Foundations of Computer Science
Example Sheet 3

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?

Exercise 2 [Anonymous functions]

(a) What is the syntax for *anonymous functions* in OCaml?

(b) Why are they useful?

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`.

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)`

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?

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.

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

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.

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?

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.

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]

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?

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?

2 Lecture 9

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

Exercise 15 [Unit type]

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

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

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;;`

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]

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

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]]

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]

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]]

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]

3 Lecture 10

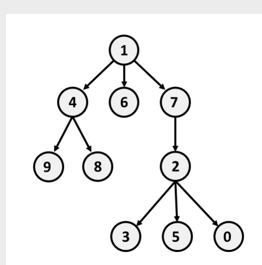
Exercise 24 [Queues] Write brief notes on the *queue* data structure and how it can be implemented efficiently in OCaml. In a precise sense, what is the cost of the main queue operations? (It is not required to present OCaml code.)

[Source: [2014P1Q2]]

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`.

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

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



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.

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?

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

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

[Exercise 12.4 in Lecturer's handout]

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`.

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`.

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.