compute the result, and then store it in a table for future lookup.

   a) Write a program that tests if memoisation is being performed.
   b) With reference to an example, like lazy append, explain why memoisation is important for lazy lists.

4. Why are lazy lists (sequences) useful not "natively" supported by OCaml? How do we simulate lazy lists in OCaml and why does that not have the issues you described above?

5. Code an analogue of `map` for sequences.

6. A *lazy binary tree* is either empty or is a branch containing a label and two lazy binary trees, possibly to infinite depth. Present an OCaml datatype to represent lazy binary trees, along with a function that accepts a lazy binary tree and produces a lazy list that contains all of the tree's labels. The order of the elements in the lazy list does not matter, as long as it contains all potential tree nodes.

7. Code the lazy list whose elements are all ordinary lists of zeroes and ones, namely `[]`, `[0]`, `[1]`, `[0;0]`, `[0;1]`, `[1;0]`, `[1;1]`, `[0;0;0]`, ....

8. (Continuing the previous exercise.) A palindrome is a list that equals its own reverse. Code the lazy list whose elements are all palindromes of 0s and 1s, namely `[]`, `[0]`, `[1]`, `[0;0]`, `[0;0;0]`, `[0;1;0]`, `[1;1]`, `[1;0;1]`, `[1;1;1]`, `[0;0;0;0]`, ... You can use the list reversal function `List.rev`.

# 3 Queues and search strategies

1. Suppose that we have an implementation of queues, based on binary trees, such that each operation takes logarithmic time in the worst case. Outline the advantages and drawbacks of such an implementation compared with one presented in the notes.
2. The traditional way to implement queues uses a fixed-length array. Two indices into the array indicate the start and end of the queue, which wraps around from the end of the array to the start. How appropriate is such a data structure for implementing breadth-first search?
3. Why is iterative deepening inappropriate if $b \approx 1$, where b is the branching factor? What search strategy would make more sense in this case?
4. The version of a queue from the notes performs normalisation whenever `hds` is empty. Another alternative is to perform normalisation whenever `hds` is shorter than `tls`.

   a) Implement this variant queue

   b) Assume we have a queue `q`, where the length of `hds` is the same as the length of `tls`, and call this length `n`.
   Analyse the time complexity of the following program.

   ```
   for i = 1 to n do
     let _, new_q = deq q
   end
   ```

   c) In the earlier example, each `deq` potentially involves a normalisation:

   ```
   append hds (reverse tls)
   ```

   Now consider using lazy lists for `hds` and `tls`. Assume an `appendq` function for appending two queues. Define `reverseq` as follows:

   ```
   let reverseq xf = fun () ->
     let rec reverse xf acc = match xf with
         | Nil -> acc
         | Cons(x, xf) -> reverse (xf ()) (Cons(x, acc))
     in reverse xf Nil
   ```

   That is, `reverseq xf` is a suspended computation, that, once called, is executed all at once. Explain why, with memoisation, repeated normalisation

   ```
   appendq hds (reverseq tails)
   ```

   might be more efficient in this case. What about without memoisation?

5. An interesting variation on the data structures seen in the lectures is a *deque*, or *double-ended queue* [1]. A deque supports efficient addition and removal of elements on both ends (either its front or back). Suggest a suitable implementation of deques, justifying your decision.
6. Implement deques as an abstract data type in OCaml (including a type declaration and suitable operations). Estimate the amortised complexity of your solution.
7. Write a version of the function `breadth` shown on Page 88 using a nested `let` construction rather than `case`.
8. Mathematical sets can be treated as an abstract data type for an unordered collection of unique elements. One approach we may take is to represent a set as an *ordered list* without duplicates: $\{5, 3, 8, 1, 18, 9\}$ would become the OCaml list `[1;3;5;8;9;18]`. Code the set operations of membership test, subset test, union and intersection using this ordered-list representation. Remember that you can assume the ordering invariant for the inputs (and thereby make your functions more efficient), and your output should maintain this invariant.

## 4 Elements of procedural programming

1. a) Assume `e` and `e'` are OCaml commands of type `int`. Is it true that `e + e'` is always the same as `e' + e`? Either explain why it is true, or write down `e` and `e'` that will exhibit different behaviour.
   b) What if `e` and `e'` are expressions?
2. Comment, with examples, on the differences between an `int ref list` and an `int list ref`. How would you convert between the two?
3. What is the effect of `while (C1; B) do C2 done`? Where would such a formulation be useful?
4. Write a version of function power (Page 10) using `while` instead of recursion.
5. Write a function to exchange the values of two references, `xr` and `yr`.
6. Arrays of multiple dimensions are represented in OCaml by arrays of arrays. Write functions to (a) create an $n \times n$ identity matrix, given $n$, and (b) to transpose an $m \times n$ matrix. Identity matrices have the following form:

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

## 5 Optional questions

1. Code a function to make change using lazy lists, delivering the sequence of all possible ways of making change. Using sequences allows us to compute solutions one at a time when there exists an astronomical number. Represent lists of coins using ordinary lists. (*Hint*: to benefit

[1] Deque is usually pronounced "deck", which is convenient because a deck of cards is a good example of a double-ended queue – computer scientists are often way too clever with naming things.

from laziness you may need to pass around the sequence of alternative solutions as a function of type `unit -> (int list) seq`.)

2. With some exceptions (such as appending or concatenation), we can adapt many common list operations to work on lazy lists. In addition, lazy lists can be used to generate infinite sequences without the risk of nontermination. This exercise explores some interesting examples using the `seq` type.

   a) Define the analogues of `filter` and `zipWith` for sequences. The `zipWith` list functional is similar to `zip` but it applies a binary function to the pair it constructs. For example, `zipWith (+) [1;2;3;4] [5;6;7;8] = [6;8;10;12]` where `(+)` is the function version of the `+` operator.

   ```
   val filterS  : ('a -> bool) -> 'a seq -> 'a seq = <fun>
   val zipWithS : ('a -> 'b -> 'c) -> 'a seq -> 'b seq -> 'c seq
                = <fun>
   ```

   b) Consider the following two definitions. What do they represent and how do they work?

   ```
   let s = let rec s_aux (Cons (x, xf)) =
                          Cons (x * x, fun () -> s_aux (xf()))
           in s_aux (from 0)

   let rec f = Cons (0, fun () ->
               Cons (1, fun () -> zipWithS (+) f (tail f)))
   ```

   c) Define the lazy list `sieve : int seq` that uses the Sieve of Eratosthenes to calculate the infinite sequence of prime integers. *Hint*: Define an auxiliary function `sieve_aux` such that `sieve = sieve_aux (from 2)`. You may want to use `filterS` for the "sieving".