Foundations of Computer Science
Solution Notes for Example Sheet 2

In this document you will find some solution notes for the problems of Example Sheet 2 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.

1 Lecture 4

Exercise 1 [Wildcard operator] What does the *wildcard pattern* do? Why is it useful? Is it a best practice to use whenever possible?

The wildcard pattern matches any value, as does any variable name in a pattern-matching expression. The benefit of using it is to increase code readability. The reader of the code immediately understands that the value will not be used, e.g., `let hd x::_ = x`.
It is also useful for catching errors. If some argument is not used, then a warning is output. If you are always using wildcards then it help you catch bugs, e.g., in the code `let f x1 x2 = x1 + x1;;`.
However, there are also cases when it is arguable if removing it makes the code more readable.

```
let bool_to_int = function
  false -> 0
| true  -> 1;;
```

Discussion:

- In some coding styles, unused variables have to be explicitly named as `unusedA`.

Exercise 2 [Take and Drop] Write OCaml functions for `take` and `drop` and describe their operation.

There are several possible implementations.
Implementation without exceptions: Same as the one in the lecture notes

```
let rec take i = function
| [] -> []
| x::xs ->
   if i > 0 then x :: take (i - 1) xs
   else []
```

Implementation with exceptions:

```
let rec take i = function
| [] ->
   if i = 0 then []
   else failwith "Invalid argument" # Could also split into other failing modes
       (negative argument or insufficient items)
| x::xs ->
   if i > 0 then x :: take (i - 1) xs
   else if i = 0 then []
   else failwith "Invalid argument";;

let rec take i ls =
   if i < 0 then failwith "Negative index"
   else if i = 0 then []
   else if ls = [] then failwith "Requesting more items than there are"
   else (List.hd ls) :: take (i-1) (List.tl ls)
```

Testcases:

```
# take 2 [1;2;3;4;5];;
- : int list = [1; 2]
# take 0 [];;
- : 'a list = []
# take 1 [];;
Exception: Failure "Requesting more items than there are".
# take (-1) [1;2;3];;
Exception: Failure "Negative index".
# take 5 [1;2;3];;
Exception: Failure "Requesting more items than there are".
```

Discussion:

- There exists a similar implementation with options.
- *Which one is better?* Calling the `take` function with negative argument makes no sense and it might be hiding a bigger error. On the other hand, allowing to take more elements than are available could simplify code in some implementations.
- *What are the types of these functions?* `int -> 'a list -> 'a list`.
- *Why is it implemented as a curried function?* So that it can be partially applied e.g., as part of `map`.
- *Why did you choose this order of the arguments?* Maybe it makes more sense for the list to be first(?) Depends on the application.

Similarly, for the `drop` function,
Implementation with exceptions:

```
let rec drop i ls =
   if i < 0 then failwith "Negative index"
   else if i = 0 then ls
   else if ls = [] then failwith "Requesting more items than there are"
   else drop (i-1) (List.tl ls);;
```

Testcases:

```
# drop 2 [1;2;3;4;5];;
- : int list = [3; 4; 5]
# drop 0 [1;2;3];;
- : int list = [1; 2; 3]
# drop 0 [1;2];;
- : int list = [1; 2]
# drop 0 [];;
- : 'a list = []
# drop (-2) [1;2;3];;
Exception: Failure "Negative index".
# drop 10 [1;2;3];;
Exception: Failure "Requesting more items than there are".
```

```
drop 2 [1;2;3;4;5];;
drop 0 [1;2;3];;
drop 2 [1;2];;
drop 0 [];;
drop (-2) [1;2;3];;
drop 10 [1;2;3];;
```

Exercise 3 [Cyclic rotation] A cyclic rotation of a list by one position moves the first element to the end of the list. Similarly, we define a cyclic rotation by k positions. For example, cyclically rotating `[1; 2; 3; 4; 5; 6; 7]` by two positions gives `[3; 4; 5; 6; 7; 1; 2]`. Write an OCaml function that performs a cyclic rotation by k positions.

```
let cyclic x k = (drop k x) @ (take k x);;
```

```
(* Implementation being cautious at the edge cases. *)
let cyclic x k =
   if k < 0 then raise (Invalid_argument "k must be non-negative")
   else if k = 0 then x
   else
      let len = List.length x in
      let rlen = k mod len in
         (drop rlen x) @ (take rlen x);;
```

Testcases:

```
# cyclic [1;2;3;4;5] 3;;
- : int list = [4; 5; 1; 2; 3]
# cyclic [1;2;3;4;5] 0;;
- : int list = [1; 2; 3; 4; 5]
# cyclic [1;2;3;4;5] 20;;
- : int list = [1; 2; 3; 4; 5]
# cyclic [1;2;3;4;5] 23;;
- : int list = [4; 5; 1; 2; 3]
# cyclic [1] 23;;
- : int list = [1]

cyclic [1;2;3;4;5] 3;;
cyclic [1;2;3;4;5] 0;;
cyclic [1;2;3;4;5] 20;;
cyclic [1;2;3;4;5] 23;;
cyclic [1] 23;;
```

Discussion:

- Questionable: What should happen if the length is negative and what should happen if the list is empty (probably fail with an error).
- *What is the time complexity of your implementation?* It takes $\mathcal{O}(n)$ time if implemented with mod. Otherwise it may take $\mathcal{O}(n+k)$.
- *What is the memory complexity?* $\mathcal{O}(n)$ since take and drop have worst case $\mathcal{O}(n)$ memory.

Exercise 4 [nth] Write an OCaml function to find the n-th element of a list.

Implementation with exceptions:

```
let rec nth n l =
match n, l with
| _, [] -> failwith "Empty list"
| 0, x::_ -> x
| _, _::xs -> nth (n-1) xs;;

let rec nth n ls =
   if n < 0 then failwith "Negative index requested"
   else if n > 0 then nth (n-1) (List.tl ls)
   else if ls = [] then failwith "Index greater than list length"
   else List.hd ls;;
```

Testcases:

```
# nth 2 [0;1;2;3;4;5];;
- : int = 2
# nth 2 [0;1;2];;
- : int = 2
# nth 3 [0;1;2];;
Exception: Failure "Index greater than list length".
# nth (-1) [0;1;2];;
Exception: Failure "Negative index requested".
# nth 0 [];;
```

```
Exception: Failure "Index greater than list length".

nth 2 [0;1;2;3;4;5];;
nth 2 [0;1;2];;
nth 3 [0;1;2];;
nth (-1) [0;1;2];;
nth 0 [];;
```

Exercise 5 [Binary search (+)] You are given a function `f(n)` that is increasing. For example, `let f n = nth [1; 4; 6; 10; 14; 18; 21] n`. Write a function `close` that searches for the largest `n` such that `f(n) < v`. This function will take as arguments `f`, `v` and a lower and an upper bound for the answer (`l` and `u`). For `v = 8`, in the example above, the answer would be `2`, which corresponds to `f(2) = 6` (the call would be `close f v 0 (len[1; 4; 6; 10; 14; 18; 21])`.

```
let close f v mn mx =
   let rec helper b e =
      if b = e then b
      else let mid = (b + e + 1) / 2 in
         if f mid >= v then helper b (mid-1)
         else helper mid e
   in helper mn mx;;

let f n = nth n [1; 4; 6; 10; 14; 18; 21];;

# close f 0 0 6;;
- : int = 0
# close f 8 0 6;;
- : int = 2
# close f 5 0 6;;
- : int = 1
# close f 10 0 6;;
- : int = 2

close f 0 0 6;;
close f 8 0 6;;
close f 5 0 6;;
close f 10 0 6;;
```

Discussion:

- *What is the time complexity of this method?* It makes $\mathcal{O}(\log(u - \ell))$ queries to the function evaluation.
- How would you use this method for finding the square root of a value?

  ```
  let sqrt x = close (fun x -> x * x) (x+1) 0 x;;
  ```

- What would you do if you did not know the upper bound? You can binary search for the upper bound by guessing $\ell + 2^0$, then $\ell + 2^1$, $\ell + 2^2$ and so on to find u and then do the binary search.
- What if you don't know both the lower and the upper bound? You cannot find the answer more efficiently than looping through the values.

Exercise 6 [Member function] Write an OCaml function that takes a list and an element, and returns true if the element is present, or false if it is not.

```
let rec member v = function
| [] -> false
| x::xs -> x = v || member v xs;;
```

Discussion:

- What is the time complexity of this function? In the worst-case this takes $\mathcal{O}(n)$ time (e.g., when the element is near the end of the list). In the best-case it takes $\mathcal{O}(1)$ time (e.g., when the element is within the first few elements).
- Did you use *orelse*?

Exercise 7 [Sublist function] Write an OCaml function that takes two lists A and B and returns whether A appears in B. For example, `[1; 2; 3]` appears in `[1; 2; 1; 2; 3; 2; 1]`, but `[1; 3]` does not appear in `[2; 4; 5]`. What is the worst-case time complexity of your implementation?

Implementation:

```
let rec take i = function
| [] -> []
| x::xs ->
   if i > 0 then x :: take (i - 1) xs
   else []

let rec sublist x yy = match yy with
    [] -> x = []
  | y::ys ->
      if x = take (List.length x) yy then true
      else sublist x ys;;
```

Optimised implementation:

```
let rec are_prefix_equal = function
  [], _ -> true
| _, [] -> false
| x::xs, y::ys -> x = y && are_prefix_equal (xs, ys);;

let rec sublist x = function
  [] -> x = []
| y::ys as yy -> are_prefix_equal (x, yy) || sublist x ys;;

# sublist [1;2;3] [4;8;1;2;9;1;2;3;9;8];;
- : bool = true
# sublist [] [1;2;3];;
- : bool = true
# sublist [1;2;3] [1;2;4;1;2;8;3];;
- : bool = false
# sublist [1;2;3] [];;
- : bool = false
# sublist [1;2;3]
    [4;8;1;2;9;1;2;9;8;4;8;1;2;9;1;2;9;8;4;8;1;2;9;1;2;9;8;4;8;1;2;9;1;2;9;8];;
- : bool = false
# sublist [1;2;1;2;3] [1;2;1;2;1;2;3];;
- : bool = true

sublist [1;2;3] [4;8;1;2;9;1;2;3;9;8];;
sublist [] [1;2;3];;
sublist [1;2;3] [1;2;4;1;2;8;3];;
sublist [1;2;3] [];;
sublist [1;2;3]
    [4;8;1;2;9;1;2;9;8;4;8;1;2;9;1;2;9;8;4;8;1;2;9;1;2;9;8;4;8;1;2;9;1;2;9;8];;
sublist [1;2;1;2;3] [1;2;1;2;1;2;3];;
```

In the worst-case, this takes $\mathcal{O}(n_1 \cdot n_2)$, where n_1 and n_2 are the lengths of the two lists. In the best case, it takes $\mathcal{O}(\min(n_1, n_2))$. There are some more efficient algorithms for pattern matching in $\mathcal{O}(n_1 + n_2)$, e.g., using KMP (or the Z-algorithm).

Discussion:

- *Why use the && and || symbols?* Because this makes the implementation faster. In particular, if there is a character mismatch then the rest of the strings are not matched.

- *How would you write code to check if a list is a subsequence of another list?* This can be done linearly in $\mathcal{O}(n_1 + n_2)$.

Exercise 8 [List intersection]

(a) Write an OCaml function that computes the intersection of two lists.

(b) What is the type of your function?

(c) What is the time complexity of your implementation?

(d) (+) If the given lists were sorted, would you be able to speed up the algorithm?

(a) There is a little bit of vagueness as to whether repeated elements should be included more than once. Ideally, yes.

Implementation in $\mathcal{O}(n_1 \cdot n_2)$ time:

```
let rec mem_idx v pos = function
  [] -> -1
| x::xs ->
   if x = v then pos
   else mem_idx v (pos+1) xs;;

let rec drop i = function
| [] -> []
| x::xs ->
if i > 0 then drop (i-1) xs
else x::xs

let rec intersect xx y = match xx with
  [] -> []
| x::xs ->
   let x_pos = mem_idx x 0 y in
      if x_pos = -1 then intersect xs y
      else x :: intersect xs (take x_pos y @ drop (x_pos+1) y);;
```

Implementation in $\mathcal{O}(n_1 + n_2)$ time:

```
let rec increasing_intersect xx yy = match xx, yy with
| [], _ -> []
| _, [] -> []
| x::xs, y::ys ->
   if x = y then x :: increasing_intersect xs ys
   else if x < y then increasing_intersect xs yy
   else increasing_intersect xx ys;;

let intersect x y = increasing_intersect (List.sort compare x) (List.sort
    compare y);;
```

Testcases:

```
# intersect [1;2;3] [4;3;8;8;3;2;2;10];;
- : int list = [2; 3]
# intersect [] [1;2;3];;
- : int list = []
# intersect [1;2;3] [];;
- : int list = []
# intersect [] [];;
- : 'a list = []
# intersect [1;2;3;3;4;5;5;4] [1;5;6;4;6;10;1;4;4;1;3];;
- : int list = [1; 3; 4; 4; 5]

intersect [1;2;3] [4;3;8;8;3;2;2;10];;
intersect [] [1;2;3];;
intersect [1;2;3] [];;
```

```
intersect [] [];;
intersect [1;2;3;3;4;5;5;4] [1;5;6;4;6;10;1;4;4;1;3];;
```

(b) The type of the function is `'a list -> 'a list -> 'a list`, meaning that it takes two lists of the same type and returns a list of the same type.

(c) The time complexity of the first implementation is $\mathcal{O}(n_1 \cdot n_2)$, where n_1 is the length of the first list and n_2 is the length of the second list, since in the worst-case for each element of the first list we will examine all elements of the second list.

(d) The efficient algorithm takes $\mathcal{O}(n_1 + n_2)$ time, since in each call of the algorithm we are "removing" at least one element from the two lists. Since there are $n_1 + n_2$ elements, there can be at most $n_1 + n_2$ calls.

If the lists are not sorted, the implementation takes $\mathcal{O}(n_1 \log n_1 + n_2 \log n_2)$ time.

Exercise 9 [List union]

(a) Write an OCaml function that computes the union of two lists.

(b) If the given lists were sorted, would you be able to speed up the algorithm?

(a) In this case, we could take union as meaning, the elements that appear at least once in every list.

Implementation (slow):

```
let rec remove_all v = function
| [] -> []
| x::xs ->
    if x = v then remove_all v xs
    else x :: remove_all v xs;;

let rec union xx y = match xx with
  [] -> if y = [] then [] else union y []
| x::xs ->
   x :: union (remove_all x xs) (remove_all x y);;
```

Implementation (fast):

```
let rec remove_duplicates = function
| [] -> []
| [x] -> [x]
| x::y::xs ->
    if x <> y then x :: remove_duplicates (y::xs)
    else remove_duplicates (y::xs);;

let union x y = remove_duplicates (List.sort compare (x @ y) );;
```

Testcases:

```
# union [1;2;3] [4;3;8;8;3;2;2;10];;
- : int list = [1; 2; 3; 4; 8; 10]
# union [] [1;2;3];;
- : int list = [1; 2; 3]
# union [1;2;3] [];;
- : int list = [1; 2; 3]
# union [] [];;
- : 'a list = []
# union [1;2;3;3;4;5;5;4] [1;5;6;4;6;10;1;4;4;1;3];;
- : int list = [1; 2; 3; 4; 5; 6; 10]

union [1;2;3] [4;3;8;8;3;2;2;10];;
union [] [1;2;3];;
union [1;2;3] [];;
union [] [];;
union [1;2;3;3;4;5;5;4] [1;5;6;4;6;10;1;4;4;1;3];;
```

(b) The time complexity of the slow algorithm is $\mathcal{O}(n_1 + n_2)$ and of the efficient one it is $\mathcal{O}((n_1 + n_2)\log(n_1 + n_2))$. One can make it run in $\mathcal{O}(n_1 \log n_1 + n_2 \log n_2)$ time by sorting each one and then using two pointers as before.

Exercise 10 [zip/unzip] How does the following version of `zip` differ from the one defined in the course?

```
let rec zip = function
  (x::xs,y::ys) -> (x,y) :: zip(xs,ys)
| ([], []) -> [];;
```

The function given in the lectures was

```
let rec zip_lecture xs ys =
match xs, ys with
| (x::xs, y::ys) -> (x, y) :: zip_lecture xs ys
| _ -> []
```

The difference is that this implementation fails if the lengths of the two lists are different and also it has a different type. `'a list -> 'a list -> ('a * 'a) list` versus `('a list * 'a list) -> ('a * 'a) list`.

Exercise 11 [zip/unzip - inverses] *Attempt this exercise only if you are familiar with mathematical concepts referred to. Otherwise, come back to it after Discrete Maths* The functions `zip` and `unzip` seem like they are each other's opposites. Mathematically, two functions $f : A \to B$ and $g : B \to A$ are inverses of each other if for all arguments $a \in A$, $g(f(a)) = a$, and if for all arguments $b \in B$, $f(g(b)) = b$. If only the first condition holds, we call g the left inverse of f, and if only the second condition holds, we call g the right inverse of f. Is `unzip` the inverse, right inverse or left inverse of `zip`? Justify your answer.

[Exercise 4.1.1 in Lecturer's handout]

Exercise 12 You are given a list of names and a list of phones. `phones[i]` corresponds to the phone of the `names[i]` person. Given a phone number find the person who owns it.

Exercise 13 [Local declaration]

(a) Describe the syntax for declaring local values.

(b) What are the type constraints for when declaring local values?

(c) When are local values useful?

(d) How can you use local values to hide the internals of functions? Demonstrate this by providing a single function that implements the iterative factorial.

(a) `let <pattern_match> = e1 in e2`.

Note: In pattern matching, you can define more than one local values, e.g., `let (v1, (v2, v3)) = (1, (2, 3));;`.

Discussion: *How would you compare* `let a = 14 in let b = 10 in a + b;;` *and* `let a = 14, b = 10 in a + b;;`*?* The second one is easier to read, so it should be used when possible. However, note that we cannot use `let a = 14, b = a + 1 in a + b;;`.

(b) The type of `e2` is the type of the entire expression.

Note: There is also a more involved type constraint that the values defined by the pattern matching should type check `e2`.

(c) They are useful to avoid computing the same thing multiple times, to decompose a value into constituents and also hide internals of the function (e.g., helpers).

(d)
```
let fact n =
  let rec facti n a =
    match n with
```

```
        | 1 -> a
        | _ -> facti (n-1) (a*n)
    in facti n 1;;
```

Exercise 14 [Pattern expressions] Illustrate using examples how pattern expressions are used in OCaml.

```
let (v1, (v2, v3)) = (1, (2, 3));;

let to_bool = function
  0 -> false
| 1 -> true;;

let simple_and x y = match x, y with
  true, true -> true
| _, _ -> false;;
```

Discussion: *Can you have (v, v) = (1, 1)?* No, it outputs "Error: Variable v is bound several times in this matching".

Exercise 15 [All binary sequences (+)] Write an OCaml function that returns all the possible binary sequences of length k. For example, `all_bin_seqs 2` should return `[[0;0];[0;1];[1;0];[1;1]]` (or any permutation of this list).

Let's assume we know all binary sequences of length k, e.g., for $k = 2$ we know `[[0;0]; [0;1]; [1;0]; [1;1]]`. Then how can we obtain the sequences of length $k = 3$?
Observe that grouping the sequences of length 3 by their first digit, we get

```
[[0;0;0]; [0;0;1]; [0;1;0]; [0;1;1]; [1;0;0]; [1;0;1]; [1;1;0]; [1;1;1];]
```

Therefore, given `all_bin_seqs 2` we just have to append 0 to all of them and then append 1 to all of them and merge these two lists. We do this by defining an auxiliary function called `all_cons` which appends one item to all lists of a list.

```
(* Auxiliary function to append an element to all elements in the list. *)
let rec all_cons v = function
   [] -> []
 | x::xs -> (v::x) :: all_cons v xs;;

let rec all_bin_seqs k =
   if k = 0 then [[]]
   else all_cons 0 (all_bin_seqs (k-1)) @ all_cons 1 (all_bin_seqs (k-1));;

all_bin_seqs 2;;
all_bin_seqs 3;;
```

Testcases:

```
# all_bin_seqs 2;;
- : int list list = [[0; 0]; [0; 1]; [1; 0]; [1; 1]]
# all_bin_seqs 3;;
- : int list list =
[[0; 0; 0]; [0; 0; 1]; [0; 1; 0]; [0; 1; 1]; [1; 0; 0]; [1; 0; 1]; [1; 1; 0];
 [1; 1; 1]]
```

Discussion: *Why did we choose to return [[]] for $k = 0$?* We return this so that case $k = 1$ works and returns `[[0];[1]]`. If we had chosen to return `[]` it would return an empty list. Alternatively, we could handle $k = 1$ as the base case.

Exercise 16 [All partitions (++)] Write an OCaml function that returns all the possible ways to partition a given list into sublists of length 1 or 2. For example, `[1; 2; 3]` should return `[[[1]; [2]; [3]]; [[1; 2]; [3]]; [[1]; [2; 3]]`.

Let's consider an example: Assume that we know all partitions for [3; 4; 5] which are [[[3]; [4]; [5]]; [[3; 4]; [5]]; [[3]; [4; 5]]] and all partitions for [2; 3; 4; 5] which are [[[2]; [3]; [4]; [5]]; [[2; 3]; [4]; [5]]; [[2; 3]; [4; 5]]; [[2]; [3; 4]; [5]]; [[2]; [3]; [4; 5]]]. Then we can compute all partitions for [1; 2; 3; 4; 5] by: (1) pairing [1; 2] and then getting all partitions for [3; 4; 5] or (2) pairing [1] on its own and getting all partitions for [2; 3; 4; 5].
Hence, this gives

[[[1; 2];[3]; [4]; [5]]; [[1; 2];[3; 4]; [5]]; [[1; 2];[3]; [4; 5]]]@

[[[1];[2]; [3]; [4]; [5]]; [[1];[2; 3]; [4]; [5]]; [[1];[2; 3]; [4; 5]]; [[1];[2]; [3; 4]; [5]]; [[1];[2]; [3]; [4; 5]]]

Of course, this generalises for a list of n elements: First, compute the partitions for $[x_3; x_4; \ldots; x_n]$ and for $[x_3; x_4; \ldots; x_n]$ and then append $[x_1; x_2]$ to all partitions for the first list and append $[x_1]$ to all partitions returned for the second list. Since, there is no overlap for the partitions, this gives us each possible partition exactly once.

```
(* Auxiliary function to append an element to all elements in the list. *)
let rec all_cons v = function
   [] -> []
 | x::xs -> (v::x) :: all_cons v xs;;

let rec all_partitions = function
  [] -> [[]]
| (x::[]) -> [[[x]]]
| (x::y::xs) ->  all_cons [x] (all_partitions (y::xs)) @ all_cons [x; y]
    (all_partitions xs);;
```

Testcases:

```
# all_partitions [1];;
- : int list list list = [[[1]]]
# all_partitions [1;2];;
- : int list list list = [[[1]; [2]]; [[1; 2]]]
# all_partitions [1;2;3];;
- : int list list list = [[[1]; [2]; [3]]; [[1]; [2; 3]]; [[1; 2]; [3]]]
# all_partitions [1;2;3;4];;
- : int list list list =
[[[1]; [2]; [3]; [4]]; [[1]; [2]; [3; 4]]; [[1]; [2; 3]; [4]];
 [[1; 2]; [3]; [4]]; [[1; 2]; [3; 4]]]

all_partitions [1];;
all_partitions [1;2];;
all_partitions [1;2;3];;
all_partitions [1;2;3;4];;
```

Discussion:

- *What is the time complexity of your algorithm?* It is exponential because in each step we are examining two possible choices.
- *Can we do better?* No. Why? Because the time complexity is equal to the output size. With lazy lists we can produce exactly as many partitions as we need.
- *What is the size of the output list?* The size of the output list is $f(n) = f(n-1) + f(n-2)$. Hence, it is the n-th Fibonacci number.
- *What is the type of `all_cons`? What is the return value when you apply it with `all_cons [[[1]]] [1;2]`?* Its type is `'a list list * 'a -> 'a list list`. In the application `'a` is equal to `int list`, so the return type will be `int list list list`.

Exercise 17 We know nothing about the functions `f` and `g` other than their polymorphic types:

```
> val f = fn: 'a * 'b -> 'b * 'a
> val g = fn: 'a -> 'a list
```

Suppose that `f(1, true)` and `g 0` are evaluated and return their results. State, with reasons, what you think the resulting values will be.

The first function f has to be the swap function.
The second g can only be the function that creates a list with the same element (e.g., [], [0], [0; 0], [0; 0; 0], ...).
The main reasoning is that a function with one argument of type 'a, can only duplicate this value. It cannot create a new instance or modify it (because the transformation would need to hold for any possible type).
Discussion: *What other things could occur when calling these functions?*

- There could be an exception thrown.
- They can loop for ever.

2 Lecture 5

Exercise 18 [Selection sort] Another sorting algorithm (*selection sort*) consists of looking at the elements to be sorted, identifying and removing a minimal element, which is placed at the head of the result. The tail is obtained by recursively sorting the remaining elements.

(a) Write an OCaml function that implements selection sort.

(b) State, with justification, the time complexity of your function.

[Exercise 5.1 & 5.2 in Lecturer's handout]

```
let rec selection_sort = function
    | [] -> []
    | x::xs ->
        let rec select_r small output = function
        | []     -> small::(selection_sort output)
        | x::xs ->
            if x < small then
              select_r x (small::output) xs
            else
              select_r small (x::output) xs
        in
          select_r x [] xs;;
```

Testcases:

```
# selection_sort [1;5;3;2;1;10;23];;
- : int list = [1; 1; 2; 3; 5; 10; 23]
# selection_sort [];;
- : 'a list = []
# selection_sort [1];;
- : int list = [1]
# selection_sort [1;2];;
- : int list = [1; 2]
# selection_sort [2;1];;
- : int list = [1; 2]
# selection_sort [1;2;3];;
- : int list = [1; 2; 3]
# selection_sort [1;3;2];;
- : int list = [1; 2; 3]
# selection_sort [2;1;3];;
- : int list = [1; 2; 3]
# selection_sort [2;3;1];;
- : int list = [1; 2; 3]
# selection_sort [7;1;5;3;2;1;10;23];;
- : int list = [1; 1; 2; 3; 5; 7; 10; 23]

selection_sort [];;
selection_sort [1];;
selection_sort [1;2];;
selection_sort [2;1];;
selection_sort [1;2;3];;
```

```
selection_sort [1;3;2];;
selection_sort [2;1;3];;
selection_sort [2;3;1];;
selection_sort [7;1;5;3;2;1;10;23];;
```

Each step takes $\mathcal{O}(n)$ time, where n is the length of the list. This is repeated n times (in all cases), hence it requires $\mathcal{O}(n^2)$ times.

Discussion:

- How would you make your sorting method more generic? Use the type function that is used in the standard library which first accepts a comparison function and then the list.
- *What is the type of `List.sort`?* `('a -> 'a -> int) -> 'a list -> 'a list`

Exercise 19 [Bubble sort] Another sorting algorithm (*bubble sort*) consists of looking at adjacent pairs of elements, exchanging them if they are out of order and repeating this process until no more exchanges are possible. State, with justification, the time complexity of this approach.

(a) Write an OCaml function that implements bubble sort.

(b) State, with justification, the time complexity of your function.

[Exercise 5.3 & 5.4 in Lecturer's handout]

Implementation:

```
let rec bubble = function
    [] -> []
| [x] -> [x]
| (x::y::xs) -> if x < y then x::bubble(y::xs)
                else y::bubble(x::xs);;

let rec is_sorted = function
    (x::y::xs) -> if x > y then false
                  else is_sorted (y::xs)
|   _ -> true

let rec bubble_sort = function
    [] -> []
| xs -> if is_sorted xs then xs
        else bubble_sort (bubble xs);;
```

Testcases:

```
# bubble_sort [1];;
- : int list = [1]
# bubble_sort [1;2];;
- : int list = [1; 2]
# bubble_sort [2;1];;
- : int list = [1; 2]
# bubble_sort [1;2;3];;
- : int list = [1; 2; 3]
# bubble_sort [1;3;2];;
- : int list = [1; 2; 3]
# bubble_sort [2;1;3];;
- : int list = [1; 2; 3]
# bubble_sort [2;3;1];;
- : int list = [1; 2; 3]
# bubble_sort [7;1;5;3;2;1;10;23];;
- : int list = [1; 1; 2; 3; 5; 7; 10; 23]

bubble_sort [];;
bubble_sort [1];;
bubble_sort [1;2];;
bubble_sort [2;1];;
```

```
bubble_sort [1;2;3];;
bubble_sort [1;3;2];;
bubble_sort [2;1;3];;
bubble_sort [2;3;1];;
bubble_sort [7;1;5;3;2;1;10;23];;
```

Each step takes $\mathcal{O}(n)$ time. In the best case, this requires $\mathcal{O}(1)$ steps and in the worst case, it requires $\mathcal{O}(n)$ steps. So the worst-case time complexity is $\mathcal{O}(n^2)$.

Exercise 20 [Quicksort]

(a) Describe how *quicksort* works.

(b) What is the worst-case running time for quicksort?

(a) Quicksort chooses a pivot element x and then splits the list into two halves: the elements that are smaller than x and the elements that are larger than x. Then it recursively sorts these parts. The base case is a list of length 1.

(b) The worst-case complexity is when the input is sorted (or reversely sorted). Then we need $\mathcal{O}(n)$ choices of a pivot point and this gives an $\mathcal{O}(n^2)$ algorithm.

Discussion:

- *How are equal elements treated?* They should be placed in a separate sublist and then merged in the middle of the sorted lower and upper parts. If you place them either to the left or to the right of the pivot, then in the worst-case when there are many copies of an item, two parts would be very unbalanced with high probability.

  Another option is to make the elements unique by converting the list of values to a list of (index, value) pair. This way there are no equal elements. But this means that you need extra memory to do the sorting, which is not ideal.

- *How would you choose the pivot?* Choose a random index (or equivalently shuffle the array). Do not expect the input to be random.

Exercise 21 [Mergesort]

(a) Describe how *mergesort* works.

(b) By solving its recurrence relation, show that its running time is $\mathcal{O}(n \log n)$ for n being a power of 2.

(c) When would one prefer mergesort over quicksort?

(a) Read the lecture notes.

(b) Let $T(n)$ be the time required to sort a list of n elements. Sorting each of the halves takes $T(n/2)$ time each and merging takes roughly n operations. For a list with one element we can assume it takes $\mathcal{O}(1)$ time. Hence, the recurrence relation is given by

$$T(n) = 2 \cdot T(n/2) + n.$$

Again, we can start by trying out a few values.

$$\begin{aligned} T(2^1) &= 2 \cdot T(1) + 2 = 2^1 + 2 \\ T(2^2) &= 2 \cdot T(2^1) + 2^2 = 2 \cdot (2^1 + 2^1) + 2^2 = 2^2 + 2^2 + 2^2 \\ T(2^3) &= 2 \cdot T(2^2) + 1 = 2 \cdot (2^2 + 2^2 + 2^2) + 2^3 = 2^3 + 2^3 + 2^3 + 2^3 \end{aligned}$$

Hence, we formulate the conjecture that $T(2^k) = 2^k \cdot (k+1)$ and we will prove it using induction. The base case is true since $T(1) = T(2^0) = 2^0 \cdot (0+1) = 1$. Assume that it is true for $n = k$, i.e., $T(2^k) = 2^k \cdot (k+1)$ then

$$T(2^{k+1}) = 2 \cdot T(2^k) + 2^{k+1} = 2 \cdot (2^k \cdot (k+1)) + 2^{k+1} = 2^{k+1} \cdot (k+2).$$

Hence, by replacing $k = \log_2 n$, we get $T(n) = 2^k \cdot (k+1) = n \cdot (\log_2 n + 1)$ which is $\leq 2n \cdot (\log_2 n + 1)$ so $\mathcal{O}(n \log n)$.

(c) Mergesort should be preferred in applications where we want guaranteed worst-case performance (e.g., real-time medical applications), since it always has the $\mathcal{O}(n \log n)$ guarantee.

Exercise 22 [Hardness for sorting (+)] Explain the lower bound for the sorting time of a comparison based algorithm.

Consider a deterministic algorithm which makes $K(n)$ comparisons in the worst case. Consider the outcome of each comparison for each of the $n!$ possible permutations of the input list. Some of the permutations will yield a 1 and some others will yield a 0. We always follow the branch which is taken by most permutations. Hence, in this branch the number of permutations must be at least half of the starting ones. So, there have to be at least $\log_2(n!)$ comparisons until we remain with only one permutation.[1] Finally note that

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 \geq n \cdot \underbrace{(n-1) \cdot \ldots \cdot (n/2)}_{n/2 \text{ items}} = (n/2)^{n/2},$$

and so in the worst case the algorithm must make at least $n/2 \log(n/2)$ comparisons (i.e., $K(n) = \Omega(n \log n)$).
Discussion:

- *In an imperative language, how do you sort in faster than $\mathcal{O}(n \log n)$?* You can use counting sort if the values are small.
- *What about hashing?* Theoretically it takes some time to compute a hash value. It it just that on hardware (or on specific computation models) hashing is efficient. The lower bound only works for comparison-based sorting.
- *How many comparisons have to be made to find the maximum entry in a list?* If the algorithm asks fewer than $n-1$ questions, then we can pick a sequence for which the algorithm is wrong. (cf Part IA Algorithms)

Exercise 23 [Removing duplicates] Write an OCaml function that takes a list and returns a list containing all elements in the list with no duplicates (in any order). For example, given `[1; 2; 6; 2; 1; 3; 2]`, it returns `[1; 6; 3; 2]`. [*Hint:* Your algorithm should be running in $\mathcal{O}(n \log n)$, where n is the number of elements in the given list].

```
let rec remove_duplicates = function
| [] -> []
| [x] -> [x]
| x::y::xs ->
    if x <> y then x :: remove_duplicates (y::xs)
    else remove_duplicates (y::xs);;

let rem_duplicates x = remove_duplicates (List.sort compare x);;
```

The function works in $\mathcal{O}(n)$ time. Sorting takes $\mathcal{O}(n \log n)$ time.

3 Lecture 6

Exercise 24 [Enumeration types] Give the declaration of an OCaml type for the days of the week. Comment on the practicality of such a type in a calendar application. Why is it better than creating a custom encoding using integers?

```
type weekdays =
  Monday
| Tuesday
| Wednesday
| Thursday
| Friday
| Saturday
```

[1]There cannot be two permutations terminating with the same comparison outcomes, since this would mean that we made the same swaps to sort them (and so they would be the same permutation).

```
| Sunday;;

(* If you need to convert it to int, then
   just define a function and always use that. *)
let weekday_to_int = function
  Monday -> 0
| Tuesday -> 1
| Wednesday -> 2
| Thursday -> 3
| Friday -> 4
| Saturday -> 5
| Sunday -> 6;;
```

Exercise 25 [Datatypes] Describe the syntax and types for user-defined datatypes.

`type ('a) <name> = <constructor_name> of type | ...;;`

Discussion: What does it mean for a type abbreviation to be cyclic? It means that there is no finite instance for this type. Consider for example, `type example = Simple of example;;`.

Exercise 26 [Enumeration types]

(a) Describe how to define enumeration types using datatypes.

(b) What does the following code output?

```
type example = Aa | Bb | Cc;;
let process = function
  Aa -> "aa"
| bb -> "bb";;
process Cc;;
```

(a) This is just a generalisation of the weekdays example. Enumeration types can be defined as `type enum = Enum1 | Enum2 | ...;;`.

(b) The following code outputs “bb" since `bb` is not a constructor, so it matches everything (just like any variable name would do). Hence, it also matches `Cc` (since it does not match `Aa`). So, it outputs “bb".

More plainly, the code is equivalent to

```
type example = Aa | Bb | Cc;;
let process = function
  Aa -> "aa"
| _ -> "bb";;
process Cc;;
```

Exercise 27 [List datatype] Using the following datatype:

```
type 'a mylist = Nil | Cons of 'a * 'a mylist;;
```

(a) Write OCaml functions to perform the `head`, `tail`, `drop` and `take` operations.

(b) Write OCaml functions to convert your custom list from and to OCaml lists.

Implementation:

```
type 'a mylist = Nil | Cons of 'a * 'a mylist;;

let head (Cons(x, _)) = x;;
let tail (Cons(_, xs)) = xs;;

let rec take i = function
| Nil -> Nil
| Cons(x, xs) ->
    if i = 0 then Nil
    else Cons(x, take (i-1) xs);;
```

```
let rec drop i = function
| Nil -> Nil
| Cons(x, xs) ->
   if i = 0 then xs
   else drop (i-1) xs;;

let rec to_list = function
| Nil -> []
| Cons(x, xs) -> x :: to_list xs;;

let rec from_list = function
| [] -> Nil
| x::xs -> Cons(x, from_list xs);;
```

Testcases:

```
# head (Cons(3, Cons(2, Cons(1, Nil))));;
- : int = 3
# head Nil;;
Exception: Match_failure ("//toplevel//", 2, 9).
# tail (Cons(3, Cons(2, Cons(1, Nil))));;
- : int mylist = Cons (2, Cons (1, Nil))
# tail Nil;;
Exception: Match_failure ("//toplevel//", 1, 9).
# take 2 (Cons(3, Cons(2, Cons(1, Nil))));;
- : int mylist = Cons (3, Cons (2, Nil))
# take 4 (Cons(3, Cons(2, Cons(1, Nil))));;
- : int mylist = Cons (3, Cons (2, Cons (1, Nil)))
# drop 2 (Cons(3, Cons(2, Cons(1, Nil))));;
- : int mylist = Nil
# drop 4 (Cons(3, Cons(2, Cons(1, Nil))));;
- : int mylist = Nil
#
  from_list [];;
- : 'a mylist = Nil
# from_list [1;2;3];;
- : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
#
  to_list Nil;;
- : 'a list = []
# to_list (Cons(1, Cons(2, Nil)));;
- : int list = [1; 2]


head (Cons(3, Cons(2, Cons(1, Nil))));;
head Nil;;
tail (Cons(3, Cons(2, Cons(1, Nil))));;
tail Nil;;
take 2 (Cons(3, Cons(2, Cons(1, Nil))));;
take 4 (Cons(3, Cons(2, Cons(1, Nil))));;
drop 2 (Cons(3, Cons(2, Cons(1, Nil))));;
drop 4 (Cons(3, Cons(2, Cons(1, Nil))));;

from_list [];;
from_list [1;2;3];;

to_list Nil;;
to_list (Cons(1, Cons(2, Nil)));;
```

Exercise 28 [Integer expressions]

(a) Give an OCaml datatype that represents an integer expression. The expression could be: i) an

integer value, ii) the addition of two expressions, iii) the subtraction of two expressions, iv) the negation of a single expression and the multiplication of two expressions. For example, you should be able to write expressions of the form `Add( Mult(Int 2, Int 4), Int 1)`.

(b) Write an OCaml function `eval` that takes an expression and evaluates it. For example, `eval Add ( Int 2, Int 3)` should return 5.

(c) Now, we would like to add variables to the datatype. A variable will be indexed by a string, e.g., `Var("x")`. The `eval` function will take an expression and a context, i.e. a mapping from variables to values. For example, `eval Add(Int 2, Var "x") [("x", 4); ("y", 2)]` should return 6.

[Exercise 6.2.3, 6.2.4 in Lecturer's handout]

Implementation:

```
type expr = Int of int
| Neg of expr
| Add of expr * expr
| Sub of expr * expr
| Mul of expr * expr
| Div of expr * expr
| Var of string;;

let rec find_var ((x, vx)::xs) v =
   if x = v then vx
   else find_var xs v;;

let rec eval ls = function
| Var x -> find_var ls x
| Int i -> i
| Neg a -> - (eval ls a)
| Add(a, b) -> (eval ls a) + (eval ls b)
| Sub(a, b) -> (eval ls a) - (eval ls b)
| Mul(a, b) -> (eval ls a) * (eval ls b)
| Div(a, b) -> (eval ls a) / (eval ls b);;
```

Testcases:

```
# eval [] (Add(Add(Int 2, Int 3), Mul(Neg (Int 4), Int 5)));;
- : int = -15
# eval [("z", 5); ("y", 4); ("x", 3)](Add(Add(Int 2, Var "x"), Mul(Neg (Var "y"),
    Int 5)));;
- : int = -15
# eval [] (Div(Add(Int 5, Int 7), Int 6));;
- : int = 2

eval [] (Add(Add(Int 2, Int 3), Mul(Neg (Int 4), Int 5)));;
eval [("z", 5); ("y", 4); ("x", 3)](Add(Add(Int 2, Var "x"), Mul(Neg (Var "y"), Int
    5)));;
eval [] (Div(Add(Int 5, Int 7), Int 6));;
```

Discussion: *Can you extend this further to include if statements?* Yes, you can actually extend this to include a large subset of the OCaml language. But code will need to be provided as an abstract syntax tree. This is similar to interpreting a language.

Exercise 29 [Exceptions]

(a) What is the syntax associated with exceptions?

(b) What is the type of `let f x = raise Division_by_zero`? Why?

(c) What is the type of a function that throws an exception? Are there any benefits for this?

(d) How are exceptions used in the problem of making change?

(a) Defining an exception using `exception <name> of type`.

To raise an exception, write `raise e`.

To handle/catch an exception write `try e with |p1 -> e1 .. |pn -> en`.

(b) The type is `'a -> 'b` because there is nothing to restrict it. The type of `raise E` is `'a`.

Discussion:

- What is the type of the function `let f x = if x < 0 then raise Division_by_zero else x + 1;;`? The type of this function is `int -> int` because `raise E` unifies with anything.
- Which exception is raised in each of the following examples?
  - `let x = raise A in raise B` (A)
  - `(raise A) (raise B)` (B since the argument is evaluated first)
  - `(raise A, raise B)` (the execution order depends on the compiler, the language does not specify)

Source: here

(c) The type of a function that throws an exception is not altered. This means that the signature of the function does not record which exceptions it may throw. This means that the execution of a function has three possible outcomes: gives a value, throws an exception or loops forever. Other languages, such as Java, record the exception types in the signature and, for typed expressions, they require that these are handled in the parent method (or also included in the signature of the parent method).

(d) They are used for backtracking.

Exercise 30 [Options]

(a) Give the datatype for *options*.

(b) When would one use them?

(c) A friend of yours argues that returning integer error codes is much more effective than options. Do you agree?

(d) Rewrite `nth`, `take` and `drop`, so that they return options (with none for invalid arguments).

(a) `type 'a option = Some of 'a | None;;`

(b) One would use them in functions that request a value, which in some cases does not occur. The problem is that if there are more than one possible errors, it is not possible to distinguish between them.

Note: In Java, it is highly recommended by some style guides to choose Optionals instead of allowing a function to return null.

(c) It allows to distinguish between different types of errors, but documentation becomes harder. Also, what happens if a non-recognised integer is returned.

(d)
```
type 'a option = Some of 'a | None;;

let rec take i ls acc =
   if i < 0 then None
   else if i = 0 then Some (List.rev acc)
   else if ls = [] then None
   else take (i-1) (List.tl ls) ((List.hd ls)::acc)

let take_opt i ls = take i ls [];;

let rec drop_opt i ls =
   if i < 0 then None
   else if i = 0 then Some ls
   else if ls = [] then None
   else drop_opt (i-1) (List.tl ls);;

let rec nth_opt n ls =
   if n < 0 then None
   else if ls = [] then None
   else if n > 0 then nth_opt (n-1) (List.tl ls)
   else Some(List.hd ls);;
```

Testcases:

```
# take_opt 3 [1;2;3;4;5];;
- : int list option = Some [1; 2; 3]
# take_opt 10 [1;2;3];;
- : int list option = None
# drop_opt 3 [1;2;3;4;5];;
- : int list option = Some [4; 5]
# drop_opt 10 [1;2;3];;
- : int list option = None
# nth_opt 3 [1;2;3;4;5];;
- : int option = Some 4
# nth_opt 10 [1;2;3];;
- : int option = None

take_opt 3 [1;2;3;4;5];;
take_opt 10 [1;2;3];;
drop_opt 3 [1;2;3;4;5];;
drop_opt 10 [1;2;3];;
nth_opt 3 [1;2;3;4;5];;
nth_opt 10 [1;2;3];;
```

Exercise 31 [Binary trees]

(a) Give the datatype for *binary trees.*

(b) Write an OCaml function to find the longest path in a binary tree, starting from the root node. Modify it to find the shortest path. Reason by induction that your algorithm finds indeed the longest/shortest path.

(c) Write an OCaml function to count the number of elements in a tree.

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

(b) The base case is when the tree is just a leaf node. Then the longest and shortest path has length 1. Then assuming n_1 and n_2 are the longest (shortest) paths for the left and right node, then the maximum depth is `max(n_1, n_2)`.

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

let rec longest = function
| Lf -> 1
| Br(a, b) -> 1 + max (longest a) (longest b);;

let rec shortest = function
| Lf -> 1
| Br(a, b) -> 1 + min (shortest a) (shortest b);;
```

Discussion:

- What is the time complexity of the algorithm? Linear to the number of elements in the binary tree.
- What if the nodes are positively weighted? How do you find the longest weighted path from the root?
- (+) How would you find the longest distance between two nodes in a tree? For each node check the sum of the two longest paths. This is because we are checking all possible LCAs. (cf Part IA Algorithms and beyond)
- Can we make the longest function tail recursive?

```
let rec count = function
| Lf -> 1
| Br(a, b) -> 1 + (count a) + (count b);;
```

Testcases:

```
# shortest (Br(Br(Br(Lf, Lf), Lf), Br(Br(Lf, Lf), Br(Lf, Lf))));;
- : int = 3
# shortest (Br(Br(Lf, Lf), Lf));;
- : int = 2
# longest (Br(Br(Br(Lf, Lf), Lf), Br(Br(Lf, Lf), Br(Lf, Lf))));;
- : int = 4
# longest (Br(Br(Br(Lf, Br(Lf, Lf)), Lf), Br(Br(Lf, Lf), Br(Lf, Lf))));;
- : int = 5
# count (Br(Br(Br(Lf, Br(Lf, Lf)), Lf), Br(Br(Lf, Lf), Br(Lf, Lf))));;
- : int = 15

shortest (Br(Br(Br(Lf, Lf), Lf), Br(Br(Lf, Lf), Br(Lf, Lf))));;
shortest (Br(Br(Lf, Lf), Lf));;
longest (Br(Br(Br(Lf, Lf), Lf), Br(Br(Lf, Lf), Br(Lf, Lf))));;
longest (Br(Br(Br(Lf, Br(Lf, Lf)), Lf), Br(Br(Lf, Lf), Br(Lf, Lf))));;
count (Br(Br(Br(Lf, Br(Lf, Lf)), Lf), Br(Br(Lf, Lf), Br(Lf, Lf))));;
```

4 Lecture 7

Exercise 32 [Binary Search Trees]

(a) What is *binary search tree*? What is its main property?

(b) Which operations does a BST support? What is the worst-case time for each operation?

(a) A binary search tree is a binary tree with values stored in each node such that all nodes on the left subtree of A, have smaller values and all nodes on the right subtree of A have larger values.

(b) It supports the following operations:

- Insert : $\mathcal{O}(height)$
- Delete : $\mathcal{O}(height)$
- Update : $\mathcal{O}(height)$
- Find : $\mathcal{O}(height)$

For all these cases the worst-case complexity is triggered when the BST is just a chain and the requested element is at the bottom of the chain. As you will see in the Part IA Algorithms course, it is possible to balance the tree so that it has both an $\mathcal{O}(\log n)$ height and all operations running in $\mathcal{O}(\log n)$ time.

Exercise 33 Draw the binary search tree that arises from successively inserting the following pairs into the empty tree: `(Alice, 6)`, `(Tobias, 2)`, `(Gerald, 8)`, `(Lucy, 9)`. Then repeat this task using the order `(Gerald, 8)`, `(Alice, 6)`, `(Lucy, 9)`, `(Tobias, 2)`. Why are results different?

[Exercise 7.1.1 in Lecturer's handout]

```
let _ =
  Br(("Alice", 6),
     Lf,
     Br(("Tobias", 2),
        Br(("Gerald", 8),
           Lf,
           Br(("Lucy", 9), Lf, Lf)), Lf))

let _ =
  Br(("Gerald", 8),
     Br(("Alice", 6), Lf, Lf),
     Br(("Lucy", 9),
        Lf,
        Br(("Tobias", 2), Lf, Lf)))
```

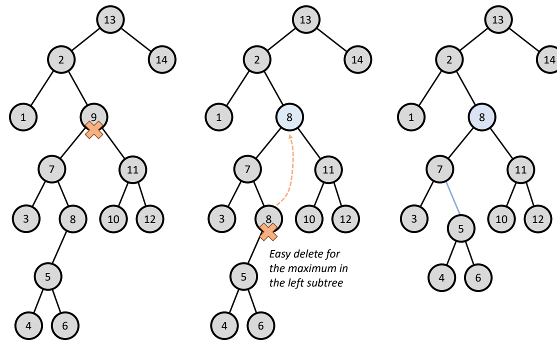
Exercise 34 [BST update] Explain using an example, how the BST update works.

Exercise 35 [BST delete (++)] Write an OCaml function to remove an entry from a BST. [*Hint:* if you remove a node, what can you use as its replacement.]

[Exercise 7.4, 7.5 in Lecturer's handout]

The main idea is to use a function similar to search to find the element we want to delete. Then:

- If this element has no children, then we can just remove it.
- If this element has one child, then we can replace it with that child.
- Otherwise, one possible replacement is the maximum node in its left subtree. This element will satisfy the properties of the BST and also will not have a right child (Why?).




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

(* Returns (max, new tree without max) *)
let rec deleteAndReturnMax = function
   Br(x, l, r) ->
      if r = Lf then (x, l)
      else let (mx, new_r) = deleteAndReturnMax r in
         (mx, Br(x, l, new_r));;

let rec delete x (Br(v, l, r)) =
   if x < v then Br(v, delete x l, r)
   else if x > v then Br(v, l, delete x r)
   else
      if r = Lf then l
      else if l = Lf then r
      else let (mx, new_l) = deleteAndReturnMax l in
         Br(mx, new_l, r);;

let tr = Br(
  13,
  Br(2,
     Br(1, Lf, Lf),
     Br(9,
        Br(7,
           Br(3, Lf, Lf),
           Br(8,
              Br(5,
                 Br(4, Lf, Lf),
                 Br(6, Lf, Lf)
              ),
              Lf)),
        Br(11,
           Br(10, Lf, Lf),
```

```
            Br(12, Lf, Lf)))),
      Br(14, Lf, Lf));;

delete 9 tr;;
delete 7 tr;;
delete 1 tr;;
delete 13 tr;;
delete 8 tr;;

# delete 9 tr;;
- : int btree =
Br (13,
 Br (2, Br (1, Lf, Lf),
  Br (8, Br (7, Br (3, Lf, Lf), Br (5, Br (4, Lf, Lf), Br (6, Lf, Lf))),
   Br (11, Br (10, Lf, Lf), Br (12, Lf, Lf)))),
 Br (14, Lf, Lf))
# delete 7 tr;;
- : int btree =
Br (13,
 Br (2, Br (1, Lf, Lf),
  Br (9, Br (3, Lf, Br (8, Br (5, Br (4, Lf, Lf), Br (6, Lf, Lf)), Lf)),
   Br (11, Br (10, Lf, Lf), Br (12, Lf, Lf)))),
 Br (14, Lf, Lf))
# delete 1 tr;;
- : int btree =
Br (13,
 Br (2, Lf,
  Br (9,
   Br (7, Br (3, Lf, Lf), Br (8, Br (5, Br (4, Lf, Lf), Br (6, Lf, Lf)), Lf)),
   Br (11, Br (10, Lf, Lf), Br (12, Lf, Lf)))),
 Br (14, Lf, Lf))
# delete 13 tr;;
- : int btree =
Br (12,
 Br (2, Br (1, Lf, Lf),
  Br (9,
   Br (7, Br (3, Lf, Lf), Br (8, Br (5, Br (4, Lf, Lf), Br (6, Lf, Lf)), Lf)),
   Br (11, Br (10, Lf, Lf), Lf))),
 Br (14, Lf, Lf))
# delete 8 tr;;
- : int btree =
Br (13,
 Br (2, Br (1, Lf, Lf),
  Br (9, Br (7, Br (3, Lf, Lf), Br (5, Br (4, Lf, Lf), Br (6, Lf, Lf))),
   Br (11, Br (10, Lf, Lf), Br (12, Lf, Lf)))),
 Br (14, Lf, Lf))
```

Discussion: *Which other element could we use instead of the maximum in the right subtree?* We could use the minimum in the left subtree.

Exercise 36 [Binary tree traversals]

(a) Describe the three tree traversal procedures.

(b) Write an OCaml function for each.

(c) What is the time and space complexity of each?

(a) There are the followig tree traversal procedures:

- Preorder: the node is printed, the function is called recursively on the left and then on the right subtree.
- Inorder : the function is called recursively on the left subtree, then the node is printed and then on the right.

- Postorder : the left, the right and then the current node.

Discussion: (++) How would you construct a binary tree given its preorder and inorder traversals? First find the node root node. This is just the first one in the preorder traversal. Then in the inorder traversal, split the nodes into left and right. Then also split in the preorder traversal and call the function recursively.

(b)
```
let rec preord = function
| Lf, vs -> vs
| Br (v, t1, t2), vs ->
    v :: preord (t1, preord (t2, vs))
let rec inord = function
| Lf, vs -> vs
| Br (v, t1, t2), vs ->
    inord (t1, v::inord (t2, vs))
let rec postord = function
| Lf, vs -> vs
| Br (v, t1, t2), vs ->
    postord (t1, postord (t2, v::vs))
```

Exercise 37 [Perfect binary tree] A *perfect* binary tree is one where all paths have the same length.

(a) Write an OCaml function to test whether a binary tree is perfect.

(b) Given that a perfect tree has height h, how many nodes does it contain?

```
let is_perfect tr = (longest tr) = (shortest tr);;
```

Alternatively, one can check that the number of nodes in the binary tree satisfy that number or nodes $= 2^h - 1$.
Testcases:

```
is_perfect (Br(Br(Lf, Lf), Br(Lf, Lf)));;
is_perfect Lf;;
is_perfect (Br(Lf, Br(Lf, Br(Lf, Lf))));;
```

Discussion: What is the time complexity for this? It is $\mathcal{O}(n)$, where n is the number of nodes in the binary tree.

Exercise 38 [Efficiently constructing a balanced BST (+)] A BST need not be balanced. Describe an efficient procedure that takes a sorted list of entries and creates a balanced BST.

```
(* Previous implementations. *)
let rec take i = function
| [] -> []
| x::xs ->
   if i > 0 then x :: take (i - 1) xs
   else []

let rec drop i ls =
   if i < 0 then failwith "Negative index"
   else if i = 0 then ls
   else if ls = [] then failwith "Requesting more items than there are"
   else drop (i-1) (List.tl ls);;

type 'a btree = Lf | Br of 'a * ('a btree) * ('a btree);;

(* New implementations. *)
let rec create_balanced_aux ls n = match ls, n  with
 | [] -> Lf
 | [x], _ -> Br(x, Lf, Lf)
 | (x::y::[]), _ -> Br(x, Br(y, Lf, Lf), Lf)
 | ls, n ->
      let left = take (n / 2) ls in
      let (r::right) = drop (n / 2) ls in
```

```
        Br(r, create_balanced_aux left (n / 2), create_balanced_aux right (n - (n
            / 2) - 1));;
let create_balanced xs = create_balanced_aux xs (List.length xs);;
```

Testcases:

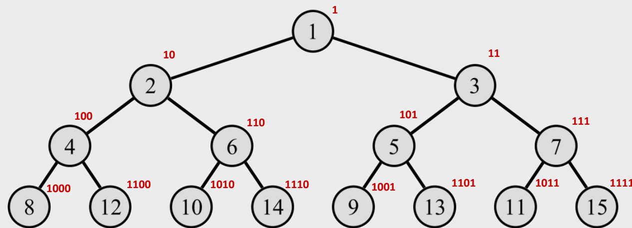
```
# create_balanced [1;2;3;4;5;6;7;8;9;10];;
- : int btree =
Br (6, Br (3, Br (1, Br (2, Lf, Lf), Lf), Br (4, Br (5, Lf, Lf), Lf)),
 Br (9, Br (7, Br (8, Lf, Lf), Lf), Br (10, Lf, Lf)))

create_balanced [1;2;3;4;5;6;7;8;9;10];;
```

The time complexity of this algorithm is linear, since $T(n) = 2 \cdot (n/2 + T(n/2)) = n + 2T(n/2)$, so in the same way that we did for mergesort we get $T(n) = \mathcal{O}(n \log n)$.

Discussion:

- *What would be the time complexity if we implemented this using arrays in a language that allows $\mathcal{O}(1)$ access time?* Take and drop would not be needed, so each step would need constant time. Hence, this would give the recurrence relation $T(n) = 2 \cdot T(n/2) + 1$ which gives $\mathcal{O}(n)$ time complexity.
- *What if we used functional arrays?* Then this would take $\mathcal{O}(\log n)$ time per access hence, $T(n) = 2 \cdot T(n/2) + \log n$ and hence it would give $T(n) = \mathcal{O}(n \log n)$ (not so easy to see).

Exercise 39 [Functional array (++)] With the aid of the diagram:



(a) Describe how *lookup* in a functional array works. What is its time complexity?

(b) Show that nodes at depth i have the bit set to 1 in the i-th position of their binary representation.

(c) Show that taking any subtree at depth i and removing the first i bits for all indices results in another functional array.

(a) The functional array is a binary tree representing an array, where the index of the element is determined by the path in the tree. More specifically, for every left turn append 0 and for every right turn append 1, then in the end append 1. For instance, element $11 = 1011_2$ is located at right (1), right (1), left (0). Note that the last (1) is ignored. The time complexity to access the i-th element is $\Theta(\log_2 i)$.

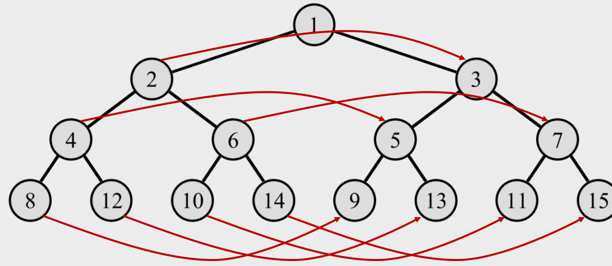
(b) This follows from the definition of the functional array, there are at most $\log_2 n$ bits in the binary representation of n.

(c) The i-th bit can only affect the path up to the i-th depth. After that it cannot.

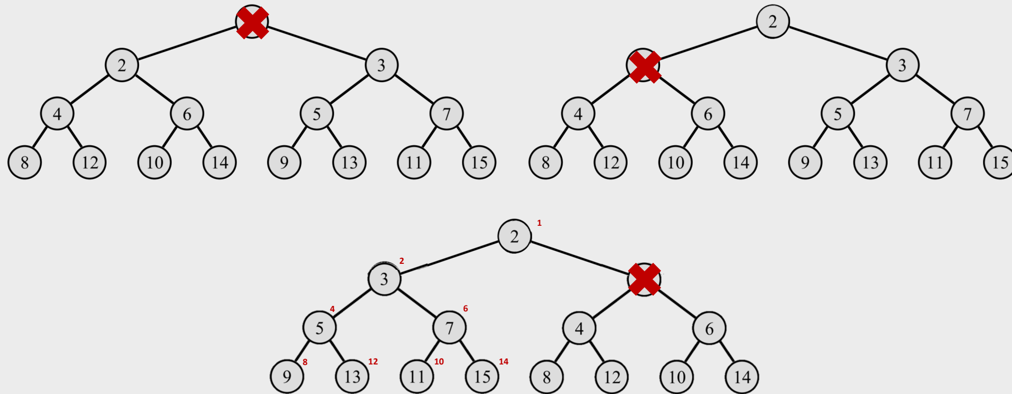
Discussion: *Why do we use functional arrays?* We use them because lists are inefficient to access at arbitrary points. OCaml also supports arrays, but they are not implemented in a functional way.

Exercise 40 [Functional array remove first element (+++)]

(a) Given a functional array, show that corresponding node indices in the left and the right subtree differ by one.



(b) Using the hint on the following figures, write an OCaml function that efficiently removes the first element of the functional array. All other elements in the array should be moved one position to the left.



(c) How efficient is your algorithm?

[Exercise 7.8 in Lecturer's handout]

(a) Only the first turn was different for corresponding elements, hence they must have the same binary representation (excluding the first bit).

(b)
```
type 'a btree = Lf | Br of 'a * ('a btree) * ('a btree);;

let rec remove_initial = function
|   (Br(v, Br(vl, ll, lr), r)) -> Br(vl, r, remove_initial (Br(vl, ll, lr)))
|   (Br(v, Lf, r)) -> Br(v, r, Lf)
|   (Lf) -> Lf;;
```

Testcases:

```
# remove_initial (Br(1, Br(2, Br(4, Lf, Lf), Br(6, Lf, Lf)), Br(3, Br(5, Lf,
    Lf), Br(7, Lf, Lf))));;
- : int btree =
Br (2, Br (3, Br (5, Lf, Lf), Br (7, Lf, Lf)),
 Br (4, Br (6, Lf, Lf), Br (4, Lf, Lf)))

remove_initial (Br(1, Br(2, Br(4, Lf, Lf), Br(6, Lf, Lf)), Br(3, Br(5, Lf, Lf),
    Br(7, Lf, Lf))))
```

(c) The algorithm takes $\mathcal{O}(\log n)$ time, since it only performs one operation per depth.

Discussion:

- In the functional array, one can append an element at the end of the array in $\mathcal{O}(\log n)$ time.
- The functional array is also a persistent data structure, meaning that it can keep track of previous versions of the data structure with just $\mathcal{O}(\log n)$ references. This is a benefit of functional programming languages that you can get these persistent implementations almost no effort. If you want to read more, look at the book "Persistent data structures" by Chris Okasaki.