Computer Science 320 (Spring, 2020)
Principles of Programming Languages

Programming Assignment 2: Recursion, Pattern Matching, Data Types

Out: Tuesday, February 11, 2020
Due: Friday, February 21, 2020, by 11:59 pm

Problem 1. The nth Tetranacci number T_n is mathematically defined as follows:

$$\begin{aligned} T_0 &= 0 \\ T_1 &= 1 \\ T_2 &= 1 \\ T_3 &= 2 \\ T_n &= T_{n-1} + T_{n-2} + T_{n-3} + T_{n-4}. \end{aligned}$$

Note carefully that the inductive (or recursive) part in this definition returns the value of T_n based on the four preceding values $\{T_{n-1}, T_{n-2}, T_{n-3}, T_{n-4}\}$. Your task is to translate the preceding definition into a recursive OCaml function `tetra1 : int -> int` which accepts an integer n (you may assume that $n \geq 0$), and computes the n-th Tetranacci number (don't worry about efficiency, only about making your OCaml code as close as possible to the mathematical definition of T_n).

Ungraded Bonus:[1] What is the approximate number of times this function will be called on an input of size n? Express this approximate number as a function of n.

Problem 2. You will now implement a different and more efficient version of the function `tetra1`, called `tetra2 : int -> int`, for computing Tetranacci numbers.

(a) Implement an OCaml function `sum_top_4 : int list -> int list`, which takes a list of integers and adds the sum of the top four elements to the head of the list (e.g. the input list `[1;1;1;1;3;2]` should become the output list `[4;1;1;1;1;3;2]`).

(b) Implement a recursive OCaml function `ascending : int -> int list`, which accepts an integer n as input (again, assume $n \geq 0$), and returns a list of integers from 0 to n in ascending order.

[1] You should be able to answer this question if you took, or are now taking, the course CS 330 "Introduction to Analysis of Algorithms" (or an equivalent course at another university).

(c) Implement a recursive OCaml function `tetra2 : int -> int`, which computes the nth Tetranacci number in "*time linear in n*", i.e. it computes T_n by making one recursive call (instead of 4 recursive calls as it was for function `tetra1` in Problem 1). Put differently, the number of recursive calls that `tetra2` makes is linear in n.

Problem 3. The Fibonacci k-step numbers are a generalization of the Fibonacci and Tetranacci sequences, where $F_i = 0$ for $i \leq 0$, $F_1 = 1$, $F_2 = 1$, and for every $j \geq 3$:

$$F_j = F_{j-1} + F_{j-2} + \cdots + F_{j-k},$$

i.e. F_j is the sum of the k previous numbers in the sequence. You will implement an OCaml function `fib_k_step : int -> int -> int` for computing these numbers efficiently.

(a) Implement an OCaml function `sum_top_k : int -> int list -> int`, which accepts an integer k along with a list of integers, and sums up the first k elements of the list. If the list has length less than k, simply sum all the elements in the list (assume that an empty list evaluates to a sum of 0).

(b) Implement an OCaml function `fib_k_step : int -> int -> int`, which accepts two integers n and k and, for a fixed k, computes in "*time linear in n*" the nth Fibonacci k-step number.[2]

For the three next problems you will deal with the data type of *polymorphic binary trees*. The solution to each part is meant to be very short (one to four lines). You may (and should) use functions from earlier parts to solve later parts. Unless otherwise specified, your solutions may utilize functions from the standard OCaml library as well as material from lecture notes.

Problem 4. Define a polymorphic data type called `'a binTree` as follows:

```
type 'a binTree =
    | Leaf
    | Node of 'a * ('a binTree) * ('a binTree)
```

Trees of this type will contain a single piece of data of type `'a` at each node, and no data at their leaves.

(a) Define an OCaml function

```
mapT : ('a -> 'b) -> 'a binTree -> 'b binTree
```

which operates on trees just as `map` operates on lists. In other words, `mapT` takes a function and applies it to every data item of type `'a` in a tree of type `'a binTree`.

[2] *Hint*: Do part (c) of Problem 2 before attempting part (b) of Problem 3; the latter generalizes the former.

(b) Define an OCaml function

```
foldT : ('a -> 'b -> 'b -> 'b) -> 'a binTree -> 'b -> 'b
```

which operates on trees the same way that `foldr` operates on lists. In other words, the base item of type `'b` should replace the leaf constructor in the tree, and the function of type `'a -> 'b -> 'b -> 'b` should replace the node constructor in the tree.[3]

Problem 5. In each of the three parts in this problem, you will get full credit if you use `foldT` and define at most one helper function.

(a) Define an OCaml function `leafCount : 'a binTree -> int` which returns an integer representing the total number of leaves in a tree.

(b) Define an OCaml function `nodeCount : 'a binTree -> int` which returns an integer representing the total number of non-leaf nodes in a tree.

(c) Define a function `height : 'a binTree -> int` which returns an integer representing the height of the tree. Trees consisting of only a leaf have height `0`.

Problem 6.

(a) A tree is *perfect* if all the leaves of the tree are at the same depth. Define an OCaml function `perfect : 'a binTree -> bool` which returns `true` if the tree supplied is perfect and `false` otherwise. You may use any approach to implement this function.

(b) A tree is a *degenerate* if all the nodes are arranged in a single path. Equivalently, a tree is degenerate if all nodes have at least one leaf child. Define an OCaml function `degenerate : 'a binTree -> bool` which returns `true` if and only if the tree supplied is degenerate.[4]

(c) Define a function `treeToList (t : 'a binTree) : ('a list) option`. If the supplied tree is degenerate, the function should return `Some l`, where `l` corresponds to a list constructed by replacing the `Node` constructors with `(::)` constructors and replacing the `Leaf` constructors with the `[]` constructor where appropriate. If the supplied tree is not degenerate, the function should return `None`. You are encouraged to use `degenerate` and `foldT` to implement this function.[5]

---

[3] *Hint*: It is useful to recall how the function `foldr_right` operates on lists. If you ask for the type of `foldr_right` at the top level, you get the following:

```
# List.fold_right ;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Try to understand how `foldr_right` works before attempting to program `foldT`. AAA

[4] *Hint*: You do not need to make your function recursive if you use functions you have already defined. How many leaves does a degenerate tree have? How many nodes?

[5] *Hint*: Do not make your use of `foldT` more complicated than necessary. Do you need to check that the tree is degenerate more than once? Use your `nodeCount` implementation as a guide.