Foundations of Computer Science

Exercises

2020

Contents

1. Introduction to programming

1.1. Conceptual questions

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

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!)

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)`

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

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.

   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.

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 $x = 0.1$.

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`.

2. Recursion and efficiency

2.1. Conceptual questions

1. Add a column to the table shown in Slide 206 with the heading *60 hours*.
2. Use a *recurrence relation* to find an upper bound for the recurrence given by $T(1) = 1$ and $T(n) = 2T(n/2) + 1$. You should be able to find a tighter bound than $O(n \log n)$. Prove that your solution is an upper bound for all n using mathematical induction.

2.2. Exercises

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

2.3. Optional questions

4. 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 $f(n) = O(g_1(n) + \cdots + g_k(n))$.

3. Lists

3.1. Conceptual questions

1. Explain why seeing the following expressions in OCaml code should be a cause for concern:
   - `1 :: [2, 3, 4]`
   - `"hello " @ "world"`
   - `xs @ [x]`
   - `[x] @ xs`
   - `hd xs + length (tl xs)`
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?

3.2. Exercises

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

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.

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], []]`.

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`.

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?

4. More on lists

4.1. Conceptual questions

1. 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.

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 "return their results" wasn't a requirement?

4.2. Exercises

3. a) Use the `member` function on Page 32 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.

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).

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?

4.3. Optional questions

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

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

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

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?

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

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

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.

5.2. Exercises

3. Implement bubble sort in OCaml.

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.

   b) Implement selection sort using OCaml.

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))
```

6.2. Exercises

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

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

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

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!

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.

6.3. Optional questions

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

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

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!

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.

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.

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?

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?

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.

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 cost of this operation should be linear in the size of the array.

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.

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

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?

```ocaml
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>
```

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 pairwise and lexicographic ordering for the type of pairs `string * int`.
   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.
   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:
$$(s_1,(m,s_2)) < \left(s_1',(n,s_2')\right) \iff s_1 \le s_1' \ \land \ \left(m > n \ \lor \ (m = n \ \land \ s_2 < s_2')\right)$$

3. Without using `map`, write a function `map2` such that `map2 f` is equivalent to the composition `map (map f)`. The obvious solution requires declaring two recursive functions. Try to get away with one by exploiting nested pattern-matching.

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 an analogue of the function `map` for type `option`.

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

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.

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
```

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?

9.2. Exercises

3. Code an analogue of map for sequences.
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.

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.)
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], ...

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`.

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>
```

   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)))
```

   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".

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 69.

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))))
   ```

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
```

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]
```

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.

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?

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?

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.

10.2. Exercises

5. Write a version of the function breadth shown on Page 88 using a nested let construction rather than case.

6. 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.

[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.

10.3. Optional questions

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

11. Elements of procedural programming

11.1. Conceptual questions

1. Comment, with examples, on the differences between an `int ref list` and an `int list ref`. How would you convert between the two?

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

11.2. Exercises

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

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

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:

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