Foundations of Computer Science

Supervision 3 – Solutions

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:

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

> 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]
```