Foundations of Computer Science

Supervision 1

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?