Foundations of Computer Science

Supervision 1 – Solutions

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 blah
> ```
>
> 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. Add a column to the table shown in Slide 206 with the heading *60 hours*.

Complexity	1 second	1 minute	1 hour	60 hours	Gain
n	1000	$60,000$	$3,600,000$	$216,000,000$	$\times 60$
$n \log n$	140	4893	$204,095$	$8,367,895$	$\times 41$
n^2	31	244	$1,897$	$15,176$	$\times 8$
n^3	10	39	153	612	$\times 4$
2^n	9	15	21	27	$+6$

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.

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

3. 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

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

One way to show this is to notice that $a_1 g_1(n) + \cdots + a_k g_k(n)$ is bounded by $ag_1(n) + \cdots + ag_k(n)$ where $a = \max\{a_1, \ldots, a_k\}$. As O-notation gives upper bounds, we have that $f(n) = O(ag_1(n) + \cdots + ag_k(n)) = O(a(g_1(n) + \cdots + g_k(n)))$, 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. Explain why seeing the following expressions in OCaml code should be a cause for concern:

   - `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.

   - `hd xs + length (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.