Foundations of Computer Science

Solutions

2023

Contents

1. Introduction to programming

1.1. Conceptual questions

1. What is the main idea behind *abstraction barriers*? Why are they useful?

   > Writing complex programs would be humanly impossible if we had no way of structuring the code and using abstractions. They let us focus on one particular piece or aspect of the program, while ignoring the other components: for example, when writing a function, we only need to worry about the inputs and the return value, not the other things happening in the program. The abstraction barrier is then a "mental" separation between all the things that the function has to care about, and everything that it can abstract over. Sometimes abstraction barriers can be more concrete, such as the access modifiers in object-oriented languages or module systems like the one found in OCaml.

2. Why is it silly to write an expression of the form `if b then true else false`? What about expressions of the form `if b then false else true`? How about `if b then 5 else 5`? (You'd be surprised how many times I have to refer back to this exercise!)

   > Any `if`-expression returning a Boolean value can be simplified to a logical expression. The first case evaluates to `true` if `b` is true, and `false` if `b` is false – it behaves the same way as `b` itself. In the second case, we can get the same behaviour by negating `b`. In general, `if b1 then true else b2` is the same as `b1 || b2`, and `if b1 then b2 else false` is equivalent to `b1 && b2`.
   >
   > In the third case the type of the clauses is an integer, but both clauses have the same value, so conditional branching is unnecessary – the expression reduces to `5` in either case.

3. Briefly discuss the meaning of the the terms *expression*, *value*, *command* and *effect* using the following examples:

   - `true`
   - `57 + 9`
   - `print_string "Hello world!"`
   - `print_float (8.32 *. 3.3)`

   > Expressions and values are functional concepts; commands (statements) and effects are characteristic of imperative code. Values are irreducible expressions such as numbers and booleans: `5`, `true`. Expressions are compound structures, usually consisting of subexpressions combined with a top-level operator, like the `+` in `57 + 9`. Statements are *executable* instructions, which are usually used for their side effects that change the external world. For example, printing commands print a string on the screen without returning a useful value. The fourth example shows that statements can also contain expressions (but usually not vice-versa): the arithmetic expression is first evaluated to `27.456`, which is then printed as a string onto the screen by the print statement.

4. Which of these is a valid OCaml expression and why? Assume that you have a variable `x` declared, e.g. with `let x = 1`.

- `if x < 6 then x + 3 else x + 8`

  > Valid; the two branches have the same return type, so this is a well-formed expression.

- `if x < 6 then x + 3`

  > Invalid: we are only given the `then`-clause, so the expression cannot reduce if the Boolean condition is false. This would be allowed in an imperative language: if the condition doesn't hold, the statement in the `then` branch is not executed. Expression-based languages have no way of "skipping" the evaluation of subexpressions, so both cases must be handled. OCaml is special in that it *does* have imperative features and a one-branch if-statement, so the above would be syntactically valid; however, it would not type-check, since OCaml expects a statement in the `then` branch (which have type `unit`), rather than an integer expression.

- `if x < 6 then x + 3 else "A"`

  > Invalid: the two clauses have a different type. As every OCaml expression has to have a well-defined type, the type of an if-expression cannot depend on the value of the condition – whatever the condition is, we must be able to treat the expression in the same way. Hence the only way to ensure well-typedness is to enforce that both clauses must have the same type.

- `x + (if x < 6 then 3 else 8)`

  > Valid: as if-expressions are just expressions, they can appear anywhere we need an expression, as long as types match up. As the return type of the inner if-expression is an integer, the whole thing can be treated as an integer – for example, it can be added to another integer.

1.2. Exercises

5. One solution to the year 2000 bug mentioned in Lecture 1 involves storing years as two digits, but interpreting them such that 50 means 1950, 0 means 2000 and 49 means 2049.

   a) Comment on the merits and drawbacks of this approach.

   > **Merits**: We can retain the same crude, 2-digit representation, and the dates will take up the same space as before.
   >
   > **Drawbacks**: can be ambiguous, difficult to infer the format, and only defers the problem by 50 years.

   b) Using this date representation, code an OCaml function to compare two years (just like the `<=` operator compares integers).

   c) Using this date representation, code an OCaml function to add/subtract some given number of years from another year.

There are two options we can take: either convert to a four-digit year representation, compare/modify the dates, then convert back, or do the arithmetic on the two-digit representation directly.

The conversion to and from four-digit years can be done with the following functions:

```
let two_to_four d =
    if 0 <= d && d <= 49 then 2000 + d
   else if 50 <= d && d <= 99 then 1900 + d
   else -1

let four_to_two d =
    if 1950 <= d && d <= 2049 then d mod 100 else -1
```

Instead of returning `-1` in case of an error, it may be preferable to use exceptions or options. Comparing the two dates amounts to comparing their four-digit representations:

```
let leq_dates1 d1 d2 = two_to_four d1 <= two_to_four d2
```

Modifying dates "temporarily" converts to the four-digit representation.

We can define a general function that adds a positive or negative number to a date, the specialise it to an `add_to_date1` and `subtract_from_date1` function. With our current implementation, any erroneous input will produce the `-1` error code, but we need to check and test this in general.

```
let modify_date1 d n = four_to_two (two_to_four d + n)
let add_to_date1        d n = modify_date1 d n
let subtract_from_date1 d n = modify_date1 d (-n)
```

To directly compare and modify two-digit dates, we use the built-in `mod` function:

```
let leq_dates2 d1 d2 = (d1 + 50) mod 100 <= (d2 + 50) mod 100
let modify_date2 d n = let d2 = (d + 50) mod 100 + n
                       in if 0 <= d2 && d2 <= 99
                          then (d2 + 50) mod 100
                          else -1
let add_to_date2        d n = modify_date2 d n
let subtract_from_date2 d n = modify_date2 d (-n)
```

As both versions perform simple arithmetic operations, there is not a huge difference between their efficiency. The first approach might be more readable, but requires

> defining two auxiliary functions (which may nevertheless be useful in the rest of the program).

1.3. Optional questions

6. Because computer arithmetic is based on binary numbers, simple decimals such as 0.1 often cannot be represented exactly. Write a function that performs the computation

$$\underbrace{x + x + \cdots + x}_{n}$$

where x has type `float`. (It is essential to use repeated addition rather than multiplication!) See what happens when you call the function with $n = 1000000$ and $x = 0.1$.

> The function below performs the multiplication by repeated addition. Evaluating `mult_by_add (1000000, 0.1, 0.0)` gives `100000.000001332883` (on my system).

```
let rec mult_by_add n x acc = match n with
  | 0 -> acc
  | n -> mult_by_add (n - 1) x (acc +. x)
```

7. Another example of the inaccuracy of floating-point arithmetic takes the golden ratio $\varphi = 1.618\ldots$ as its starting point:

$$\gamma_0 = \frac{1+\sqrt{5}}{2} \qquad \text{and} \qquad \gamma_{n+1} = \frac{1}{\gamma_n - 1}$$

In theory, it is easy to prove that $\gamma_n = \cdots = \gamma_1 = \gamma_0$ for all $n > 0$. Code this computation in OCaml and report the value of γ_{50}. *Hint*: in OCaml, $\sqrt{5}$ is expressed as `sqrt 5.0`.

> The code below defines `phi` and the iterated function. The computed value starts diverging significantly around γ_{35}, but eventually converges to about `-0.6181` around γ_{48}.

```
let phi = (1.0 +. sqrt 5.0) /. 2.0

let rec iterg n x = match n with
  | 0 -> x
  | n -> iterg (n-1) (1.0 /. (x -. 1.0))

let gamma n = iterg n phi
```

2. Recursion and efficiency

2.1. Conceptual questions

1. Use a *recurrence relation* to find an upper bound for the recurrence given by $T(1) = 1$ and $T(n) = 2T(n/2) + 1$. Prove that your solution is an upper bound for all n using mathematical

induction.

Given a recursive algorithm, we estimate its runtime (or space complexity) with *recurrence relations*: an abstract function $T(n)$ for the amount of "time" needed to process an input of size n. As the algorithm is recursive, the time it takes to process an input is usually dependent on the time it takes to process a smaller input – therefore the function T will be recursive itself.

In this case, we are given the recurrences

$$T(1) = 1 \quad \text{and} \quad T(n) = 2T(n/2) + 1.$$

To determine the asymptotic time complexity of the algorithm, we need a closed-form, non-recursive expression for its runtime – this is what solving the recurrence relation will give us. One method for doing this is via *substitution*: looking at the recursive case for the definition of T, and repeatedly substituting the definition itself for the recursive subexpression until we notice some pattern.

$$\begin{aligned} T(n) &= 2T(n/2) + 1 \\ &= 2(2T(n/4) + 1) + 1 && = 2^2 T(n/4) + 2^1 + 1 \\ &= 2^2(2T(n/2^3) + 1) + 2^1 + 2^0 && = 2^3 T(n/2^3) + 2^2 + 2^1 + 2^0 \end{aligned}$$

Now we can conjecture a general formula after k expansions:

$$T(n) = 2^k T(n/2^k) + \sum_{i=0}^{k-1} 2^i = 2^k T(n/2^k) + (2^k - 1)$$

where we used the standard identity for the sum of the geometric series. Now, to achieve a closed-form expression, we need to find a k which makes $T(n/2^k)$ a constant value: looking back to our definition of T, we can get a constant value $T(n/2^k) = 1$ if $n/2^k = 1$. This holds for $k = \log_2 n$, which we can substitute into our formula above to get

$$T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + (2^{\log_2 n} - 1) = nT(1) + (n-1) = 2n - 1$$

That is, the closed-form solution to our recurrence relation is $T(n) = 2n - 1$, i.e. the algorithm has linear time complexity $O(n)$.

To prove that our conjecture above was correct, we use proof by mathematical induction. The base case is easy: $T(1) = 2 \times 1 - 1 = 1$. The inductive case assumes the induction hypothesis $T(n) = 2n - 1$ and requires us to prove the case for $T(2n) = 2T(n) + 1$ (we can assume that the argument is a multiple of two without a loss of generality):

$$T(2n) = 2 \times 2n - 1 = 2 \times (2n - 1) + 1 = 2T(n) + 1$$

That is, our closed-form expression satisfies the recurrence relation. □

2.2. Exercises

2. Code an iterative version of the efficient `power` function from Section 1.6.

> We initialise an accumulator argument to `1.0`, and return it when the exponent is 0 (which we determine by pattern-matching). Otherwise we check if the exponent is even: if it is, we halve the exponent and square the base; if it is odd, we do the same thing, but also multiply the accumulator by the base. As usual, the tail-recursive version of a function is very similar in structure to the recursive one, except we manipulate the accumulator instead of the result value of the recursive call.

```
let rec power_aux x n acc = match n with
  | 0 -> acc
  | n -> if n mod 2 = 0
         then power_aux (x *. x) (n / 2) acc
         else power_aux (x *. x) (n / 2) (acc *. x)
let power x n = power_aux x n 1.0
```

2.3. Optional questions

3. Let $g_1, \ldots, g_k$ be functions such that $g_i(n) \geq 0$ for $i = 1, \ldots, k$ and all sufficiently large n. Show that if $f(n) = O(a_1 g_1(n) + \cdots + a_k g_k(n))$ then $f(n) = O(g_1(n) + \cdots + g_k(n))$.

> One way to show this is to notice that $a_1 g_1(n) + \cdots + a_k g_k(n)$ is bounded by $a g_1(n) + \cdots + a g_k(n)$ where $a = \max\{a_1, \ldots, a_k\}$. As O-notation gives upper bounds, we have that $f(n) = O(a g_1(n) + \cdots + a g_k(n)) = O(a(g_1(n) + \cdots + g_k(n)))$, but as a is just a constant factor, it can be ignored, giving us $f(n) = O(g_1(n) + \cdots + g_k(n))$.

3. Lists

3.1. Conceptual questions

1. I often see some recurring stylistic/syntactic mistakes and "code smells" in students' work. Learn to avoid them by pointing out what is incorrect, redundant, or suspicious in the following fragments. (*Note*: you can assume that `x : int` and `xs : int list` and the relevant `List` functions are in scope.)

   - `1 :: [2, 3, 4]`

   > The list element separator in OCaml is the semicolon `;` rather than the comma. Somewhat annoyingly, the above expression would be a type error, rather than a syntax error (i.e. it's valid OCaml code, but the types don't match up): OCaml interprets `[2, 3, 4]` as a singleton list containing the tuple `2, 3, 4` (omitting the parentheses is sometimes allowed), and fails because we are trying to cons an `int` to a `(int * int * int) list`. Consequently, doing something like `[1, 2] @ [3, 4]` *does*

> typecheck, but gives a list `[(1,2), (3,4)] : int * int list` – probably not what was intended.

- `"hello " @ "world"`

  > While strings can be *thought of* as lists of characters, they are not implemented as such in OCaml, so standard list operations cannot be used to manipulate strings. Concatenation of strings is done via the `s1 ^ s2` operator, for example. Other string operations are found in the String module of the OCaml standard library.

- `xs @ [x]`

  > This expression typechecks and does what we intended: add an element to the end of a list. However, the append operator `@` is linear in its *first* argument, so `xs @ [x]` has to process most of the list only to add a single element. As a one-off occurrence this might be okay (indeed, it's impossible to add an element to the end of a list without traversing the whole list) – the problem occurs when this is repeated multiple times, such as in a recursive function `nrev` in Section 3.6. If this linear operation is repeated a number of times proportional to the length of the list, we end up with *quadratic* cost list operation. Instead, we can often refactor the function in a way that avoids accessing the end of the list, and keep the costs linear (for example, with tail recursion optimisation).

- `[x] @ xs`

  > Appending a singleton list to a list is the same as consing: `[x] @ xs = x :: xs`. This is more of a stylistic issue, though there is a tiny amount of overhead in calling the append function (which completes after two iterations) rather than directly using the data constructor.

- `let rec f xs = if len xs = 0 then 0 else hd xs + f (tl xs)`

  > The operations `hd : 'a list -> 'a` and `tl : 'a list -> 'a list` are so-called *partial functions*, because they don't handle all possible values of their input type `'a list`. OCaml gives a warning about non-exhaustive pattern-matching when these functions are defined, since neither of them has a pattern handling the `[]` case. It's impossible to write a non-partial (total) function of the type `'a list -> 'a` – if `hd` is given the empty list, it has no way of producing a polymorphic value of type `'a`. In contrast, we *could* define `tl []` to be `[]`, since this typechecks and makes `tl` total; however, it's not obvious if that definition makes logical sense, since it doesn't match the "list without its first element" intuition of tail. Therefore `tl []` is usually also left undefined, leading to the same issues as `hd`.
  >
  > The recommended way of accessing the head and tail of a list in a function is by pattern-matching, making sure to handle both the empty and the cons cases. This way our function becomes "obviously" total, and this can be confirmed by the compiler (as opposed to, for example, checking if a list is empty with an if-expression and using

> `hd` and `tl` in the else-branch – we will never encounter an error at runtime, but this may not be obvious to the compiler and it may complain).

2. We've seen how tail-recursion can make some list-processing operations more efficient. Does that mean that we should write all functions on lists in tail-recursive style?

> As the notes put it, "Never add an accumulator merely out of habit". It would be an instance of *premature optimisation* (the "root of all evil" according to Donald Knuth), where we are trying to make code more efficient at the expense of simplicity, maintainability and sometimes even correctness, without even considering if the optimisation is necessary or the code is "good enough" as it is. Tail-recursion as an optimisation procedure is fairly simple, but it already has a noticeable overhead in terms of code simplicity: we need to define an auxiliary function with an extra accumulator argument, turn the intuitive recursive implementation "inside out", and make a wrapper function to initialise the accumulator. In some cases this indeed leads to constant space complexity by reusing the call stack (at least in languages where this optimisation is available and enabled), but even that is not guaranteed: `append` would benefit nothing from a tail-recursive implementation. You may also have noticed the awkward tendency of list accumulators to end up reversed (unless the accumulator is extended with `acc @ [x]`, which is of course a big no-no), requiring an extra linear processing step at the end – or not, if we happen to be defining list reverse!
>
> It's also worth considering if an optimisation will lead to a noticeable improvement in practical use – and the best way to gauge this is to write the most straightforward implementation first, and see if that becomes the bottleneck in standard use. Computers tend to have quite a lot of memory these days (certainly many orders of magnitude more than at the time when this course was written), so we can often get away with a space-inefficient algorithm if it will not run for problematically large inputs. Consider, for example, the factorial function (Tick 1): it can certainly be made more "space-efficient" with tail-recursion, but we won't really notice problems due to call stack size unless the number of recursive calls is in the thousands – and when was the last time you wanted to calculate the factorial of 1000?
>
> A third point to mention is that a tail-recursive function is not always the result of a dedicated optimisation step – sometimes the natural implementation of an operation is already tail-recursive. An example is the `last` function below (and in Tick 2), which drops the head of the list until the last element is reached. Since we don't do anything to the head of `x::xs`, the recursive call is simply `last xs`, which is automatically a tail call (top-level recursive call).

3.2. Exercises

3. Code a recursive and an iterative function to compute the sum of a list's elements. Compare their relative efficiency.

> ```
> let rec rsum = function
>   | []    -> 0
> ```

```
  | x::xs -> x + rsum xs

let rec isum_aux xs acc = match xs with
  | []    -> acc
  | x::xs -> isum_aux xs (acc + x)
let isum xs = isum_aux xs 0
```

Both functions have linear time complexity (in the length of the list), as we need to traverse the whole list. However, the recursive version also uses linear space, while the tail-recursive one keeps the space usage constant.

4. Code a function to return the last element of a non-empty list. How efficiently can this be done? See if you can come up with two different solutions.

   There are two sensible approaches: either by a recursive function (dropping the head element of a list until only one element is left), or by taking the head of the reversed list. Both are linear in the length of the list (we cannot do better), but the first version is lighter on memory use, as the second one reverses the whole list, even though we only care about a single element. In both cases the function is undefined if the list is empty – we can make this an obvious error by defining a custom exception.

   ```
   let rec last1 = function
     | [x]     -> x
     | (_::xs) -> last1 xs

   let last2 xs = List.hd (List.rev xs)
   ```

5. Code a function to return the list consisting of the even-numbered elements of the list given as its argument. For example, given `[a,b,c,d]` it should return `[b,d]`. *Hint*: pattern-matching is a very flexible concept.

   While there are multiple ways of accomplishing this, the most concise and elegant way is to use nested pattern-matching: in the pattern `x::xs`, we can further pattern-match on the list `xs` to access the head element of the tail.

   ```
   let rec evens = function
     | []       -> []
     | [x]      -> []
     | x::y::ys -> y :: evens ys
   ```

   While it is conventional to have patterns of increasing generality, in this case we can make the function even more concise by treating the `[]` and `[x]` cases in one pattern placed *after* the `x::y::ys` one. Since the patterns are checked one-by-one from top to bottom,

this wildcard will only be reached when the list argument has fewer than two arguments – the result is `[]` in both the `[]` and the `[x]` cases.

```
let rec evens = function
  | x::y::ys -> y :: evens ys
  | _        -> []
```

6. Code a function `tails` to return the list of the tails of its argument. For example, given the input list `[1, 2, 3]` it should return `[[1, 2, 3], [2, 3], [3], []]`.

   A simple recursive solution works well here. Tail-recursion would require us to reverse the resulting list (which is often a drawback of writing tail-recursive functions on lists).

   ```
   let rec tails = function
     | []      -> [[]]
     | (x::xs) -> (x::xs) :: tails xs
   ```

3.3. Optional questions

7. Consider the polymorphic types in these two function declarations:

   ```
   let id x = x
   val id : 'a -> 'a = <fun>
   let rec loop x = loop x
   val loop : 'a -> 'b = <fun>
   ```

   Explain why these types make logical sense, preventing runtime type errors, even for expressions like `id [id [id 0]]` or `loop true / loop 3`.

   The type of the `id` function is the polymorphic function `'a -> 'a`: it can be instantiated with any type. In `id [id [id 0]]`, all occurrences of `id` have different types: the innermost one is `int -> int`, the one outside it is `int list -> int list`, the outermost one is `int list list -> int list list`. allows the same polymorphic function to be used at different types in the same expression, as the constraint condition of polymorphism significantly restricts the ways a function can be implemented. In fact, the only function that can have the type `'a -> 'a` is `id`, as any manipulation of the input would require knowing something about its type.

   The `loop` function is a bit stranger: its type says that it can take any input and its return value can have any type. It seems to act like a very general conversion function, but looking at the implementation we can see that this is not the case: the reason why the return value can have any type is that `loop` never returns. The function continuously loops and therefore will never return anything – so its return type “might as well” be anything we want. This means that we can arbitrarily produce an expression of a type we require, but it

> will actually be unusable: any program containing a call to `loop` will loop forever.

8. Looking at the tail-recursive functions you've seen or written so far, think about why they are called *tail*-recursive: what is the common feature of their evaluation that would explain this terminology? If you have previous understanding of how functions are evaluated in a computer (stack frames), can you explain why tail-recursive functions are often more space-efficient than recursive ones?

   > The "tail" part of tail-recursion refers to *tail calls*: the last function called before a function returns. In imperative languages this would usually be the last statement in the definition; in functional languages it is the outermost function call. The reason it is important is that the return value of the tail call will be the return value of the whole function, so instead of using a new stack frame (the local region of memory which is used by a function and then freed up (popped) once the function returns), we can just reuse the current one.
   >
   > This is particularly useful in recursion, as deeply recursive functions may build up a lot of stack frames and potentially cause overflow. This is because the result of the recursive call is often modified (e.g. incremented), but that computation has to be *suspended* until the recursive call returns. If we use tail-recursion, the recursive call will also be the tail call, so we can reuse the existing stack frame without any suspended computation. This results in constant memory usage, which is sometimes (but not always!) desirable.

4. More on lists

4.1. Conceptual questions

1. Suppose we use a variant of the list zipping function which takes the two list arguments as a pair and has the type `zip : 'a list * 'b list -> ('a * 'b) list`. This way, 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.

   > The two functions would be inverses of each other if their composition was the identity in both directions, for any input. However, we can exhibit counterexamples for this: if two lists being zipped have different lengths, the "surplus" elements are dropped, so we would not get back the two full lists after unzipping:
   >
   > ```
   > unzip (zip ([1;2;3;4], [5;6])) = unzip [(1,5); (2,6)]
   >         = ([1;2],     [5;6])
   > ```
   >
   > Hence we know that `unzip` cannot be the left inverse of `zip`, and hence it cannot be its full inverse. However, as unzipping cannot drop any elements (the resulting two lists must have the same length), we do have `zip (unzip l) = l` for all inputs, e.g.

```
zip (unzip [(1,5); (2,6); (3,7)]) = zip ([1;2;3], [5;6;7])
         = [(1,5); (2,6); (3,7)]
```

This can be formally proved using *structural induction* on lists (see later), but should be clear by looking at the definition of the two functions. Therefore we can conclude that `unzip` is the right inverse of `zip`.

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

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

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, and how the functions can be defined. Can any of the definitions be changed if termination wasn't a requirements?

The crux of this exercise is that polymorphism provides flexibility for the user but serious constraints for the implementer. If the user should be able to call a function on an input of any type (e.g. the `reverse` function on a list of any type), the writer of the function cannot assume anything about the input, i.e. they cannot call any non-polymorphic function on those arguments. This severly limits the ways in which a polymorphic function can be implemented, but this is a good and desired thing – the stricter our type system, the more bugs it can catch and avoid.

In this example, the function `f` has a tuple argument and a tuple result, with the polymorphic types flipped around. As we cannot assume anything about the type of the arguments, we cannot modify them in any way – the only thing we can do is swap them around in a pair.

```
let f (a,b) = (b,a)
```

This is similar to how the only possible function of the type `'a -> 'a` is the identity function: it cannot examine or modify an argument of polymorphic type, so the only thing it can do is return it unchanged.

Polymorphism is not just (or necessarily) about constraining the *number* of ways a function can be implemented, but the *way* in which an implementation can be given. For example, the second function `g` can have an infinite number of implementations, but they are all quite boring: we can either just return the empty list (which would actually have the more general type `'a -> 'b list`), or the singleton list containing the input once, or the two-element containing two copies of the input, and so on. Any return value will simply be a list of repetitions of the input – again, because we don't know anything about the type of the argument, so we cannot modify it in any way.

There is actually another way to implement `g`, but even though the definition typechecks, it

would not return anything: an infinite list of repetitions of the input, defined as a recursive function

```
let rec g x = x :: g x
```

Just as `loop` in the previous exercise sheet, the typechecker is more "lenient" if a function doesn't terminate, as there is no harm that can be done even if the return value of the function is absurd – it never returns anyway.

4.2. Exercises

3. a) Use the `member` function on Page 31 to implement the function `inter` that calculates the *intersection* of two OCaml lists – that is, `inter xs ys` returns the list of elements that occur both in `xs` and `ys`. You may assume `xs` and `ys` have no repeated elements.

   We recurse through one list and use the membership test to check whether the element is in the other list as well.

   ```
   let rec inter xs ys = match xs with
     | []    -> []
     | x::xs -> if member x ys then x::inter xs ys
                               else    inter xs ys
   ```

   b) Similarly, code a function to implement set union. It should avoid introducing repetitions, for example the union of the lists `[4;7;1]` and `[6;4;7]` should be `[1;6;4;7]` (though the order should not matter).

   We can write a function that resembles `inter` from Slide 404, except the base case returns the nonempty list, and if the head added to the union if it does not appear in the second list:

   ```
   let rec union xs ys = match xs with
     | []    -> ys
     | x::xs -> if member x ys then    union xs ys
                               else x::union xs ys
   ```

   Returning the second argument if the first list is empty somewhat resembles how we return the accumulator in the base case of tail-recursive functions. In fact, we can write `union` tail-recursively, but treating the second list as the accumulator – we simply move the new elements from the first list to the second:

   ```
   let rec union xs ys = match xs with
     | []    -> ys
     | x::xs -> union xs (if member x ys then ys else x::ys)
   ```

4. Code a function that takes a list of integers and returns two lists, the first consisting of all nonnegative numbers found in the input and the second consisting of all the negative numbers. How would you adapt this function so it can be used to implement a sorting algorithm?

> The simplest solution is similar to `unzip`, where we want to pattern-match on the result of the recursive call. This can be done either with `case`-expressions (upcoming), or local bindings. We can avoid the binding by using tail-recursion with two accumulators, but then we would have the resulting lists in reverse order.

```
let rec split = function
  | []    -> ([], [])
  | x::xs -> let (nonneg, neg) = split xs
             in if x < 0 then (   nonneg, x::neg)
                         else (x::nonneg,    neg)
```

> The implementation is very similar to the partition step of quicksort, except it "hardcodes" the pivot to 0. To make it more general, we should take the pivot as an argument and perform the comparison on it.

4.3. Optional questions

5. How does this version of `zip` differ from the one in the course?

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

> Compiling this function gives a "Matches not exhaustive" warning, which explains the problem with this definition. The two patterns we declare are either for two nonempty lists, or two empty lists. The function would work fine if the two lists were of the same length – every recursive call would happen on the two tails of the lists (which also have equal length) so eventually the lists will become empty simultaneously. However, if one of the lists is shorter, the recursive call will eventually happen on an empty and a nonempty list – but we have no patterns which cover this possibility. Instead of dropping the elements of the nonempty lists (like the normal `zip` function), we encounter a pattern match exception.

6. What assumptions do the "making change" functions make about the variables `till` and `amt`? What could happen if these assumptions were violated?

> Most importantly, the `till` contains positive coin values in nonincreasing order. Having duplicate coin values wouldn't cause issues, but a larger coin value will never be used if it appears later in the list than a lower coin value. If any of the values is negative or zero, the function will not terminate – the amount will either increase or not change, but will never

> reach zero. If the `amt` is initially negative, the code terminates with the correct answer that no solution is possible, but it has to perform the whole algorithm before concluding that.

5. Sorting

5.1. Conceptual questions

1. You are given a long list of integers. Would you rather:

   a) Find a single element with linear search or sort the list and use binary search?

   > Finding the last element is linear in the worst case (the scenario being that it is the last element), sorting can generally be considered to be $O(n \log n)$ (e.g. using mergesort or quicksort), so for a single element sorting is wasteful.

   b) Find a lot of elements with linear search or sort the list and use binary search?

   > Finding k elements is $k \times O(n) = O(kn) \approx O(n^2)$ with linear search, but $k \times O(\log n) = O(k \log n) \approx O(n \log n)$ with binary search. The latter requires a preliminary step of sorting the list, which can be done in $O(n \log n)$ time, but this does not increase the worst-case complexity of the second method. Hence, if we are planning to look up a lot of elements in a list (e.g. if it models a database), it is heavily recommended to sort the list first and then use the much more efficient binary search.

   c) Find all duplicates by pairwise comparison or sort the list and check for adjacent values?

   > Pairwise comparison performs $O(n)$ comparisons for each element, giving quadratic complexity. Sorting is $O(n \log n)$, and checking for adjacent elements is linear, so sorting the list first is recommended.

2. 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. Analyse the time complexity of this approach.

   > The best-case scenario is when the list is already sorted – the algorithm will not do anything to the list and terminate in linear time (contrast this with more efficient algorithms, which may nevertheless try sorting a sorted list – such as quicksort with a naive choice of the pivot). The worst case is a reverse-sorted list: every pass of the algorithm will bubble the head of the list up to the very end of the unsorted portion of the list. This takes $n + (n-1) + (n-2) + \cdots + 1 = \sum_{k=1}^{n} k$ steps, which equals $\frac{n(n-1)}{2}$ – multiplying this gives us a quadratic runtime in the worst case.

5.2. Exercises

3. Implement bubble sort in OCaml.

   > The easiest approach is to split the algorithm into two functions: a single sweep across the list which performs all possible exchanges, and a recursive sorting function that performs

the sweep until the list is sorted. The former uses nested pattern matching to compare the first two elements, and the latter checks if the sweeping changed anything – if not, the sort is complete.

```
let rec sweep = function
  | []           -> []
  | [x]          -> [x]
  | (x1::x2::xs) -> if x1 > x2 then x2::sweep (x1::xs)
                               else x1::sweep (x2::xs)

let rec bubblesort l = let l' = sweep l
                       in if l <> l' then bubblesort l' else l
```

5.3. Optional questions

4. 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) State, with justification, the time complexity of this approach.

   To find the minimum element in an unsorted list, we cannot do better than checking the entire list. We need to do this for every element in the unsorted part of the list, so there will be $n + (n-1) + (n-2) + \cdots + 1$ comparisons, which is characteristic of an $O(n^2)$ algorithm.

   b) Implement selection sort using OCaml.

   This solution uses two *mutually recursive* functions which call each other (and therefore indirectly call themselves recursively). As before, we separate the algorithm into a selection step and a sorting step that repeatedly performs the selection of the minimum element. The `select` function traverses the list argument, moving all the elements to the third argument (which acts as an accumulator) except the minimum one – this is determined by keeping track of the "running minimum" in the `min` argument and comparing it to the head of the list. When the list is depleted and the minimum element is found, we mutually recursively sort the rest of the list while adding the minimum to the front. The outer `selection_sort` function kicks off the process by calling `select` with the tail and the head as the running minimum.

```
let rec selection_sort = function
  | [] -> []
  | x::xs ->
   let rec select min ys rest = match ys with
     | []    ->  min::selection_sort rest
```

```
      | y::ys -> if y < min then select y ys (min::rest)
                              else select min ys (y::rest)
    in select x xs []
```

6. Datatypes and trees

6.1. Conceptual questions

1. Examine the following function declaration. What does `ftree (1,n)` accomplish?

```
let rec ftree k = function
  | 0 -> Lf
  | n -> Br(k, ftree (2*k) (n-1), ftree (2*k+1) (n-1))
```

> The function builds a *full binary tree* of depth n, i.e. a binary tree that has the maximum number of elements for a given depth (the equal case for the inequality above arises for full binary trees). The labels are calculated from some initial root label – when it is initialised to 1, we get a full binary tree labelled with integers from 1 to $2^n - 1$.

6.2. Exercises

2. Write an function taking a binary tree labelled with integers and returning their sum.

> A simple recursive solution works well here.
>
> ```
> let rec sum = function
>   | Lf             -> 0
>   | Br (x, lt, rt) -> x + sum lt + sum rt
> ```

3. Give the declaration of an OCaml datatype for arithmetic expressions that have the following possible shapes: floats, variables (represented by strings), or expressions of the form

$$-E \qquad \text{or} \qquad E + E \qquad \text{or} \qquad E \times E$$

Hint: recall how expressions differ from statements, and how their characteristic structure could be captured as a data type. It's a lot simpler than it may seem!

> As we saw before, datatypes consist of a list of constructors with arguments. The different constructors are various *alternatives* for the shape of the datatype: a list is *either* empty *or* a cons cell, a tree is *either* a leaf *or* a branch, a vehicle is *either* a lorry *or* a car *or* a motorbike, etc. Each alternative can be associated with one or more types given as the constructor arguments. Hence you can think of every constructor as representing a product of types (which, in OCaml, is the product type `'a * 'b` itself), and a datatype is a disjoint union/sum of these products as constructors. This is why OCaml-style datatypes are called *algebraic datatypes*: they are simply sums of products of types. The type

```
type ty = A of int * bool | B of string | C
```

can be thought of as the algebraic type

$$\texttt{ty} = (\texttt{int} \times \texttt{bool}) + \texttt{string} + \texttt{unit}$$

where OCaml's | is interpreted as the type-theoretic sum +, the product * is the type-theoretic product ×, and the nullary constructor simply represents the unit type. Unlike in OCaml, the constructors do not have explicit names – all we care about is a very simplified representation of the type. If you are familiar with set theory, you can think of a type as a set, and the operators + and × as the disjoint union and Cartesian product of sets.

But this notation is a bit confusing in this case. The sum and product mentioned in the question are different from the type-theoretic sum and product: the operations in the question refer to the *syntax* of the expression language. Where the algebraic view of datatypes comes in helpful is decoding the "specification" of the type. The question lists the alternatives for what shape a value of our `expr` type might take: it should be *either* a float, *or* a variable, *or* a negation of an expression, *or* sum of two expressions, *or* product of two expressions. From this we know that we will need five alternatives, i.e. five data constructors. Now to deduce the number and type of arguments to the constructors, we consider what we need to construct an appropriate value. To get a real expression (in our syntax), we need an argument of the type `real`. Similarly, to get a variable expression, we need a string (as per the question). To get a negated expression, we need an inner subexpression – as expected, `expr` is a recursive type. Similarly, to create the sum and product of two expressions, we need the two subexpressions. Putting everything together, we have the algebraic type

$$\texttt{expr} = \texttt{float} + \texttt{string} + \texttt{expr} + (\texttt{expr} \times \texttt{expr}) + (\texttt{expr} \times \texttt{expr})$$

where each term of the sum corresponds to the constructor, and the terms themselves are the arguments to the individual constructors. In OCaml syntax we can also give the constructors arbitrary names, so the datatype would look like:

```
type expr = Float of float
          | Var   of string
          | Neg   of expr
          | Plus  of expr * expr
          | Times of expr * expr
```

4. Continuing the previous exercise, write a function `eval : expr -> float` that evaluates an expression. If the expression contains any variables, your function should raise an exception indicating the variable name.

In the previous question we defined the *syntax* of our expression language: the formal grammar that describes the structure of the expressions that we write. However, a value of type `expr` is nothing more than a nested hierarchy of the constructors which – on their own – don't "do" or mean anything. For example, we know that the term

```
let e = Plus (Neg (Times (Float 2.0, Float 3.0)), Float 8.0);
```

represents the mathematical expression $-(2 \times 3) + 8$, but `e` is not *equal* to the expression `Float 2.0`. In fact, if we didn't have descriptive names for the constructors (which we can choose arbitrarily), we might not even know what an expression represents. What we want is to give a meaning to our expression language, which in computer science is usually called the *semantics* for a particular grammar or syntax. There are multiple ways to give a semantics to some syntax (all of which will be covered in the Computer Science course), but in our case we want an *evaluation function*, which reduces the expression to an irreducible value (this is related to the approach of denotational semantics). Writing evaluation functions for recursive expression languages is really simple and elegant, as the implementation closely follows the inductive definition of the `expr` type.

We begin with declaring an exception for expressions with free variables (so-called *open terms*): this is supposed to return the name of the variable, so the exception will have a string argument. Then, we define the `eval : expr -> float` function by pattern-matching on the expression, and essentially translate the constructors to their corresponding mathematical operation – the `Neg` constructor becomes negation, the `Plus` becomes the operator `+.`, etc. These operators work on `float` values, so we need to convert the subexpressions of type `expr` into `float` – but this is exactly what our `eval` function does, so we just call it recursively on the subexpressions. In the end, we get a function which, given any `expr` value, converts it into an OCaml arithmetic expression, which then gets evaluated to the corresponding floating-point number.

```
exception Variable of string

let rec eval = function
  | (Float f)        -> f
  | (Var v)          -> raise (Variable v)
  | (Neg e)          -> -. (eval e)
  | (Plus  (e1, e2)) -> eval e1 +. eval e2
  | (Times (e1, e2)) -> eval e1 *. eval e2;
```

Calling `eval` on the expression `e` above now correctly evaluates to the float `2.0`.

As simple as this looks, it has very deep implications and the technique is used all over computer science – in fact, OCaml itself can be defined by giving a datatype specifying its

> syntax, and an evaluation function which determines the semantics.

6.3. Optional questions

5. Prove the inequality involving the depth and size of a binary tree `t` from Page 53.

$$\forall \text{ trees } \texttt{t}.\ \text{count}(\texttt{t}) \leq 2^{\text{depth}(\texttt{t})} - 1$$

Hint: One way to do this is with a generalisation of mathematical induction called *structural induction*, where you analyse the *shape* (top-level constructor) of the tree `t`, and prove the property for the base case (leaf) and recursive case (branch). You can also try standard mathematical induction on some numerical property of the tree, but be careful with that you are assuming and what you are proving!

> The proof is doable via mathematical induction on the height of the tree, but to make it fully rigorous, you need to be careful with your proof goals and assumptions. The tree may not be balanced so in the inductive case ($\text{depth}(\texttt{t}) = k + 1$) you will only be able to apply the hypothesis to one of the subtrees (of depth k), but not the other. You can overcome this by doing *strong induction* (see the Discrete Maths course), or by proving a stronger theorem for perfectly balanced binary trees and deriving the above theorem as a corollary.
>
> Instead, we will try a more general proof technique that can be applied to any inductively defined datatype (in fact, mathematical induction is just structural induction on the type `type nat = zero | succ of nat`). To use structural induction to prove a predicate about every value of some type, we prove it for the base cases of the type, then prove it for the general cases, assuming that the predicate holds for the data arguments. In this case, the base case of the tree `t` is that it is a `Lf`. Then,
>
> $$\text{count}(\texttt{Lf}) = 0 \leq 2^{\text{depth}(\texttt{Lf})} - 1 = 2^0 - 1 = 0$$
>
> For the inductive case, assume that `t = Br (v, lt, rt)` and the predicate holds for the two subtrees, i.e.
>
> $$\text{count}(\texttt{lt}) \leq 2^{\text{depth}(\texttt{lt})} - 1 \qquad \text{and} \qquad \text{count}(\texttt{rt}) \leq 2^{\text{depth}(\texttt{rt})} - 1$$
>
> Then, we have
>
> $$\begin{aligned} \text{count}(\texttt{Br (v, lt, rt)}) &= 1 + \text{count}(\texttt{lt}) + \text{count}(\texttt{rt}) \\ &\leq 1 + (2^{\text{depth}(\texttt{lt})} - 1) + (2^{\text{depth}(\texttt{rt})} - 1) \\ &= 2^{\text{depth}(\texttt{lt})} + 2^{\text{depth}(\texttt{rt})} - 1 \end{aligned}$$
>
> where we used the two induction hypotheses to replace the count of the subtrees with the exponential of their depth. Now, assuming that the left subtree is deeper (without loss of generality), we have that
>
> $$2^{\text{depth}(\texttt{Br (v, lt, rt)})} = 2^{1+\text{depth}(\texttt{lt})} = 2 \times 2^{\text{depth}(\texttt{lt})}$$

> Going back to our previous inequality, we have that $2^{\texttt{depth(rt)}} \leq 2^{\texttt{depth(lt)}}$, so
>
> $$2^{\texttt{depth(lt)}} + 2^{\texttt{depth(rt)}} - 1 \leq 2 \times 2^{\texttt{depth(lt)}} - 1 < 2 \times 2^{\texttt{depth(lt)}}$$
>
> which establishes the required inequality
>
> $$\texttt{count(Br (v, lt, rt)} \leq 2^{\texttt{depth(Br (v, lt, rt))}}.$$

6. Give a declaration of the data type `day` for the days of the week. Comment on the practicality of such a datatype in a calendar application.

> The `day` datatype is a good example of an enumeration type: we simply give all possible values of the type, without any constructor arguments. As we have seven days in the week, we will have seven constructors.
>
> ```
> type day = Monday | Tuesday  | Wednesday | Thursday
>          | Friday | Saturday | Sunday
> ```
>
> The benefits of this representation is that it is clear and unambiguous: a function taking a `day` needs to handle these seven possibilities and only them. If we represented days with normal integers, we would need to worry about out-of-bounds numbers, representation conventions (e.g. is `Monday` `0` or `1`, etc.) and the function types would be less descriptive (e.g. the function `dayFromDate` in the next exercise would just have the type `int -> int -> int -> int`). However, integers admit arithmetic operations, such as adding or subtracting days (though we need to check bounds), which would need to be explicitly defined for our `day` datatype.

7. Write a function `dayFromDate : int -> int -> int -> day` which calculates the day of the week of a date given as integers. For example, `dayFromDate 2020 10 13` would evaluate to `Tuesday` (or whatever encoding you used in your definition of `day` above). *Hint*: Using Zeller's Rule might be the easiest approach.

> Zeller's Rule gives a simple arithmetic method of computing the day of the week for a given date. It converts the day, month, century and year to number, then computes the number of the day in the week that those numbers correspond to. Using our algebraic datatype representation, we need to convert this number into a value of type `day` – this can be done by pattern-matching or other ways (e.g. indexing into a list of those days).
>
> The algorithm first converts the dates into numbers, following specific rules that make the arithmetic simpler (e.g. months are shifted, numbering March as 1 and February as 12 – this way, the leap day is always at the end of the "year"). Finally, these numbers are plugged into the appropriate expression to get the number of the weekday, which is then converted to a `day`. We use the OCaml's integer division operator to implicitly take the integer part of the quotients, as described in the algorithm.

```
let numToDate = function
  | 0 -> Sunday    | 1 -> Monday
  | 2 -> Tuesday   | 3 -> Wednesday
  | 4 -> Thursday | 5 -> Friday
  | 6 -> Saturday

let dayFromDate yr mnth dy =
  let c = yr / 100
  and d = yr mod 100 - (if mnth < 3 then 1 else 0)
  and m = (mnth - 3) mod 12 + 1
  and k = dy
  in numToDate ((k + ((13 * m - 1) / 5) + d
                  + (d / 4) + (c / 4) - 2*c) mod 7)
```

7. Dictionaries and functional arrays

7.1. Conceptual questions

1. 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? How could we avoid the issue encountered in the first case?

```
1.      (Alice, 6)          2.      (Gerard, 8)
          /  \                        /      \
         Lf  (Tobias, 2)       (Alice, 6) (Lucy, 9)
               /      \          /  \     /     \
         (Gerald, 8) Lf        Lf  Lf    Lf   (Tobias, 2)
           /   \                                  /    \
          Lf  (Lucy, 9)                          Lf    Lf
                /   \
               Lf   Lf
```

The difference in the two is the order in which we add the elements to the tree. The binary ordering means that the first element added will always be the root, and if that element is the first element in the ordering, its left subtree will be empty. In the extreme case, we may end up with a non-branching binary tree which is effectively a list. Ideally we want the root of the tree to be the middle element of the ordering so the subtrees are balanced, but of course we can't know that a priori. This is exactly the same issue we had with choosing a pivot for quicksorting, so we can use similar heuristics: for example, choosing the median of 3/5/7 randomly chosen elements.

7.2. Exercises

2. Code an insertion function for binary search trees (with keys of `string` type, and values of polymorphic `'a` type). It should resemble the existing `update` function except that it should raise the exception `Collision` if the item to be inserted is already present. Now try modifying your function so that `Collision` returns the value previously stored in the dictionary at the given key. What problems do you encounter and why?

   > The insertion function is very similar to the `update` function.
   >
   > ```
   > exception Collision
   > let rec insert t (b: string) y = match t with
   >   | Lf -> Br ((b,y), Lf, Lf)
   >   | Br ((a,x), lt, rt) ->
   >           if b < a then Br ((a,x), insert lt b y, rt)
   >      else if a < b then Br ((a,x), lt, insert rt b y)
   >      else raise Collision
   > ```
   >
   > The problem with returning the previous value in the exception is that exceptions cannot be polymorphic: we would need to fix the type of the argument to `Collision`, which means the type of the values in the tree will have to match this type. We would give up the polymorphic nature of the `insert` function, which is undesirable.

3. Describe and code an algorithm for deleting an entry from a binary search tree. Comment on the suitability of your approach. There are two reasonable methods – one is simple, the other is efficient but a bit more tricky.

   > Locating the required node uses the same sequence of comparisons and recursive calls as the insertion and update operations. However, the difficulty with removing a node from a tree is that it might break up our tree: the two (possibly nonempty) subtrees would need to be reattached while retaining the ordering constraint of binary trees. Consider deleting the root of the tree (which is ultimately what any deletion will be like, once all the recursive calls reach the required element): we are left with a left and right subtree, where all elements in `lt` are less than all elements in `rt`. One way to combine the two trees is to attach `rt` as the right subtree of the rightmost lowermost element of `lt`: by the ordering condition, that element will be the greatest node in the left subtree, so all the greater elements in `rt` must be its right subtree.
   >
   > ```
   > let rec join t1 t2 = match t1 with
   >   | Lf -> t2
   >   | Br (n, lt, rt) -> Br (n, lt, join rt t2)
   >
   > let rec delete t x = match t with
   >   | Lf -> raise (Missing x)
   > ```

```
  | Br ((n,v), lt, rt) ->
          if x < n then Br ((n,v), delete lt x, rt)
     else if n < x then Br ((n,v), lt, delete rt x)
     else join lt rt
```

While this approach is quite simple, it's drawback is that it modifies the shape of the tree quite significantly: it makes the tree deep and unbalanced, which hinders performance.

The trick comes from our previous observation that the rightmost lowermost element of the left subtree is the greatest element in that subtree. This means that it must come immediately before the root element in the ordering of elements (indeed, inorder traversal would put these one after the other). Therefore, if a node has two nonempty subtrees, we can delete the node (while retaining the ordering) by replacing it with the "previous" element, the rightmost lowermost node of the left subtree. To find (and remove) this element in a tree, we can use the function below. It returns two results (the maximum element, and the tree without the maximum element) at the same time, but we can also define two independent functions to find the maximum and remove it from the tree (though this would require two traversals). When the right subtree is empty, the maximal element is the root node; otherwise we recurse into the right subtree.

```
let rec remove_max = function
  | Br (x, lt, Lf) -> (x, lt)
  | Br (x, lt, rt) ->
      let (m,t) = remove_max rt in (m, Br (x, lt, t))
```

We use the function above to locate and remove the maximal element in a tree, which we have to use when deleting a node with two nonempty subtrees. If a node has one subtree, it can be replaced by the root of its subtree (this handles the case when both subtrees are leaves, since then we replace the node with a leaf). This suggests the three patterns for our deletion function:

```
let rec delete t x = match t with
  | Lf -> raise (Missing x)
  | Br ((n,v), Lf, rt) -> if n = x then rt
     else if n < x then Br ((n,v), Lf, delete rt x)
                   else raise (Missing x)
  | Br ((n,v), lt, Lf) -> if n = x then lt
     else if x < n then Br ((n,v), delete lt x, Lf)
                   else raise (Missing x)
  | Br ((n,v), lt, rt) -> if n = x then
     let (m, lt2) = remove_max lt in Br (m, lt2, rt)
     else if n < x then Br ((n,v), lt, delete rt x)
```

```
                    else Br ((n,v), delete lt x, rt)
```

The final case uses the `remove_max` function above, getting the last element of the left subtree and putting it in place of the removed node. This approach, while a bit more complicated, keeps the tree balanced and makes the minimum number of changes to its structure (moving at most one node).

4. Write a function to remove the first element from a functional array. All the other elements are to have their subscripts reduced by one.

   The solution exploits the indexing convention of functional arrays: the first index is the root, all even indices are in the left subtree and all odd indices are in the right. If all the indices in the subtrees are divided by two (rounding down), we get back the same indexing convention. The parity separation means that if all indices are decreased by one, every odd-indexed element will be even-indexed, and vice versa. Thus, after deleting the first element and shifting everything back by an index, the whole right subtree (of odd-indexed elements) moves to the left unchanged, while the original left subtree moves to the right with a recursive call removing its root element (which becomes the root of the whole array, at position 1).

```
let top = function
  | Lf -> raise Subscript
  | Br (v,_,_) -> v

let rec pop = function
  | Lf -> raise Subscript
  | Br (_, Lf, Lf) -> Lf
  | Br (_, lt, rt) -> Br (top lt, rt, pop lt)
```

7.3. Optional questions

5. Show that the functions `preorder`, `inorder` and `postorder` all require $O(n^2)$ time in the worst case, where n is the size of the tree.

   The worst case for each case is when the tree is left-linear, i.e. when the right subtree of every node is a leaf. In that case, every append will have the form `xs @ [ ]`, with `xs` increasing linearly for every level. As appending is linear in the length of the first argument, we get the linear sum $n + (n-1) + \cdots 1$ which makes the complexity quadratic.

6. Show that the functions `preord`, `inord` and `postord` take linear time in the size of the tree.

   At every step, the functions examine one tree element and perform a constant-time consing operation. Hence the runtime is $1 + 1 + \cdot + 1 = n$ so the algorithm is linear.

8. Functions as values

8.1. Conceptual questions

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

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

> The functions introduced in this exercise are all *higher-order function combinators*: functions that take functions as arguments and return functions as results, thereby giving a flexible way of manipulating functions to fit our needs. They are all polymorphic functions, so we know that their implementations will be either unique or very limited in form (in this case the definitions are all fully determined by the type).
>
> There are two ways to read the types of these functions, which are equivalent by the right-associativity of the function type constructor `->`. We can either read them as functions of type `('a -> 'b -> 'c) -> 'a * 'b -> 'c` that take a function (e.g. `'a -> 'b -> 'c`) and various arguments (e.g. `'a * 'b`), then returning the result by applying the function to the arguments in some way (resulting in a `'c`). This is how we view the implementation of the functions. However, a more high-level way is to simply see these functions as combinators which modify functions in some way: `ucr : ('a -> 'b -> 'c) -> ('a * 'b -> 'c)` takes a function `f` of type `('a -> 'b -> 'c)` and returns a modified function of type `('a * 'b -> 'c)`. Therefore, the partial application `ucr f` is simply a function of type `('a * 'b -> 'c)`.
>
> **Swap** Swapping the order of two arguments of a function
>
> ```
> let sw f x y = f y x
> val sw : ('a -> 'b -> 'c) -> ('b -> 'a -> 'c) = <fun>
> ```
>
> Partial application is a very powerful feature of functional languages, and should be used as much as possible – it encourages abstraction and code reuse. However, one apparent limitation is that the order in which we can partially apply arguments is fixed: we cannot (by default) supply the second argument to create a function of the first argument. The trick is a higher-order function which swaps the order of the arguments: it takes a two-argument function and returns the same function, but with the order of the arguments swapped. This is reflected in the definition and the type as well.

For example, if we want to partially apply the list consing function `cons : 'a -> 'a list -> 'a list` to the tail of a list instead, and get a function which adds an element to the beginning, we use `sw cons : 'a list -> 'a -> 'a list` which has the tail as the first argument and the head as second. Then, `sw cons [2,3,4] : 'a -> 'a list` is a function that takes a head and conses it to `[2,3,4]`.

Composition Applying two functions one after another

```
let co g f x = g (f x)
val co : ('b -> 'c) -> ('a -> 'b) -> ('a -> 'c) = <fun>
```

Writing functional programs is about composing smaller functions into bigger ones. The actual operation of composition makes this concrete: it takes two functions and applies them one after the other. This is like the definition of the mathematical operation

$$(g \circ f)(x) = g(f(x))$$

The main benefit of this operation is *point-free style programming* which is covered in the last part of this exercise sheet. In brief, we can define functions by glueing together smaller functions, instead of explicitly defining the function on its arguments (or points). For example, we can define the (inefficient) `last` function (the head of the reverse of a list) as `val last = co head rev`.

Currying Converting from a tuple-argument function to a curried function

```
let cr f a b = f (a,b)
val cr : ('a * 'b -> 'c) -> ('a -> 'b -> 'c) = <fun>
```

As discussed above, *currying* is the operation of transforming an "old-style" two-argument function (taking a pair) into a "new-style" two-argument function (taking the first and second arguments one after the other). This operation can be made explicit with the function `cr`, which takes a non-curried function and returns a curried function. This is useful if we want to partially apply a non-curried function `f : 'a * 'b -> 'c`: as we know, we have to supply both arguments in the pair, but if we curry the function, we can partially apply it to just the first argument: `cr f : 'a -> 'b -> 'c`. For example, the `fst : 'a * 'b -> 'a` function returns the first element of a tuple. If we curry the function to get `cr fst : 'a -> 'b -> 'a`, we simply get the *constant* function that returns its first argument and ignores the second: `let const5 = cr fst 5 : 'a -> int`.

Identity Do-nothing function

```
let id x = x
val id : 'a -> 'a = <fun>
```

While not strictly a function combinator (and not obviously a useful function), the identity function nevertheless often comes in handy as the "do nothing" operation. One important property is that `id` is the only polymorphic function with the type `'a -> 'a`, which also means that given any type `T`, we know that there must be at least one function of type `T -> T`, namely `id` (this is true even if `T` is the so-called *empty type*, which has no inhabitants). The identity function also acts in a predictable way when given as an argument to higher-order function; in fact, one of the correctness properties of the `map` function for any "container" type (such as lists, sequences, trees, etc.) is that mapping the identity over a value must not change the value: `map id l = l`. Surprisingly enough, the `id` function and function composition form the basis for a recent, very abstract field of mathematics called *category theory*, which generalises diverse areas such as logic, algebra and set theory, and has surprisingly close ties with the theory of programming.

Application Applying the first argument to the second

```
let ap f x = f x
val ap : ('a -> 'b) -> 'a -> 'b = <fun>
```

The function application function is quite strange: all it does is take a function and an argument, and applies the function to the argument. But actually, an analogous function in the Haskell programming language (written as `f $ x` instead of `ap f x`) is probably the most often used construct in any piece of code. The reason for this is that in realistic programs we often end up calling functions on very complicated expression arguments, but given that the precedence of normal OCaml function application (denoted simply by juxtaposition, as in `f x`) is very high, we have to surround the argument with parentheses. As the nesting increases, the parentheses become quite obnoxious and the code can be difficult to read. Instead, we use the `ap` or `$` function, which has very low precedence – this means that its two arguments get evaluated before the function application happens. So instead of `f (...)` we write `f $ ...`, thereby avoiding one set of parentheses. Similarly, instead of writing nested parentheses like `f (g (...))`, we can do `f (g $ ...)` or `f $ g $ ...` or even `co f g $ ...`. Another interesting use for `ap` is reverse function application with `sw ap : 'a -> ('a -> 'b) -> 'b`. Partially applying this function to an `x` gives us a function that takes another function and applies it to `x`. A slightly contrived example of this is given at the end of this exercise sheet.

Uncurrying Converting a curried function to a tuple-argument function

```
let ucr f (a,b) = f a b
val ucr : ('a -> 'b -> 'c) -> ('a * 'b -> 'c) = <fun>
```

> Sometimes we have a curried function, but we actually want to *uncurry* it to get a function from pairs. The function `ucr` can be used to accomplish this. For example, given a function `f : 'a -> 'b -> 'c` and a list of pairs `l : ('a * 'b) list`, we cannot map `f` over `l` because `map f` has type `'a list -> ('b -> 'c) list`. However, `ucr f : 'a * 'b -> 'c`, so `map (ucr f) : ('a * 'b) list -> 'c list` is just what we need.

8.2. Exercises

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

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

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

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

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

> Straightforward translation of the specification into OCaml code.

```
let pairwise_s_i (x1,y1) (x2,y2) =
      (x1 < x2) && (y1 < y2)
let lex_s_i (x1,y1) (x2,y2) =
      (x1 < x2) || (x1 = x2 && (y1 < y2))
```

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

> We are looking for *higher-order functions* to combine two ordering relations `'a * 'a -> bool` and `'b * 'b -> bool` into a lexicographic ordering on pairs `('a * 'b) * ('a * 'b) -> bool`. As usual, this is just a direct translation of the mathematical definition into OCaml. Note how instead of `x1 <= x2` and `y1 <= y2`, we are using the parameterised ordering operations that are given to `lex` as arguments.

```
let pairwise o1 o2 (x1,y1) (x2,y2) =
      (o1 x1 x2) && (o2 y1 y2)
let lex o1 o2 (x1,y1) (x2,y2) =
      (o1 x1 x2) || (x1 = x2 && (o2 y1 y2))
val lex : ('a * 'a -> bool) -> ('b * 'c -> bool)
      -> (('a * 'b) * ('a * 'c) -> bool)
```

The inferred type of `lex` is interesting for two reasons. The inferred type of the first argument is `'a` in both cases, as the values need to be compared for equality. Second, it lets the two arguments of the second ordering to be different – we would usually not need this in an ordering, but we can see that the definition is more general.

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

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

This is a nice example of the higher-order programming style common in the functional paradigm: we build more complex functions by combining several smaller, simpler operations with higher-order function combinators. Notice how the list argument is not even mentioned – see more details in the last part of this exercise sheet. The type annotation is optional – without it, OCaml would infer a more general, (weakly) polymorphic type for the function.

```
let weird_sort : (string * (int * string)) list
              -> (string * (int * string)) list
  = insort (pairwise (<=) (lex (>) (<)))
```

3. Without using (or redefining!) `map`, write a function `map2` such that `map2 f` is equivalent to the composition `map (map f)`. Make use of nested pattern matching and `let`-declarations if needed.

One solution uses nested pattern-matching and local bindings: in the most general case, we call the function recursively on the tail of the outside list and the tail of the head element, pattern-match on the result, then add back the head with `f` applied to it.

```
let rec map2 f = function
  | [] -> []
  | []::xss -> [] :: map2 f xss
  | (x::xs)::xss ->
      let (fs::fss) = map2 f (xs::xss)
```

```
        in (f x :: fs) :: fss
```

4. The built-in type `option`, shown below, can be viewed as a type of lists having at most one element. (It is typically used as an alternative to exceptions.) Declare a function that works analogously to `map` but on `option` types rather than `list`s.

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

> Mapping is a very general operation that can be specialised to container types such as lists, trees, option types, etc. It can be seen as a way to "lift" an operation over a container structure: instead of applying a function to a list, we lift it over the list structure and apply it to the things contained in the list. In the same vein, we can lift a function `f : 'a -> 'b` over the `option` structure of a value `v : 'a option` to get a value `map_opt f v : 'b option`. If the option contains a value, we apply the function to the value and repackage it into `Some`. If not, there is nothing to apply the function to, so we just return `None` again. Note that in the `None` case, the input has type `'a option`, while the returned `None` has type `'b option` – as `None` is a nullary constructor, its type variable is not fixed.
>
> ```
> let map_opt f = function
>   | None   -> None
>   | Some x -> Some (f x)
> ```

8.3. Optional questions

5. Recall the making change function of Lecture 4:

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

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

> We replace `allc` with `map cons`, where `cons` is a curried version of `::`.

```
let cons x xs = x::xs
let rec change till amt = match till, amt with
  | _, 0  -> [ [] ]
  | [], _ -> []
  | c::till, amt ->
     if amt < c then change till amt
                else map (cons c) (change (c::till) (amt-c))
                     @ change till amt
```

Instead of declaring a new named function `cons`, we could have used an anonymous function `map (fun cs -> c::cs)`.

Very optional question

Make sure that you complete Question 8.1.1 before reading this exercise! Don't worry if you can't finish it, but do give it a try sometime – it shows you the real power of functional programming.

Pointfree (or *tacit*) programming is a style of writing functional programs by composing and combining smaller functions instead of defining a function by giving its value at every point (argument). In practice, point-free functions do not mention all of their arguments before the `=` so the expression after the `=` will be a function of the hidden arguments. The basic example is simplifying a function that calls another function on its argument:

```
let firstElem xs = List.hd xs
> val firstElem : 'a list -> 'a = <fun>
```

The value of the function `firstElem` on each of its points (arguments) `xs` is the head of `xs`. The property of *function extensionality* states that two functions are equal if their values are equal at every point. That is, with the definition above, `firstElem` has exactly the same behaviour as `List.hd` and it can therefore be simply defined as a value that equals `List.hd`. The types do not change, as `firstElem` simply inherits the type of `List.hd`.

```
let firstElem = List.hd
> val firstElem : 'a list -> 'a = <fun>
```

Similarly, pointfree style can be combined with partial application to create specialised functions from more general ones. A special case of this are the auxiliary functions we define for tail recursion: to get a function of the required type we need to specify the initial value of the accumulator in the auxiliary function. The most idiomatic way of doing this would be with partial application (as long as the accumulator is the first argument):

```
let rec sum_aux acc = function
  | [] -> acc
  | x::xs -> sum_aux (acc + x) xs
```

```
> val sum_aux : int -> int list -> int = <fun>
let sum = sum_aux 0
> val sum : int list -> int = <fun>
```

That is, the `sum` function is equal to `sum_aux` when partially applied to `0`. Note that we do not mention the list argument on either side, just like we didn't always mention the list argument of `insort` on Page 67.

Before you move on, I would recommend that you go through these and similar examples to make sure you understand how partial application and pointfree programming follows from currying. Feel free to write some notes about this.

The functions in Question 8.1.1 are all utilities for combining and transforming smaller functions. For example, `co g f` is the composition of two functions `f` and `g`, mathematically defined as

$$(g \circ f)(x) = g(f(x))$$

There is no built-in composition operator in OCaml, but to simplify writing pointfree code, it's worth defining it ourselves as an *infix* operator (so instead of `co g f` we can write `g << f`).

```
let (<<) g f x = g (f x)
> val (<<) : ('b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```

Composition is one of the fundamental ways of building larger functions out of smaller ones, the crux of functional programming. Notice that using composition brings the function application to the "top level" instead of nested in several levels of parentheses, which means it plays well with pointfree style programming.

```
let last xs = List.hd (List.rev xs)
let last xs = (List.hd << List.rev) xs
let last    = List.hd << List.rev
```

That is, getting the last element of a list is the same as reversing it first and then getting the head element. (Remember, this is not an efficient implementation of this function!)

Your task will be to transform the functions given below into pointfree style. You may (and should!) use the combinators from 8.1.1, various list functionals and list processing functions from lectures and exercises such as `map` and `sum`. You may also want to remove pattern matching if it becomes redundant due to your definition of choice, or rewrite the function entirely. Basically, make it as simple and as elegant as possible – all of the functions can be made into concise almost-one-liners.

1. Apply the function twice (you can leave the `f` argument).

   ```
   let applyTwice f x = f (f x)
   ```

   Remove the first and last elements of a list.

```
let peel xs = List.rev (List.tl (List.rev (List.tl xs)))
```

As a side-note, you can use `List.(...)` to open the `List` module locally in the parentheses to avoid having to write `List.` in front of every list function:

```
let peel xs = List.(rev (tl (rev (tl xs))))
```

> The function `applyTwice` is a simple example of function composition: applying a function twice is just applying it after itself, which is compositionally expressed as `f << f`. That is,
>
> ```
> let applyTwice f = f << f
> ```
>
> The `peel` function alternates `List.tl` and `List.rev` twice. Again, using composition, this can be rewritten as `(List.rev << List.tl) ((List.rev << List.tl) xs)`. But this is just applying the composite function `List.tl << List.rev` twice, so in fact
>
> ```
> let peel = applyTwice List.(tl << rev)
> ```

2. Count the number of vowels in a sentence represented as a list of strings. The following declarations can be used freely, no need to transform them.

```
let vowels = ['a'; 'e'; 'i'; 'o'; 'u']
let strToCharList s = List.init (String.length s) (String.get s)
let rec sum = function | [] -> 0 | x::xs -> x + sum xs
```

The functions `isVowel`, `getVowels` and `countVowels` can be combined into one short expression – try transforming them individually first, then write a single function that does the same thing as `countVowels`.

```
let isVowel ch = List.mem ch vowels

let rec getVowels = function
  | [] -> []
  | x::xs -> if isVowel x then x :: getVowels xs
                          else      getVowels xs

let rec countVowels = function
  | [] -> 0
  | w::ws -> List.length (getVowels (strToCharList w)) +
             countVowels ws
```

> To simplify `isVowel`, we would like to partially apply `List.mem` (the list membership function) to `vowels`, but it is the second argument. This is exactly what the function argument swapping function `sw` can be used for:
>
> ```
> let isVowel = sw List.mem vowels
> ```
>
> `getVowels` is a simple instance of list filtering:
>
> ```
> let getVowels = List.filter isVowel
> ```
>
> In the function `countVowels` we apply three functions to the head element of the list, then add the number to the recursively computed sum of vowels in the tail. We can achieve the same result by mapping the function `List.length << getVowels << strToCharList` over the list of words to get a list of vowel counts, then adding together the elements of that list with `sum`. To make the function pointfree, we compose the mapping and summing with `<<`.
>
> ```
> let countVowels =
>       sum << List.map (List.length << getVowels << strToCharList)
> ```
>
> Given that the first two functions are quite simple, we can inline them to make one closed expression (using the `List.(...)` syntax to open the `List` module):
>
> ```
> let countVowels = List.(
>       sum << map (length << (filter (sw mem vowels))
>           << strToCharList))
> ```
>
> We can see how even a fairly complicated expression can be implemented in a purely compositional, pointfree style. That said, this is not necessarily what you *should* do!

3. Quite contrived, but also quite neat. In this case you can keep the `x` argument, but change the function so that `x` only appears once in the body!

```
let calc x = [x; x +. 1.0; 2.0 *. x; x *. x; x /. 2.0;
              Float.pow 2.0 x; Float.sin x; Float.cosh x]
```

> Pointfree style is often a balancing act between conciseness and readability: forcing a function to be pointfree may often result in a messy, unreadable expression, where the "plumbing" needed to get the implicit arguments to their right place hides the actual workings of the function. The `calc` function can also be made pointfree, but it would be rather complicated. However, we can simplify (?) it to only refer to the argument once in the RHS expression.

Every expression in the list is some `float`-valued function of `x`. Given that functions are values, we can abstract out the argument `x` and create a *list of functions* of type `(float -> float) list`. To do this, we can create helper functions at the top level, use anonymous functions, or the function combinators and existing OCaml operators. For demonstration purposes, I will use the latter two techniques (though they are clearly rather undesirable here, the function itself is kinda silly anyway).

```
[ id; (+.) 1.0; ( *.) 2.0; (fun y -> y *. y); sw (/.) 2.0;
  Float.pow 2.0; Float.sin; Float.cosh ]
```

For example, instead of `x : float`, we write `id : float -> float` and instead of `x /. 2.0 : float`, we swap the arguments of the division operator (treated as a function by wrapping it in parentheses) and partially apply it to the denominator `2.0` to get `sw (/.) 2.0 : float -> float` (other languages, such as Haskell, offer shorthand notation for partially applied operators, so this would simply be written as `(/ 2.0)`). Note the extra space needed in `( *.)`, otherwise OCaml interprets `(*` as a comment.

Now the question is: how do we apply all these functions to a single argument? We essentially want to turn a list of functions into a list of floats, using some operation of type `(float -> float) list -> float list`. One function that can have this type is `map f` for some `f : (float -> float) -> float`: that is, by applying `f` to every function in the list, we get the list of values resulting from applying every function in the list to `x`. The mapped function `f` therefore takes a function (an element of the list) and applies it to `x`, so `f = fun g -> g x`. Can we write this without using anonymous functions? Indeed we can: this is where the strange "function application function" `ap` comes in handy. Remember that `ap : ('a -> 'b) -> 'a -> 'b`, and swapping the arguments around we get the reverse function application function `sw ap : 'a -> ('a -> 'b) -> 'b`. Partially applying this to `x` fixes the type to `sw ap x : (float -> float) -> float`, which is exactly the type we need. In short, `sw ap x` is a function that takes another function and applies it to `x`. Thus our final answer is

```
let calc x =
  List.map (sw ap x) [id; (+.) 1.0; ( *.) 2.0;
                      (fun y -> y *. y); sw (/.) 2.0;
                      Float.pow 2.0; Float.sin; Float.cosh]
```

9. Sequences and laziness

9.1. Conceptual questions

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

```
let rec concat = function
```

```
  | []    -> []
  | l::ls -> l @ concat ls
```

> There are two problems that may arise with the obvious implementation below. If any of the sequences is infinite, the concatenation will never proceed past this sequence. There is no problem if there are an infinite number of nonempty but finite sequences, as they will be diligently consumed and streamed out one by one in an infinite lazy sequence. An error case however is an infinite sequence of `Nil`s, which will trigger the second pattern repeatedly and lead to nontermination (without a stack overflow, as the body is a tail call).
>
> ```
> let rec concat = function
>   | Nil                      -> Nil
>   | Cons (Nil, yf)           -> concat (yf ())
>   | Cons (Cons (x, xf), yf) -> Cons (x, fun () ->
>                                  concat (Cons (xf (), yf)))
> ```
>
> The first problem can be solved by a generalised "interleave" function that looks at the heads of every list one-by-one, so it will never get stuck on an infinite list. However, this would not work for an infinite sequence of finite lists, as the second elements of the sub-sequences will never be reached.

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

> Lazy lists are lists where the tail is evaluated lazily: it is not touched if the tail elements are not required, and can even be infinite. OCaml does not support lazy lists due to its eager evaluation strategy: it always evaluates the tail of the list and keeps all the elements in memory, even if we only need the first few elements. We simulate lazy lists using *sequences*, which have a tail function instead of the tail: the tail function is a value so it's not evaluated by default, but we can force evaluation by applying it to ( ) (the only inhabitent of the type `unit`) and get the tail sequence (of which only the head is available). This trick enables us to put the tail behind a "wall", and only look at it (evaluate it) if we need to.

9.2. Exercises

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

> As before, the aim of the mapping is to lift a function `f : 'a -> 'b` to sequences: `map_seq f : 'a seq -> 'b seq`. The function is very similar in structure to the list version of `map`, though we have to do some extra work to handle the tail function.
>
> ```
> let rec map_seq f = function
>   | Nil          -> Nil
>   | Cons (x, xf) -> Cons (f x, fun () -> map_seq f (xf()))
> ```

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

   The datatype for lazy binary trees combines the structure of normal binary trees with the delayed evaluation trick of sequences: instead of subtrees, we have subtree functions.

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

   The flattening function resembles inorder tree traversal, in that we flatten the (lazy) subtrees and combine them with the root. However, given that the subtrees may be infinite, we cannot use stream concatenation; instead, we use the `interleave` function to ensure that both subtrees get involved. The order of the elements will be somewhat unusual, but we were not required to adhere to any specific ordering.

```
let rec flatten = function
  | Lf -> Nil
  | Br (n, ltf, rtf) -> Cons (n, fun () ->
                interleave (flatten (ltf())) (flatten (rtf())))
```

9.3. Optional questions

5. 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`.)

   The main difference between the list and sequence implementation is that `map` and append need to be replaced with sequence-specific variations. One simple way is to use an auxiliary function which performs the map and append at the same time, but mapping only over the first sequence, and taking the second sequence as a function, thereby delaying the evaluation.

```
let rec mapapp f s yf = match s with
  | Nil -> yf ()
  | Cons (x, xf) -> Cons (f x, fun () -> mapapp f (xf()) yf)
```

```
let rec change till amt = match till, amt with
  | _, 0  -> Cons ([], fun () -> Nil)
  | [], _ -> Nil
  | c::till, amt ->
     if amt < c then change till amt
                else mapapp (cons c) (change (c::till) (amt-c))
                            (fun () -> change till amt)
```

6. 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]`, ...

   The idea is that given a list of length n, we create two new lists of length $n+1$ by attaching either `0` or `1`. Starting with `[]`, we get `[0]` and `[1]`, then doing the same on these two gives us `[0;0]`, `[0;1]`, `[1;0]` and `[1;1]`, which are all possible two-element lists. Continuing the same for each generation will give us a complete list of all possible lists of `0` and `1` (or binary numbers).

   The following function encapsulates this idea: given an initial list `xs`, it returns the (infinite) sequence starting with the given list, then the interleaved sequences of of the recursive calls on `0::xs` and `1::xs`. We need the `interleave` function because each of the recursive calls returns an infinite sequence.

```
let rec binaryLists xs =
    Cons (xs, fun () -> interleave (binaryLists (0::xs))
                                   (binaryLists (1::xs)))
let allBinaryLists = binaryLists []
```

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

   One simple, but rather inefficient solution is to filter the result of the previous exercise, `allBinaryLists`, removing every non-palindrome list. Of course, the longer the list, the "rarer" the palindromes, so the number of lists skipped increases exponentially. Instead, we can *generate* the palindromes from `allBinaryLists` in the following way: given a list `xs`, we can create the palindromes `xs @ rev xs`, `xs @ (0 :: rev xs)` and `xs @ (1 :: rev xs)`. That is, every binary list gives rise to three unique palindromes: one of even length and two of odd length, with either a `0` or a `1` in the middle. The following function accomplishes this, given an initial nonempty sequence:

```
let rec genPalindromes (Cons(x, xf)) =
```

```
                    Cons (x @ List.rev x,
          fun () -> Cons (x @ (0 :: List.rev x),
          fun () -> Cons (x @ (1 :: List.rev x),
          fun () -> genPalindromes (xf())))))
let allBinaryPalindromes = genPalindromes allBinaryLists
```

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

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

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

   Straightforward translation of the list operations to sequences.

```
let rec filterS p = function
  | Nil -> Nil
  | Cons (x, xf) ->
          if p x then Cons (x, fun () -> filterS p (xf()))
                 else filterS p (xf())
let rec zipWithS opr xs ys = match xs, ys with
  | Nil, _ -> Nil
  | _, Nil -> Nil
  | (Cons (x, xf)), (Cons (y, yf)) ->
     Cons (opr x y, fun () -> zipWithS opr (xf()) (yf()))
```

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

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

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

Both of these are infinite sequences generated from some initial value. The first one takes a sequence and returns a sequence of squared elements; when called on `from 0`, it gives the infinite sequence of square numbers. Note that the same can be achieved by using `map_seq`, applying the squaring function to every element of `from 0`.

The second is more interesting, as it does not start from an initial value since it's not even a function: `f` is a *recursively defined value*. This is quite unusual, but the technique arises naturally when dealing with lazy infinite data structures, where the sub-structure is some modified version of the whole structure. The sequence starts off with `0` and `1`, then we perform pairwise addition (with `zipWithS (+)`) on the sequence itself, and its tail. Hence, the third value will be `0 + 1 = 1`, the fourth will be `1 + 1 = 2`, then `1 + 2 = 3`, `2 + 3 = 5`, `3 + 5 = 8` and so on, giving us the Fibonacci sequence. This definition exploits the property that pairwise adding the elements of the Fibonacci sequence with itself shifted by one, we also get the elements of the Fibonacci sequence:

```
  0, 1, 1, 2, 3, 5,  8, 13
+    0, 1, 1, 2, 3,  5,  8
=    1, 2, 3, 5, 8, 13, 21
```

The value `f` uses this property as a way to generate the elements of the sequence by initialising the first two elements, then performing pairwise additions. Even though the sequence is referencing itself as it is being constructed, laziness means that we only ask for elements that have already been created, so there is no inconsistency. This almost-one-liner is often used to show off the power of lazy computation.

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

In previous examples, the infinite stream was generated from some existing stream (e.g. square numbers), or the stream itself (Fibonacci sequence). Prime numbers are different in that they cannot be generated from a simple stream or defined recursively: they are irregular and there is no known algorithm for finding the k-th prime number. The best we can do is filtering – this is exactly what the Sieve of Eratosthenes does.

The inner function `sieve_aux` does most of the work. It takes an initial input stream, and returns a stream starting with the head of the input, but continuing by the recursively constructed tail filtered by removing everything that is divisible by the head. That is, we add the first element to the list of primes, and cross out everything divisible by this prime. This would of course only work if the first element is a prime, so we initialise the list with the sequence starting from `2`.

```
let sieve =
  let rec sieve_aux (Cons (p, pf)) = Cons (p, fun () ->
          sieve_aux (filterS (fun x -> x mod p <> 0) (pf())))
  in sieve_aux (from 2)
```

It is worth noting that this is far from an efficient method of generating the prime numbers, but it is quite amazing how it's all done using a few simple lines of code.

10. Queues and search strategies

10.1. Conceptual questions

1. Suppose that we have an implementation of queues, based on binary trees, such that each operation takes logarithmic time in the worst case. Outline the advantages and drawbacks of such an implementation compared with one presented in the notes.

   Binary tree operations usually take logarithmic time due to the branching structure of trees. This is a constant cost: although each operation may take more or less time (depending, for example, on the location of the element that we are searching for), the worst-case time complexity for the operations is logarithmic. In contrast, the list implementation of queues is constant *amortised* cost: we have to perform linear-time normalisation in the worst case, but most of the time the constant-time consing and unconsing operations are sufficient. The difference is that the second implementation has two modes of operation, one of which is more expensive but occurs rarely; the first implementation has just normal binary tree lookup which might take longer or shorter depending not on the structure of the collection, but on the element we are looking for. The binary tree version is therefore logarithmic, while the list implementation is constant amortised: while we might need to perform expensive normalisation, this "evens out" over the lifetime of the collection. That said, if some performance guarantees are required (e.g. in real-time critical systems), it may be worthwhile to give up constant amortised cost for a more consistent runtime that doesn't spike at hard-to-predict intervals.

2. The traditional way to implement queues uses a fixed-length array. Two indices into the array indicate the start and end of the queue, which wraps around from the end of the array to the start. How appropriate is such a data structure for implementing breadth-first search?

   While arrays allow for constant-time access to any element (including the front and end of the queue), its length is fixed: we would have a limit on what size tree we can traverse, or need to initialise a very large array that might end up being unnecessary for our purposes.

3. Why is iterative deepening inappropriate if $b \approx 1$, where b is the branching factor? What search strategy would make more sense in this case?

> A tree with branching factor $b \approx 1$ is almost like a list with one element at every depth. Given that iterative deepening is appropriate when the number of nodes at each level increases exponentially, it would be wasteful for a linear structure such as a list. Normal depth-first or breadth-first search is suitable.

4. An interesting variation on the data structures seen in the lectures is a *deque*, or *double-ended queue* [1]. A deque supports efficient addition and removal of elements on both ends (either its front or back). Suggest a suitable implementation of deques, justifying your decision.

> Double-ended queues must support enqueuing and dequeuing operations on both ends, so we need to find an implementation that gives constant-time access to both the "front" and "back" of the deque. Conveniently, our double list implementation for queues would work nicely for deques as well, as we can cons to and uncons from both halves of the list. The only difference is the normalisation algorithm: as we can get elements from both ends of the deque, we cannot just move every element to the first list and leave the second one empty. The alternative is to evenly split the normalised list between the two halves, so we always have elements we can access in constant time on both ends.

10.2. Exercises

5. Mathematical sets can be treated as an abstract data type for an unordered collection of unique elements. One approach we may take is to represent a set as an *ordered list* without duplicates: $\{5, 3, 8, 1, 18, 9\}$ would become the OCaml list `[1;3;5;8;9;18]`. Code the set operations of membership test, subset test, union and intersection using this ordered-list representation. Remember that you can assume the ordering invariant for the inputs (and thereby make your functions more efficient), and your output should maintain this invariant.

> The membership test is similar to the usual `member` function for lists, except we can "short-circuit" if the first element is less than the head of the list. Given that the lists are ordered (here we assume an increasing order), the head of the list must be its smallest element; if we ask if an element smaller than the smallest element is in the list, we can say "no" right away. Hence we can add the `y < x` constraint as a conjunct to the recursive call, so we stop as soon as an element is found or if the element cannot be in the list.
>
> ```
> let rec member x = function
>   | []    -> false
>   | y::ys -> (x = y) || (y < x && member x ys)
> ```
>
> An empty set is a subset of anything, and nothing is a subset of the empty set. The general case checks the heads of the lists and makes the appropriate recursive call. We make use of the ordering constraint by comparing the heads: if `x < y`, the first list contains an element

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

that cannot be in the second list, so we can short-circuit.

```
let rec subset xs ys = match xs, ys with
  | [],    _     -> true
  | _ ,    []    -> false
  | x::xs, y::ys -> (y <= x) && if x = y then subset xs ys
                                         else subset (x::xs) ys
```

Set union on ordered lists eliminates all the membership tests, as we can determine the elements contained by the union by simple comparison. The same can be done with set intersection as well.

```
let rec union xs ys = match xs, ys with
  | [],    ys    -> ys
  | xs,    []    -> xs
  | x::xs, y::ys ->
          if x < y then x :: union xs (y::ys)
     else if x > y then y :: union (x::xs) ys
     else               x :: union xs ys

let rec inter xs ys = match xs, ys with
  | x::xs, y::ys ->
          if x < y then inter xs (y::ys)
     else if x > y then inter (x::xs) ys
     else               x :: inter xs ys
  | _, _ -> []
```

10.3. Optional questions

6. Implement deques as an abstract data type in OCaml (including a type declaration and suitable operations). Estimate the amortised complexity of your solution.

As explained before, we can use the same double list implementation we gave for queues, with similar operations. The only function that needs changing is normalisation, as we want to evenly divide the elements between the two lists when either of the lists becomes empty. To accomplish this, we need to find the halfway point of the list where we can perform a split, which is a linear operation. One slight efficiency improvement we can make is a common trick when dealing with sized collections: instead of recalculating the length every time we need it, we keep track of it in the data structure itself as a separate field. Then, a `length` operation simply returns this known value, and we don't need to find the length of the list in the normalisation function. There might be some extra bookkeeping required in the definitions below, but the point of abstract data types is that the actual

implementation is hidden from view.

We make use of the helper function `split` which splits a list into two at a given index. Here we give a definition from scratch, but the result should have the property that `split l i = (take l i, drop l i)`. In `norm` we have two cases which need to be handled: when either the first or second list is empty. In those cases, we split the other list in half, and reverse one half based on which end of the deque we are looking at. The other operations are all straightforward, though we need to increment or decrement the length if we enqueue or dequeue and element.

```
let rec split xs n = match xs, n with
  | [],    _ -> ([], [])
  | x::xs, 0 -> ([], x::xs)
  | x::xs, i -> match split xs (i-1) with
                  (ys, zs) -> (x::ys, zs)

type 'a deque = D of 'a list * 'a list * int

let norm = function
  | D ([], tls, l) -> (match split tls (l / 2) with
                       (h1, h2) -> D (List.rev h2, h1, l))
  | D (hds, [], l) -> (match split hds (l / 2) with
                       (h1, h2) -> D (h1, List.rev h2, l))
  | d -> d

let dnull = function (D ([],[],0)) -> true | _ -> false
let enqFront (D (hds, tls, l), x) = norm (D (x::hds, tls, l+1))
let enqEnd   (D (hds, tls, l), x) = norm (D (hds, x::tls, l+1))
let deqFront (D (x::hds, tls, l)) = norm (D (hds, tls, l-1))
let deqEnd   (D (hds, x::tls, l)) = norm (D (hds, tls, l-1))
let length   (D (_, _, l)) = l
```

11. Elements of procedural programming

11.1. Conceptual questions

1. What is the effect of `while (C1; B) do C2 done`? Where would such a formulation be useful?

   The sequencing operator `;` executes the commands left-to-right, and ignores the return value of all statements apart from the last one, which becomes the return value of the whole sequencing. Hence `C1` is executed on every iteration, but its value is replaced by the value of the Boolean condition `B`. The body of the while loop, `C2`, is executed on every iteration as well. In addition, `C1` is executed before the Boolean condition, so its side-effects are

performed even if B evaluates to `false` right away. Therefore the expression is equivalent to `C1; while B do (C2; C1) done`. This formulation would be useful whenever we want to repeat instructions at least once; for example, making C2 the empty instruction would result in a construct equivalent to a do-while loop, which checks the Boolean condition after the first iteration.

11.2. Exercises

2. Comment, with examples, on the differences between an `int ref list` and an `int list ref`. How would you convert between the two? *Hint*: `(!) : 'a ref -> 'a` is a function.

   The first one is a list of references, e.g. `[ref 1, ref 2, ref 3] : int ref list`. Each reference can be individually assigned, but the list itself is immutable. The second one is a reference to a list, e.g. `ref [1,2,3] : int list ref`. We cannot change the elements individually, but we can change the whole list.

   Converting from a list of references to a reference to a list would involve dereferencing the list elements, and creating a reference to the resulting list. Since `(!) : 'a ref -> 'a` is just a function, we can map it over elements of a list:

   ```
   let reflistToListref rs = ref (List.map (!) rs)
   ```

   The other way is similar, except we dereference first and map `ref` after:

   ```
   let listrefToReflist lr = List.map ref (!lr)
   ```

   Of course, working with state and mutability means that the behaviour of these functions may not always be predictable.

3. Write a version of the efficient `power` function (Page 10) using `while` instead of recursion.

   Instead of tail-recursion, we use imperative iteration with a `while`-loop. We initialise an accumulator reference `accr` and references for our two inputs to allow mutability. What was an argument to the recursive call before becomes a reference assignment.

   ```
   let power x n =
     let accr = ref 1.0
     and xr = ref x
     and nr = ref n
      in while !nr <> 1 do
           if !nr mod 2 = 0
           then (xr := !xr *. !xr; nr := !nr / 2)
           else (accr := !accr *. !xr;
                 xr := !xr *. !xr; nr := !nr / 2)
         done;
   ```

```
        !accr *. !xr
```

4. Write a function to exchange the values of two references, `xr` and `yr`.

   While there are some type-specific hacks to perform the exchange "in-place", we generally need a temporary reference to avoid overwriting a value that we need to use later.

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

5. Arrays of multiple dimensions are represented in OCaml by arrays of arrays. Write functions to (a) create an $n \times n$ identity matrix, given n, and (b) to transpose an $m \times n$ matrix. Identity matrices have the following form:

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

   In both cases we make use of the `Array.init : int -> (int -> 'a) -> 'a array` function, which initialises an array with values calculated from the array index using a higher-order function argument. Matrices can be created by nesting these functions together, and using the current matrix index in the function argument of the inner `init` function. For example, an element of an identity matrix is 1 if its row and column index is equal, and 0 otherwise.

   ```
   let ident n = Array.init n (fun i ->
                 Array.init n (fun j ->
                   if i = j then 1 else 0))
   ```

   There are several approaches to transpose a matrix: we can use references to perform the transposition in-place, but `Array.init` allows us to use a more elegant, functional solution. Matrices are represented as an array of rows, so getting a single row of the matrix is just indexing into the outer matrix. Selecting an individual column c is harder, since we need to return the c^{th} element of each of the rows. However, if we have a function that can do that, we can transpose the matrix by selecting the columns of the matrix and turning them into rows (by simply putting them in an array).

   The helper function below uses `init` to return a given column of the input matrix. Given an $n \times m$ matrix, it returns an array of size n (which is just the length of the outer array), with its elements being the c^{th} element of the individual rows. "Looping" through the rows is done using `init` and `get`.

   ```
   let getCol c arr = Array.(init (length arr)
   ```

```
                                    (fun i -> get (get arr i) c))
```

The actual `transpose` function now uses `getCol` to create a new array of columns. The size of the outer array is the length of any of the inner arrays, and its r^{th} element is just the r^{th} column of the input matrix.

```
let transpose arr = Array.(init (length (get arr 0))
                                (fun r -> getCol r arr))
```