Foundations of Computer Science
Solution Notes for Example Sheet 3

In this document you will find some solution notes for the problems of Example Sheet 3 of Foundations of Computer Science. If you find any mistake or any typos, please do let me know. Also, I am happy to hear (and include them in the notes (with credit) if you want) about alternative solutions to the problems or variations of a problem that you came up with.
This supervision looks into the OCaml support for functions, common patterns for functions, lazy lists, search strategies, the stack and queue data structures and imperative programming.

1 Lecture 8

Exercise 1 [Higher-order function]
(a) What is a *higher-order* function?
(b) Why is it useful that OCaml supports higher order functions?

(a) A *higher-order function* is a function whose input is another function or returns a function as its result.

(b) There are several advantages:

- It allows for code reuse (so we have to write and maintain less code), as we can write one sorting function that works for any comparison function. Note that polymorphism alone would only allow us to sort any list but using the same order.
- It allows for shorter code. For example, compare `List.map (fun x -> x + 1) ls` and

```
for (int i = 0; i < ls.size(); ++i) {
  ls[i] += 1;
}
```

- It allows to capture several code patterns making code easier to read and perhaps easier for the compiler to optimise. For example, in the map example above, an optimised compiler could note that `fun x -> x + 1` is a pure function, so it can parallelise its execution over the elements of the list.

Discussion:

- A (very) popular paradigm in distributed computing is known as MapReduce. At its core, it implements the operations `List.map` and `fold` (see below) over a large number of machines. (This video might be helpful and this is one of the first presentations of the implementation used at Google)
- Various programming languages have adopted functional programming features including functions. For example, Java and C++ support lambda expressions.

Exercise 2 [Anonymous functions]
(a) What is the syntax for *anonymous functions* in OCaml?
(b) Why are they useful?

(a) Anonymous functions are defined using the syntax `fun x1 ... xn -> e`.

(b) They are useful to define functions with short bodies, e.g. when passing them as arguments to other functions such as 'map' or 'filter', possibly making the code easier to read.

Discussion: *Are anonymous functions necessary?* No, you can do everything with named functions. For example, you can replace `... fun x -> e ...` with `let myfun x = e in ... myfun ...` .

Exercise 3 [Curried functions]
(a) How many arguments do OCaml functions take?

(b) How does OCaml "support" functions with multiple arguments? Give examples for this.

(c) Is `npower` (from the first lecture) a curried function? What other "reasonable" types could a function with equivalent behaviour have?

(d) What is the syntax for *function application*? Explain the error you get when evaluating `f 2 3`, where `let f x = x + 3`.

(e) Write a function `convert_4` that takes a function `g : ('a * 'b * 'c * 'd) -> 'e` and returns a curried function for `g`.

(a) As we have said in previous supervisions, OCaml functions take one argument.

(b) OCaml supports "multiple" arguments by allowing functions to return other functions. So, if we have a function `f x1 x2` (with type `T1->T2->T3`), then this is evaluated as `(f x1) x2`. Then `f x1` returns a function with type `T2->T3`, i.e. takes an argument of type `T2` and returns the final value (of type `T3`).

An alternative to having a function `let f x1 x2 = e` is to have a function `let f (x1, x2) = e`.

(c) Yes, it is a curried function with type `float->int->float`. If we use the pair approach, we would get a function with type `(float->int)->float`.

(d) The syntax for function application is `(e1 e2)`.

The order of function application is `(f 2) 3`. The type of `f 2` is `int` since `f` is `int->int` and we apply it with `int` (of value 2). So, we are trying to apply an int to an int (something like `(2 3)`), which is not allowed.

On older versions of the OCaml compiler you would get the following error: `Error: This expression has type int This is not a function; it cannot be applied.`

But on newer versions of the OCaml compiler, you would get the following (more sophisticated) error: `Error: This function has type int -> int. It is applied to too many arguments; maybe you forgot a ';'.`

(e) This is a function that returns a curried function with 4 arguments

```
let convert_4 g a b c d = g (a, b, c, d);;
```

This is condensed version of

```
let convert_4 g = fun a -> fun b -> fun c -> fun d -> g (a, b, c, d);;
```

and

```
let convert_4 g = fun a b c d -> g (a, b, c, d);;
```

For example, for the function

```
let add (x, y, z, w) = x + y + z + w;;

(* Returns a curried function. *)
convert_4 add;;
```

Discussion:

- *When would you prefer using tuples instead of curried functions?* There is no hard rule. If partial application makes sense for the function, then it might be better to define it as curried (e.g. `List.sort (compare) ...`).
- How would you write the reverse function that converts a curried function to a function accepting a tuple as input? `let reverse_4 g (a, b, c, d) = g a b c d;;`.

Exercise 4 [Partial application]

(a) What is *partial application*?

(b) What functions result from partial application of the following curried functions?

i. `let plus i j = i + j`

ii. `let lesser a b = if a < b then a else b`

iii. `let pair x y = (x, y)`

iv. `let equals x y = x = y`

(c) Is there any practical difference between the following two declarations of the function `f`? Assume that the function `g` and the curried function `h` are given.

i. `let f x y = h (g x) y`

ii. `let f x = h (g x)`

(a) In OCaml, *partial application* is function application on a function that returns another function.

(b) i. Adds a constant to the given value.

ii. Compares the given value to a constant.

iii. Creates a pair with the given value and a constant.

iv. Checks equality of the given value to a constant.

(c) No there is no difference. Only that `h` is required to return a function.

Exercise 5 [Sorting] How does sorting (e.g. `List.sort` benefit from being able to pass *functions as values*? What is the type of a sorting function taking a comparison function as an argument? [**Note:** Pay attention to the order of the arguments]

(a) (Optional) What rules should the ordering function obey?

From the documentation page:
`val sort : ('a -> 'a -> int) -> 'a list -> 'a list`
Sort a list in increasing order according to a comparison function. The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see Array.sort for a complete specification). For example, compare is a suitable comparison function. The resulting list is sorted in increasing order. List.sort is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.
The current implementation uses Merge Sort. It runs in constant heap space and logarithmic stack space.
Passing functions as values allows for sorting to be done with different comparison functions. The order of the arguments is important because usually the comparison function is fixed and the given list changes.
(optional) The function should obey the following,

- If `f a b = 0` and `f b c = 0`, then `f b a = 0` and `f a c = 0`. (So `f x y = 0` forms an equivalence relation)
- (Reflexivity) `f a a = 0`.
- (Anti-Symmetry) `(f a b) = - (f b a)`.
- (Transitivity) If `f a b < 0` and `f b c < 0`, then `f a c < 0`.

Discussion:

- *What if you wanted to keep the list fixed and change the comparison function?* You can create a function that swaps the arguments, e.g. `let swap_sort x y = List.sort y x`.
- *Can you make this general?* `let swap f x y = f y x;;` *What is the type of the function?* `('a->'b->'c)->'b->'a->'c`.
- *Why does OCaml use a comparison function of type `('a -> 'a -> int)` instead of type `('a -> 'a -> bool)`?* The first function returns `0` if the elements are equal. With a boolean return value to determine if `a` and `b` are equal, we need to call both `f a b` and `f b a`. As we saw this is useful in the quicksort implementation.

Exercise 6 [Map]

(a) What does the *map* function do?

(b) Use map for the following:

i. Replace every negative element of a list of integers with 0.

ii. Add 1 to every element in the list.

iii. Truncate all lists in a list, so that they have 3 or fewer elements.

iv. Append an item to all lists in a list.

(a) The map function applies one function to all elements of a list.

(b) Answers:

i.
```
# List.map (fun x -> if x < 0 then 0 else x) [1;-10;4;3;-2;-5];;
- : int list = [1; 0; 4; 3; 0; 0]
```

or

```
# List.map (max 0) [1;-10;4;3;-2;-5];;
- : int list = [1; 0; 4; 3; 0; 0]
```

ii.
```
# List.map (fun x -> x + 1) [1;-10;4;3;-2;-5];;
- : int list = [2; -9; 5; 4; -1; -4]
```

or

```
# List.map ((+) 1) [1;-10;4;3;-2;-5];;
- : int list = [2; -9; 5; 4; -1; -4]
```

iii.
```
# List.map (take 3)
    [[1;2;3;4];[5;6;7;8;9;10];[11;12];[14;15;16];[17;18;19;20]];;
- : int list list =
[[1; 2; 3]; [5; 6; 7]; [11; 12]; [14; 15; 16]; [17; 18; 19]]
```

iv.
```
# List.map (fun x -> 3::x)
    [[1;2;3;4];[5;6;7;8;9;10];[11;12];[14;15;16];[17;18;19;20]];;
- : int list list =
[[3; 1; 2; 3; 4]; [3; 5; 6; 7; 8; 9; 10]; [3; 11; 12]; [3; 14; 15; 16]; [3;
    17; 18; 19; 20]]
```

or

```
# List.map (List.cons 3)
    [[1;2;3;4];[5;6;7;8;9;10];[11;12];[14;15;16];[17;18;19;20]];;
- : int list list =
[[3; 1; 2; 3; 4]; [3; 5; 6; 7; 8; 9; 10]; [3; 11; 12]; [3; 14; 15; 16]; [3;
    17; 18; 19; 20]]
```

Discussion:

- *Why do we give a name to such a simple function?* The map function captures a very important pattern in computation: applying the same function to each element in a collection. If this function is pure (i.e. it has no side effects), then it can be applied in parallel. This is the basis of the map-reduce paradigm in distributed computation.
- *How can we get a simpler implementation for the `all_cons` function from Supervision 2?*

```
let all_cons x xs = List.map (List.cons x) xs;;
```

Exercise 7 Complete **[2016P1Q1 (a),(b)]**.

(See solution notes)

(a) i. Passed as arguments to other functions

ii. Returned as results.

iii. Put into lists, datatypes, etc.

iv. Can be expressed using anonymous functions.

A curried function returns another function as its result, giving the effect of multiple arguments.

`exists` can make use of `||` in its implementation.

(b) The function computes `f(x1, f(x2, ... f(xn, e)))`. This is called fold right.

Discussion:

- *Is fold-left always equal to fold-right? Can you give an example?* No, they are not always equal. For example, for the power function, `pow(pow(a, b), c)` is not equal to `pow(a, pow(b, c))`.
- (Connection to discrete mathematics) *What property must they satisfy to be equal?* They must be associative. Or `zarg ls = zarg (List.rev ls)`. Also, `e` would be the identity element.
- *What is the type of `zarg`?* `('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`.
- *Should we use `foldl` whenever possible?* No, because it is not tail-recursive (e.g. for addition).

Exercise 8 [Predicates]

(a) What is a *predicate* (in OCaml)?

(b) How is `exists` defined? Give an example.

(c) How is `filter` defined? Give an example.

(a) A predicate is a function with type `'a->bool` (i.e. a boolean function).

(b)
```
let rec exists p = function [] -> false | x::xs -> p x || exists p xs;;
```

Discussion: *What is the type of the function?* `exists : ('a -> bool) -> 'a list -> bool`

(c)
```
let rec filter p = function [] -> [] | x::xs -> if p x then x :: filter p xs
    else filter p xs;;
```

Discussion: *What is the type of the function?* `filter : ('a -> bool) -> 'a list -> 'a list`

Exercise 9 [Function composition]

(a) How is *function composition* defined? Write an OCaml function that takes two functions and returns their function composition. What is its type?

(b) How are these different?

```
compose (fun x -> x + 1) (fun y -> y * 7)
compose (fun y -> y * 7) (fun x -> x + 1)
```

(c) Give equivalent single function definitions for these two function compositions?

(a) The function composition of f and g is defined as $h(x) = f(g(x))$.

```
let compose  f g = fun x -> f (g(x));;
```

or equivalently

```
let compose  f g x = f (g(x));;
```

(b) One is equal to $7x + 1$ and the other is $7(x + 1)$. These have different values, e.g. at $x = 0$.

(c) `let f1 x = 7 * x + 1` and `let f2 x = 7 * (x + 1)`.

Exercise 10 [Function iteration] The k-th iterate of a function `f : 'a -> 'a` denoted by $f^k(x)$, is the application of f to x, k times. For example $f^2(x) = f(f(x))$ and $f^3(x) = f(f(f(x)))$. Write an OCaml function that takes a function and a positive integer k that returns the k-th iterate of the function.

```
let rec iter f = function
  0 -> (fun x -> x)
| k -> (fun x -> (iter f (k-1)) (f x) );;
```

or

```
let rec f_iter f = function
1 -> f
n -> comp f (f_iter f (n-1));;
```

Discussion:

- *Does you function work for $k = 0$?*
- *What is the time complexity if you evaluate `(iter 100 (fun x -> x + 1)) 20`?* It takes 100 steps, so compared to normal addition it is slower.

Exercise 11 Show how to replace any expression of the form `List.map f (List.map g xs)` by an equivalent expression that applies `List.map` only once.

[Source: OCamlWP 5.12]

You can use:

```
List.map (compose f g) xs;;
```

Discussion:

- *What is the type of `List.map List.flatten`?* `('a list list list) -> 'a list list`.
- What is the type of `List.map List.map`?

```
# List.map List.map;;
- : ('_a -> '_b) list -> ('_a list -> '_b list) list
```

Exercise 12 [Matrices]

(a) Explain how matrices can be represented using lists. Is there a problem with that?

(b) Explain how to implement `transpose` using `map`. What is the time complexity of your implementation?

(c) Explain how to implement matrix multiplication using `map`. What is the time complexity of your implementation?

(a) Matrices can be represented as `float list list`. The outer list contains the rows of the matrix as lists. For example, `[[1;2;3];[4;5;6];[7;8;9]]` represents the following matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

The main problem with this representation is that accessing the (i, j)-th element takes $\mathcal{O}(i + j)$ time. Another smaller problem is that we need to verify that each row has the same number of elements, so we will get a lot of match warnings.

(b)
```
let rec transpose = function
| []::_ -> []
| rows -> (List.map (List.hd) rows)::(transpose (List.map (List.tl) rows));;
```

Testcases:

```
transpose [[1;2;3];[4;5;6];[7;8;9]];;

# transpose [[1;2;3];[4;5;6];[7;8;9]];;
- : int list list = [[1; 4; 7]; [2; 5; 8]; [3; 6; 9]]
```

The time complexity is $O(N \cdot M)$.

(c)
```
(* From ML version of the course *)
let rec dotprod xx yy = match xx, yy with
  [], [] -> 0.
| (x::xs), (y::ys) -> x *. y +. dotprod xs ys;;

let matprod a_rows b_rows =
   let cols = transpose b_rows
   in List.map (fun row -> List.map (dotprod row) cols) a_rows;;
```

Testcases:

```
matprod [[1.;2.;3.];[4.;5.;6.];[7.;8.;9.]] [[2.; 0.]; [1.; 1.]; [0.; 2.]];;

# matprod [[1.;2.;3.];[4.;5.;6.];[7.;8.;9.]] [[2.; 0.]; [1.; 1.]; [0.; 2.]];;
- : float list list = [[4.; 8.]; [13.; 17.]; [22.; 26.]]
```

The time complexity is $O(N \cdot M \cdot K)$, where the first matrix has dimensions $N \times M$ and the second $M \times K$.

Discussion:

- *Did you give the time complexity in terms of all three dimensions?*
- If we have multiple multiplications to perform, then their order matters. Actually, finding the order of multiplying matrices is an interesting problem of its own. (see matrix chain multiplication in the Part IA Algorithms course)
- *Look at the `List.module` and find a function that would make the code for `dotproduct` shorter.*

```
List.fold_left2 (fun z x y -> x *. y +. z) 0. [1.; 2.; 3.] [10.;20.;30.];;
```

- The optimal time complexity for efficient matrix multiplication is currently an open problem with several theoretical and practical applications. The currently (2021) most efficient algorithm performs matrix multiplication in $\mathcal{O}\left(n^{2.37286}\right)$ time (see here). It is conjectured that it can be done in $\mathcal{O}\left(n^{2+\epsilon}\right)$ for any constant small $\epsilon > 0$.

Exercise 13 [List module]

(a) Go through the functions in the List.Module (you may skip "Association lists" and "Iterators").

(b) How would you implement `flatten`, `for_all`, `mapi` and `exists2`? Give examples of how you would use these functions. How do your implementations differ from the reference implementations.

(c) Look carefully at the documentation for a few of these functions. What features do you notice? Do you find the documentation useful? Is it better to search on stackoverflow for examples than to look at the documentation?

(a) Nothing to do.

(b)
```
let rec flatten = function
  [] -> []
| l::r -> l @ flatten r
```

Discussion: *How would you write flatten using the fold functions?*

```
let flatten l = List.fold_right List.append l []
(* Note: We need to fold right otherwise it is slow. *)

let rec for_all p = function
    [] -> true
  | a::l -> p a && for_all p l

let rec mapi i f = function
    [] -> []
  | a::l -> let r = f i a in r :: mapi (i + 1) f l

let mapi f l = mapi 0 f l
```

Discussion: *What is the problem with the following implementation?*

```
let rec mapi f = function
|[] -> []
|x::xs -> (f 0 x)::(mapi (fun a b -> f (a+1) b) xs);;
```

On the n-th iteration it calls the functions n times, so it is quadratic in n.

```
let rec exists2 p l1 l2 =
  match (l1, l2) with
    ([], []) -> false
  | (a1::l1, a2::l2) -> p a1 a2 || exists2 p l1 l2
  | (_, _) -> invalid_arg "List.exists2"
```

Discussion:

- *Do we need* ***exists2****? Can we not just use the* ***zip*** *function?* We could but it would be memory inefficient.
- *Does a language need a standard library?* There are many benefits to having a standard library because it allows to share code with others and using the same components. If your program has 10 different libraries and each of them has its own definition of options, list functions, etc, it will be tedious to make them work together.

(c) There are various features such as:

- Exceptions thrown
- Implementation details (tail recursive or not)
- Version in which it was added
- Time/space complexity of the implementation

The documentation is like a contract between the implementor of the function and the user of a function, detailing which implementation characteristics the user should expect.

As you may know stackoverflow is useful for many things, such as more complicated programming tasks that are not present in the standard library. Also, sometimes the documentation may not contain sufficient detail (or is written in a weird way) and so someone on stackoverflow may give more details. However, information on stackoverflow usually comes with little guarantees (e.g. will this behaviour remain in the next version? is it the same in all implementations? and so on).

2 Lecture 9

Exercise 14 [Delayed vs Lazy] What is the difference between *delayed* and *lazy* evaluation?

Delayed evaluation is a programming style where instead of returning the result, we return a function that returns the result. Lazy evaluation is evaluation that happens only when the value is requested and it is a built-in language feature.
For example, we could have `let add3 a b c = if a > b then a + b else c` Then calling for `add3 (4 + 2) 4 (32/0)`, evaluates a and b since they are needed in the comparison and then `c` is not evaluated because it is not needed. However, in OCaml the arguments are first evaluated and then the function body is computed. Haskell implements lazy evaluation. OCaml implements eager evaluation.
Discussion: *You can implement lazy evaluation in Java and C++ using abstract classes. How would you do it?*
Here is an example implementation in Java.

```
package llist;

/* One possible way to implement lazy lists. */
public interface LazyList<T> {

   /* Returns the current element of the lazy list, i.e. the x in Cons(x, ... ). */
```

```
    T getCurrent();

    /* Returns the tail of the lazy list, i.e. the y in Cons(..., y). */
    LazyList<T> getTail();

}
```

And the instantiation for example to generate all lazy binary sequences.

```
package llist;

import java.util.ArrayList;
import java.util.List;

/* This is an immutable version of LazyLists. */
public class LazyBinarySeqs implements LazyList<List<Integer>> {

    private final ArrayList<Integer> current;

    public LazyBinarySeqs() {
        this.current = new ArrayList<>();
        current.add(0);
    }

    private LazyBinarySeqs(ArrayList<Integer> ls) {
        this.current = ls;
    }

    @Override
    public List<Integer> getCurrent() {
        return current;
    }

    @Override
    public LazyBinarySeqs getTail() {
        return new LazyBinarySeqs(addOne(current));
    }

    public static ArrayList<Integer> addOne(ArrayList<Integer> ls) {
        ArrayList<Integer> arr = new ArrayList<>();
        boolean foundZero = false;
        for (int i = 0; i < ls.size(); ++i) {
            if (foundZero) {
                arr.add(ls.get(i));
                continue;
            }
            if (ls.get(i) == 0) {
                foundZero = true;
                arr.add(1);
            } else {
                arr.add(0);
            }
        }
        if (!foundZero) arr.add(0);
        return arr;
    }

}
```

And the main function to generate the first 10 sequences.

```
package llist;

public class Main {
```

```
    public static void main(String[] args) {
        LazyBinarySeqs binarySeqs = new LazyBinarySeqs();
        for (int i = 0; i < 10; ++i) {
            System.out.println(binarySeqs.getCurrent());
            binarySeqs = binarySeqs.getTail();
        }
    }

}
```

It is also possible to reuse the same class in Java (i.e. there is no need to make each lazy sequence immutable).

```
package llist;

import java.util.ArrayList;
import java.util.List;

public class MutableLazyBinarySeqs implements LazyList<List<Integer>> {

    private ArrayList<Integer> current;

    public MutableLazyBinarySeqs() {
        this.current = new ArrayList<>();
        current.add(0);
    }

    @Override
    public List<Integer> getCurrent() {
        return current;
    }

    @Override
    public MutableLazyBinarySeqs getTail() {
        current = LazyBinarySeqs.addOne(current);
        return this;
    }
}
```

Exercise 15 [Unit type]

(a) What is the *unit type* and what is its syntax?

(b) What are the uses of unit in OCaml?

(a) The unit type is the type of `()`.

(b) The unit type can be used to implement delayed evaluation. Note that void functions also take an argument. Operations with side-effects will usually have unit return type.

Discussion: *Note that we can also implement lazy lists as* `type 'a llist = Nil | Cons of 'a * (int -> 'a llist)`*, but then we would need to pass in an integer in order to get a result, e.g.* `let getTail(Cons(_, y)) = y 0;;`.

Exercise 16 [Lazy lists] Write brief notes on programming with lazy lists in OCaml. Your answer should include the definition of a polymorphic type of infinite lazy lists, a function to return the tail of a lazy list, a function to create the infinite list of all positive integers, and an apply-to-all functional analogous to the list functional `map`.

[Source: [2015P1Q2]]

Exercise 17 [From] Explain why the following forms of `from` and `get` are wrong:

(a) `let rec wrongfrom1 k = Cons(k, wrongfrom1(k+1));;`

(b) `let rec wrongfrom2 k = Cons(k, fun () -> wrongfrom2 (n + 1));;`

(c) `let rec get n xx = match n, xx with 0, _ -> [] | n, (Cons(x, xs)) -> x :: get (n-1) xs();;`

(d) `let rec get n xx = match n, xx with 0, _ -> [] | n, (Cons(x, xs)) -> x :: get (n-1) xs;;`

(a) Compile-time error because the function call does not type check to `unit-> a' llist`.

(b) Just the parameter 'n' is wrong.

(c) The compiler considers `()` to be another argument to the `get` function.

(d) Missing the `()`.

Exercise 18 Declare a function to add adjacent elements of a sequence, transforming $[x_1; x_2; x_3; x_4; \ldots]$ to $[x_1 + x_2; x_3 + x_4; \ldots]$.

[Source: OCamlWP 5.30]

```
let rec add_adj (Cons(x, fx)) = let Cons(y, fy) = fx() in
   Cons(x + y, fun () -> add_adj (fy()) );;

let rec from n = Cons(n, fun () -> from (n+1));;
let rec take (Cons(x, y)) n = if n > 0 then x::take (y()) (n-1) else [];;
take (add_adj (from 1)) 10;;
- : int list = [3; 7; 11; 15; 19; 23; 27; 31; 35; 39]
(* which is equal to [1+2; 3+4; 5+6; 7+8; 9+10; 11+12; 13+14; 15+16; 17+18; 19+20] *)
```

Exercise 19 [Interleave] What is the problem with appending two infinite lists? How does interleave solve it?

The problem with appending two infinite lists it that by enumerating them we will never reach the end of the first list, hence we will never read any elements from the second list.
Discussion:

- *What does it mean for an element to be present in a lazy list?* There must exist an index n at which we will encounter the element.
- *What if we had three lazy lists?* We can use `interleave` more than once. For example, `interleave (interleave x1 x2) x3`. This corresponds to vertically traversing the lists. This extends to any finite number of lists. *Does this extend to an infinite number of lists?* No, look at Exercise **??**.

Exercise 20 [Lazy binary tree (++)]

(a) A lazy binary tree either is 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.

(b) Present an OCaml function that produces a lazy binary tree whose labels include all the integers, including the negative integers.

(c) Present an OCaml function that accepts a lazy binary tree and produces a lazy list that contains all of the tree's labels

[Source: [2008P1Q5]]

Look at the markscheme. Here are some further comments.

(a) `type 'a btree = Lf | Br of 'a * (unit -> 'a btree) * (unit -> 'a btree);;`

(b) The markscheme gives a simple solution that contains all integers. If you want to have each integer exactly once, then you can do the following:

```
type 'a btree = Lf | Br of 'a * (unit -> 'a btree) * (unit -> 'a btree);;

let rec all_ints_aux n sign =
  Br(sign * n, (fun () -> all_ints_aux (2 * n) sign), (fun () -> all_ints_aux
      (2 * n + 1) sign));;

let all_ints = Br(0, (fun () -> all_ints_aux 1 (-1)), (fun () -> all_ints_aux 1
    1));;
```

Testcases:

```
let Br(v1, l1, r1) = all_ints;;
let Br(v2, l2, r2) = l1 ();;
let Br(v3, l3, r3) = r1 ();;
let Br(v4, l4, r4) = l2 ();;
let Br(v5, l5, r5) = r2 ();;
let Br(v6, l6, r6) = l3 ();;
let Br(v7, l7, r7) = r3 ();;
```

Discussion: *Which data structure does this remind you of?* Functional arrays.

Exercise 21 [All binary lists (++)]

(a) 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]; ...`.

(b) 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 may take the reversal function `List.rev` as given. (*Hint:* First think how you would generate palindromes of even length.)

[Exercise 9.5 & 9.6 in Lecturer's handout]

See the handout on "Making functions lazy".

Exercise 22 [Nested infinite lists (+++)]

(a) Write a function `diag` that takes a lazy list of lazy lists,

$$\begin{bmatrix} [& [z_{11}; & z_{12}; & \ldots], & & \\ & [z_{21}; & z_{22}; & \ldots], & [z_{31}; & z_{32}; & \ldots], \ldots] \end{bmatrix}$$

and returns the diagonal, namely the lazy list $[z_{11}; z_{22}; z_{33}; \ldots]$.

(b) Write a function that takes two lazy lists $[x_1; x_2; x_3; \ldots]$ and $[y_1; y_2; y_3; \ldots]$ and a function f of two arguments; and returns a lazy list of lazy lists like above, with $z_{ij} = f x_i y_j$.

(c) Write a function that converts a lazy list of lazy lists like above to a lazy list whose elements are all of the z_{ij} , enumerated in some order.

[Source: [2015P1Q2]]

See markscheme for full solution. Some notes (and alternative implementations) follow below. We start with some tedious preliminaries that will help us test the implementations:

```
type 'a llist = Nil | Cons of 'a * (unit -> 'a llist);;

let rec from n = Cons(n, fun () -> from (n+1));;

let double = Cons(
   Cons(1, fun () -> Cons(2, fun () -> Cons(3, fun () -> Cons(4, fun () -> Nil)))),
   fun () -> Cons(
       Cons(5, fun () -> Cons(6, fun () -> Cons(7, fun () -> Cons(8, fun () ->
           Nil)))),
       fun () -> Cons(
```

```
            Cons(9, fun () -> Cons(10, fun () -> Cons(11, fun () ->  Cons(12, fun ()
                -> Nil)))),
            fun () -> Cons(
                Cons(13, fun () -> Cons(14, fun () -> Cons(15, fun () -> Cons(16, fun
                    () -> Nil)))),
                fun () -> Nil))));;

let long_2D = Cons(from 0, fun () -> Cons(from 10, fun () -> Cons(from 100, fun ()
    -> Cons(from 1000, fun () -> Cons(from 10000, fun () -> Nil)))));;

let rec take (Cons(x, xs)) n = if n = 0 then [] else x :: take (xs()) (n-1);;
```

(a)
```
let rec nth n (Cons(x, xs)) = if n = 0 then x else nth (n-1) (xs());;

let diag l =
  let rec diag_prime (Cons(y, ys)) d = Cons(nth d y, fun () -> diag_prime
      (ys()) (d+1)) in
    diag_prime l 0;;
```

Time complexity: $\mathcal{O}(n^2)$.

Alternative implementation:

```
let rec applytoall f = function
Cons (x,xs) -> Cons(f x, fun() ->(applytoall f (xs())));;

let removefirst = function
| Cons(x,xs) -> (xs());;
let removefromall lis = applytoall removefirst lis;;
let rec diag = function
| Cons(x,xs) -> (match x with Cons(a,la) -> Cons(a,fun()-> diag (removefromall
    (xs()))));;
```

Diagonal Implementation testcases:

```
take (diag double) 3;;;

# take (diag double) 3;;;
- : int list = [1; 6; 11]
```

(b) **Apply to all implementation:**

```
let rec cross_single f x (Cons(y, ys)) = Cons(f x y, fun () -> cross_single f x
    (ys()));;
let rec cross f (Cons(x, xs)) ys = Cons(cross_single f x ys, fun () -> cross f
    (xs()) ys);;

let rec seq_map2 f xl yl = seq_map (fun x -> seq_map (f x) yl) xl
```

Apply to cross tests:

```
let a = take (cross (fun x y -> (x, y)) (from 1) (from 100)) 5;;
take (List.hd a) 5;;
take (List.hd (List.tl a)) 5;;

# take (List.hd a) 5;;
- : (int * int) list = [(1, 100); (1, 101); (1, 102); (1, 103); (1, 104)]
# take (List.hd (List.tl a)) 5;;
- : (int * int) list = [(2, 100); (2, 101); (2, 102); (2, 103); (2, 104)]
```

(c) **Enumerate all entries:**

Method 1:

There are two steps in this approach:

1. Create a method that returns the (n, m)-th element.

2. Map the integer sequence to pairs of numbers.

```
let rec nth n (Cons(x, xs)) = if n = 0 then x else nth (n-1) (xs());;
let multi_nth xs (n1, n2) = nth n1 (nth n2 xs);;

let convert n =
   let x = int_of_float (( (-1.) +. sqrt (8. *. (float_of_int n) +. 1.))/.2.) in
   let d = n - x * (x+1)/2 in
      (d, x - d);;

let rec mapl f (Cons(x, xs)) = Cons(f x, fun () -> mapl f (xs()));;

let enumerate xs = mapl (multi_nth xs) (mapl convert (from 0));;
```

Tests:

```
take (mapl convert (from 0)) 10;;
take (enumerate double) 5;;
take (enumerate long_2D) 10;;

# take (mapl convert (from 0)) 10;;
- : (int * int) list =
[(0, 0); (0, 1); (1, 0); (0, 2); (1, 1); (2, 0); (0, 3); (1, 2); (2, 1);
 (3, 0)]
# take (enumerate double) 5;;
- : int list = [1; 5; 2; 9; 6]
#
  take (enumerate long_2D) 10;;
- : int list = [0; 10; 1; 100; 11; 2; 1000; 101; 12; 3]
```

Method 2: Create a `get_next` method as described in approach 1 in "Making lazy functions".

```
let rec get_next (x, y) = let (xx, yy) = if x = 0 then (y+1, 0) else (x-1, y+1)
    in
  Cons((x, y), fun () -> get_next (xx, yy));;
let enumerate xs = mapl (multi_nth xs) (get_next (0,0));;
```

Exercise 23 [Lazy enumeration of change (+++)] 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 from laziness you may need to pass around the sequence of alternative solutions as a function of type `unit -> (int list) seq`.)

[Exercise 9.3 in Lecturer's handout]

Look at the "Making functions lazy" handout (approach 3) for more details.

```
let rec seqChange = function
|  (coins, coinvals, 0, coinsf) -> Cons(coins, coinsf)
|  (coins, [], amount, coinsf) -> coinsf()
|  (coins, c::coinvals, amount, coinsf) ->
      if amount < 0 then coinsf()
      else seqChange(c::coins, c::coinvals, amount - c,
            fun () -> seqChange(coins, coinvals, amount, coinsf))
```

Note: To test if it is indeed lazy once can trace `seqChange` and call it once. If it evaluates long, then it is not lazy..

```
# get 10 (seqChange ([], [1;2;3;4;5], 10, fun () -> Nil));;
- : int list list =
[[1; 1; 1; 1; 1; 1; 1; 1; 1; 1]; [2; 1; 1; 1; 1; 1; 1; 1; 1];
 [3; 1; 1; 1; 1; 1; 1; 1]; [2; 2; 1; 1; 1; 1; 1; 1]; [4; 1; 1; 1; 1; 1; 1];
 [3; 2; 1; 1; 1; 1; 1]; [5; 1; 1; 1; 1; 1]; [2; 2; 2; 1; 1; 1; 1];
 [4; 2; 1; 1; 1; 1]; [3; 3; 1; 1; 1; 1]]
```

```
# get 10 (seqChange ([], [1;2;3;4;5], 6, fun () -> Nil));;
- : int list list =
[[1; 1; 1; 1; 1; 1]; [2; 1; 1; 1; 1]; [3; 1; 1; 1]; [2; 2; 1; 1]; [4; 1; 1];
 [3; 2; 1]; [5; 1]; [2; 2; 2]; [4; 2]; [3; 3]]
```

3 Lecture 10

Exercise 24 [Queues] Write brief notes on the *queue* data structure and how it can be implemented efficiently in OCaml. In a precise sense, what is the cost of the main queue operations? (It is not required to present OCaml code.)

[Source: [2014P1Q2]]

Discussion:

- *Is amortised complexity always good enough?* Not always. For some real-time processing it might not be. There exists a functional programming queue implementation which runs in worst case $\mathcal{O}(1)$. Alternatively, one could use the functional array and get a worst-case $\mathcal{O}(\log n)$, where n is the number of elements in the queue.
- *Why does the code in the lectures normalise the queue even on insert?*

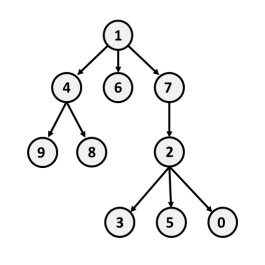
Exercise 25 [Queue example] Show the internal state of the (efficient OCaml) queue after each of the following operations: `push 1`, `push 2`, `push 3`, `pop`, `push 4`, `pop`, `push 5`, `push 6`, `pop`, `pop`, `pop`, `pop`.

```
1::[] []
2::1::[] []
3::2::1::[] []
[] 2::3::[]
4::[] 2::3::[]
4::[] 3::[]
5::4::[] 3::[]
6::5::4::[] 3::[]
6::5::4::[] []
[] 4::5::6::[]
[] 5::6::[]
[] 6::[]
[] []
```

Exercise 26 [Stacks] Write brief notes on the *stack* data structure. How can it be implemented in OCaml?

This is essentially a list. It supports push and pop operations using `::`.

Exercise 27 [BFS/DFS] Explain how *BFS* and *DFS* works. For each case, what is the order that the nodes are traversed?



DFS: First explores a node, and then starts by exploring all of its children one by one. The order is: `1 4 9 8 6 7 2 3 5 0`.

BFS: Explores the nodes in the order of the levels. The order is: `1 4 6 7 9 8 2 3 5 0`.
Note: The graph that we are searching may not explicitly exist. In the jugs problem the states are created from previous ones.
Discussion: *Does a DFS/BFS work for a graph with cycles?* Without any changes, they do not work. We also need to keep track of the states that we have visited, otherwise the searches will result in infinite loops.

Exercise 28 [Iterative Deepening]
(a) What is the main issue with BFS?

(b) How does *depth-first iterative deepening search* solve this? Derive its space and time complexity.

(a) The main issue with BFS is that it has to keep track of all nodes in the previous level. This number could be very large (and we are often limited by memory).

(b) Iterative deepening depth-first works by performing a DFS search of up to a certain depth, starting with depth 1, then with depth 2, then with depth 3 and so on. This turns out to have the same asymptotic complexity as the BFS for a branching factor greater than 1.

Assuming a constant branching factor $b > 1$, the number of states visited in d iterations is,

$$\begin{aligned}
&1 + (1+b) + (1+b+b^2) + \ldots (1+b+\ldots+b^{d-1}) \\
&\quad = \frac{b-1}{b-1} + \frac{b^2-1}{b-1} + \ldots + \frac{b^d-1}{b-1} \\
&\quad = \frac{1}{b-1} \cdot ((b-1) + (b^2-1) + \ldots + (b^d-1)) \\
&\quad = \frac{1}{b-1} \cdot ((1+b+b^2+\ldots+b^d) - (d+1)) \\
&\quad = \frac{1}{b-1} \cdot \left(\frac{b^{d+1}-1}{b-1} - (d+1) \right) \\
&\quad = \left(\frac{b^{d+1} - 1 - (d+1)\cdot(b-1)}{(b-1)^2} \right) \\
&\quad = b^d \cdot \frac{1}{(b-1)^2} - \frac{1 + (d+1)\cdot(b-1)}{(b-1)^2} \\
&\quad = \mathcal{O}\left(b^d + d\right) = \mathcal{O}\left(b^d\right),
\end{aligned}$$

since b is constant (so is $\frac{1}{(b-1)^2}$) and b^d beats d for large d.

Discussion: *What happens when $b = 1$?* In this case, the graph is just a line. So, it is better to run DFS once. Otherwise the time complexity will be quadratic.

Further Reading 1 [More on making lazy programs] Read the handout on "Techniques for generating lazy sequences". We will probably cover some of the material there in the revision session.

Further Reading 2 [More on searching for solutions] Read the handout on "Brief notes on complete search techniques". We will probably cover some of the material there in the revision session.

4 Lecture 11

Only attempt exercises in this section if the lecturer covered them.

Exercise 29 What are the guarantees that *pure* functions provide in contrast to *non-pure* functions? What are any reasons for introducing non-pure functions in a program?

Pure functions do not have side-effects, meaning that if you call the same function with the same argument, then you get the same result.
There are a few advantages to this:

- For example, the fact that `f` is pure means that we can apply `f(x)` to each element in parallel. (You may want to look at the map-reduce paradigm).
- It makes it easier to reason about programs, because we do not have to keep track of global state.
- We could cache functions calls with the same arguments. For example, Haskell uses memoisation to speed up (even exponentially programs). E.g. `fib(n) = fib(n-1) + fib(n-2)`.

Discussion: *Do you think one could implement memoisation on languages without pure functions?* In some languages it might be possible, if we also compare the global state with the previous global state (or the state accessed by the function), but this will probably entail huge memory requirements. Python has some support for this. If you are interested look here.

Exercise 30 [References] What is the syntax and types for *references* in OCaml?

- `!r` : dereferences the reference at `r`. Regarding types, if `r : ref T`, then `!r : T`.
- `r := e` : sets the value in `r` equal to the value of `e`. This expression has unit type.
- `ref e` : if `e : T` then creates a reference of type `T` and sets the value of the expression.

Note that `r` can be an expression. (e.g. `!(if 3 > 4 then p else q)`)

Exercise 31 [Swap] Write an OCaml function to exchange the values of two references `xr` and `yr`.

[Exercise 12.4 in Lecturer's handout]

```
let swap xr yr =
  let tmp = !xr in
    xr := !yr;
    yr := tmp;;
```

Testcases:

```
let a = ref 1;;
let b = ref 2;;
swap a b;;

# a;;
- : int ref = {contents = 2}
# b;;
- : int ref = {contents = 1}
```

Exercise 32 [While]

(a) What is the syntax for *while loops* in OCaml?

(b) Implement `fact`, `npow` and `foldl` using while loops in OCaml.

(c) Write an imperative version of `fib`.

(a) The syntax is

```
while e1 do e2 done;;
```

where `e1` has type bool and `e2` can have any type.

Discussion:

- *Why does OCaml give you a warning when `e2` does not have a unit type?* Because that value is not returned and in case there are no iterations in the loop it can only return the unit type.
- *Can you swap two integers without an extra variable?* `A = A xor B`, `B = B xor A`, `A = A xor B`. (But the extra space is allocated by the compiler). The `xor` between ints in OCaml is `Int.logxor`.

(b) **Fibonacci:**

```
let fact n =
   let r = ref n in
   let acc = ref 1 in
      while !r > 0 do
         acc := !acc * !r;
         r := !r - 1
      done;
      !acc;;
fact 5;;
```

Naive power:

```
let npow n x =
   let acc = ref 1.0 in
   let it = ref n in
      while !it > 0 do
         acc := !acc *. x;
         it := !it - 1
      done;
      !acc;;
npow 5 2.0;;
```

Fold left:

```
let foldl f xs x =
   let ans = ref x in
   let cur_list = ref xs in
      while !cur_list <> [] do
         ans := f (List.hd !cur_list) !ans;
         cur_list := List.tl !cur_list
      done; !ans;;

foldl (fun x -> fun y -> x + y) [1; 10; 20; 50] 0;;
foldl (fun x -> fun y -> x @ y) [[1;10]; [20;30;32]; [44]] [];;

val foldl : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
# foldl (fun x -> fun y -> x + y) [1; 10; 20; 50] 0;;
- : int = 81
# foldl (fun x -> fun y -> x @ y) [[1;10]; [20;30;32]; [44]] [];;
- : int list = [44; 20; 30; 32; 1; 10]
```

(c)
```
let fib n =
   if n = 0 then 1
   else if n = 1 then 1
   else
      let prev1 = ref 1 in
      let prev2 = ref 1 in
      let it = ref n in
      while !it > 0 do
         let tmp = !prev2 in
            prev2 := !prev1;
            prev1 := tmp + !prev2;
            it := !it - 1
      done;
      !prev1;;

# fib 10;;
- : int = 144
```

Exercise 33 [Mutable lists]
(a) Describe how *mutable lists* are implemented in OCaml.

(b) Write the `nth` OCaml function.

(c) Write an OCaml function `update` that takes a list `x`, a position `i` and a value `v`, and sets the `i`-th element of the list to `v`.

(a) Mutable lists in OCaml consist of a value and a reference to the rest of the list. This means that we can modify the reference to the tail. The type is given by

```
type 'a mlist =
| Nil
| Cons of 'a * 'a mlist ref;;
```

(b)
```
let rec nth (Cons(x, xs)) n =
  if n = 0 then x
  else nth (!xs) (n-1);;
```

Testcases:

```
let ls0 = Cons(30, ref Nil);;
let ls1 = Cons(20, ref ls0);;
let ls2 = Cons(10, ref ls1);;
let ls3 = Cons(0, ref ls2);;

nth ls3 2;;
```

(c)
```
let rec update (Cons(x, xs)) n new_val =
   if n = 1 then
      let Cons(y, ys) = !xs in
         xs := Cons(new_val, ys)
   else update (!xs) (n-1) new_val;;
```

Testcases:

```
ls3;;
update ls3 2 100;;
ls3;;

# ls3;;
- : int mlist =
Cons (0,
 {contents =
   Cons (10,
    {contents = Cons (20, {contents = Cons (30, {contents = Nil})})})})
# update ls3 2 100;;
- : unit = ()
# ls3;;
- : int mlist =
Cons (0,
 {contents =
   Cons (10,
    {contents = Cons (100, {contents = Cons (30, {contents = Nil})})})})
```

Exercise 34 [Revisiting all tails] Provide example code (and output) to demonstrate that the result returned by `all_tails` (e.g. `[1;2;3;4]`, `[2;3;4]`, `[3;4]`, `[4]`) occupies linear (to the length of the original list) space.

```
let rec all_tails = function [] -> [] | (x::xs as ls) -> ls :: all_tails xs;;
(* Construct a reference to some value, say 10 *)
let k = ref 10;;
(* Construct a list that uses this reference. *)
let r = all_tails [ref 1; ref 2; ref 3; k];;
(* By changing the contents of the reference, all tails will be updated.
```

```
   So we can deduce that the lists are shared (and are not independent copies). *)
k := 11;
r;;

# let rec all_tails = function [] -> [] | (x::xs as ls) -> ls :: all_tails xs;;
val all_tails : 'a list -> 'a list list = <fun>
# all_tails [1;2;3;4];;
- : int list list = [[1; 2; 3; 4]; [2; 3; 4]; [3; 4]; [4]]
# let k = ref 10;;
val k : int ref = {contents = 10}
# let r = all_tails [ref 1; ref 2; ref 3; k];;
val r : int ref list list =
  [[{contents = 1}; {contents = 2}; {contents = 3}; {contents = 10}];
   [{contents = 2}; {contents = 3}; {contents = 10}];
   [{contents = 3}; {contents = 10}]; [{contents = 10}]]
# k := 11;
  r;;
- : int ref list list =
[[{contents = 1}; {contents = 2}; {contents = 3}; {contents = 11}];
 [{contents = 2}; {contents = 3}; {contents = 11}];
 [{contents = 3}; {contents = 11}]; [{contents = 11}]]
```