Written exam, Functional Programming

Monday Aug 17, 2020

- The exam duration is five hours
- There are four questions. To obtain full marks you must answer all the subquestions satisfactorily
- You are allowed to use books, lecture notes, lecture slides, hand-ins, solutions to assignments, calculators, computers, software, on-line resources etc. during the examination. This includes any form of device that can execute programs written in F#.
- You may **NOT** copy code found online that you yourself have not written and hand that in as your solution. To be safe stick to resources such as *F# for fun and profit* or Microsoft's documentation of .NET and the F# language.
- You are (unless otherwise instructed) allowed to use the .NET library including the modules described in the book, e.g., List. Set, Map etc.
- If a subquestion requires you to define a particular function, then you may (unless otherwise instructed) use that function in subsequent subquestions, even if you have not managed to define it. Providing the signature of the missing function will help in such cases.
- If a subquestion requires you to define a particular function, then you may (unless otherwise instructed) define as many helper functions as you want, but in any case you must define the required function so that it has exactly the type and effect that the subquestion asked for.
- Unless explicitly stated you are required to provide functional solutions, and solutions with side effects will not be considered. The one exception to this rule concerns parallelism as `Async.Parallel` returns the results of the individual processes in an array and these results may be used.
- You are required to use the provided code project `FPExam2020_2` as a basis for your submission and you should only hand in the `Exam.fs` file (no other file). The project includes everything you need to run as an independent project, but you may also use the F# top loop. See the `README` for details. Any helper functions that we provide in `Exam.fs` file may also be part of your submission.
- Most functions that you need to write are present in the code skeleton. If an assignment asks that you write a function `isEven : int -> bool`, for instance, then there is nearly always a corresponding `let isEven _ = failwith "Not implemented"` in the source file. You may change these functions (changing a `let` to a `let rec` for instance) as long as their signatures correspond to those given in the assignment. In this case that could be `let isEven x = x % 2 = 0`. Be wary of polymorphic variables as notation sometimes differs and some IDEs, for instance, will write `MyType<'a when 'a : equality>` while others may write `MyType<'a> when 'a : equality`. These are identical.
- After the exam is done we will be doing a random check of around 20% of the students. You will get to know promptly at the end of the exam if you have been chosen for this. If so, you must be present in the provided Zoom room 30 minutes after the exam, and up until 2 hours after the exam, or until told by a teacher that you are allowed to leave.

You MUST include explanations and comments to support your solutions for the questions that require them. You simply write them as comments around your code.

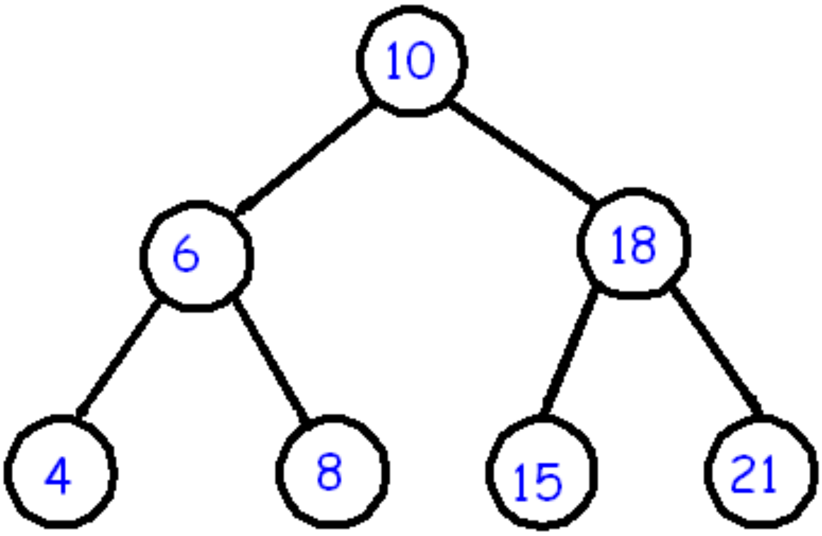
Your exam hand-in MUST be made by yourself and yourself only, and this holds for program code, examples, the explanation you provide for the code, and all other parts of the answers. It is illegal to make the exam answers as group work or to enlist the help of others in any way. This includes using solutions or code found online.

Your solution MUST compile. We reserve the right to fail any submission that does not meet this requirement.

1: Binary search trees (25%)

Binary search trees are trees where each node in the tree has two children, and one value which is greater than or equal to all values in the left sub tree and smaller than all values in the right sub tree.

A graphical representation can be seen here.



source: https://www.cs.cmu.edu/~adamchik/15-121/lectures/Trees/trees.html

Also, recall the inorder traversal and reverse inorder traversal of the tree.

- Inorder traversal traverses the nodes in order **4 6 8 10 15 18 21** by recursively
  - traversing the left sub tree
  - traversing the node
  - traversing the right sub tree
- Reverse inorder traversal traverses the nodes in order **21 18 15 10 8 6 4** by recursively
  - traversing the right sub tree
  - traversing the node
  - traversing the left sub tree

Binary search trees can be modelled in F# as follows.

```
type 'a bintree =
| Leaf
| Node of 'a bintree * 'a * 'a bintree
```

Question 1.1

Create a function `insert : 'a -> 'a bintree -> 'a bintree when 'a : comparisson` that given an element `x` and a binary search tree `t` inserts `x` into `t` such that the binary search tree property is maintained.

The standard way of doing this is that when inserting a number `x` into a tree `t`

- If `t` is a leaf node, create a new node with the value `x` and leaves as sub-trees.
- If `t` is a node with the value `y` and left sub-tree `tl` and right sub-tree `tr`
  - insert `x` into `tl` if `x` is smaller than or equal to `y`
  - insert `x` into `tr` if `x` is greater than `y`

Your function should not be tail recursive - creating a tail-recursive variant is difficult so don't spend time on it.

Examples:

```
> let t1 = insert 5 Leaf;;
- val t1 : int bintree = Node (Leaf,5,Leaf)

> let t2 = insert 3 t1;;
- val t2 : int bintree = Node (Node (Leaf,3,Leaf),5,Leaf)

> let t3 = insert 4 t2;;
- val t3 : int bintree = Node (Node (Leaf,3,Node (Leaf,4,Leaf)),5,Leaf)

> let t4 = insert 10 t3;;
- val t4 : int bintree =
    Node (Node (Leaf,3,Node (Leaf,4,Leaf)),5,Node (Leaf,10,Leaf))
```

Question 1.2

Create a tail-recursive function `fromList : 'a list -> 'a bintree when 'a : comparisson'`, using an accumulator (not a continuation) that given a list `lst` constructs a binary search tree containing all elements from lst.

Hint: use the `insert` function from Q1.1.

Examples:

```
fromList [5;3;4;10];;
 val it : int bintree =
    Node (Node (Leaf,3,Node (Leaf,4,Leaf)),5,Node (Leaf,10,Leaf))
```

Question 1.3

Create two functions `fold : ('a -> 'b -> 'a) -> 'a -> 'b bintree -> 'a` and `foldBack : ('a -> 'b -> 'a) -> 'a -> 'b bintree -> 'a` that given a function `f`, an accumulator `acc` and a binary search tree `t` folds over the tree in a similar way to the corresponding fold functions from the list library, but where `fold` operates using an inorder traversal of the tree and `foldBack` operates by using a reverse inorder traversal of the tree (as described in the start of this section).

You must fold directly over the tree and may not, for instance, translate the tree to a list and then appeal to `List.fold`.

For example,

- the inorder traversal of the tree at the top of this section is **4 6 8 10 15 18 21** and your `fold` function applied to that tree, a folding function `f`, and a starting accumulator `acc`, should produce the same result as `f (f (f (f (f (f (f acc 4) 6) 8) 10) 15) 18) 21`
- the *reverse* inorder traversal of the tree at the top of this section is **21 18 15 10 8 6 4** and your `foldBack` function applied to that tree, a folding function `f`, and a starting accumulator `acc`, should produce the same result as `f (f (f (f (f (f (f acc 21) 18) 15) 10) 8) 6) 4`

Note: In the List library the functions `fold` and `foldBack` have different types - they don't here and this makes your life easier. Nevertheless, be careful of cut-and-paste errors when you create these two functions.

```
> fold (fun acc x -> x - acc) 0 (fromList [3;5;4;10]);;
- val it : int = 6

> foldBack (fun acc x -> x - acc) 0 (fromList [3;5;4;10]);;
- val it : int = -6
```

Using one of your fold function, create a function `inOrder : 'a bintree -> 'a list` that given a binary tree `t` produces a list corresponding to the inorder traversal of `t`.

Important: Your function must have linear complexity (do not append singleton elements to the end of lists).

```
> inOrder (fromList [5;3;4;10]);;
- val it : int list = [3; 4; 5; 10]
```

Question 1.4

Consider the following map function

```
let rec badMap f =
  function
  | Leaf -> Leaf
  | Node (l, y, r) -> Node (badMap f l, f y, badMap f r)
```

Even though the type of this function is `('a -> 'b) -> 'a bintree -> 'b bintree` as we would expect from a map function, and that it does in fact apply a mapping function to all nodes in the tree, this function does not do what we want it to do. What is the problem? Provide an example that demonstrates the problem.

Create a function `map : ('a -> 'b) -> 'a bintree -> 'b bintree` that given a function `f` and a binary search tree `t` maps `f` to all elements of `t` but that does not have the problem that `badMap` has.

Hint: Use one of the `fold` or `foldBack` functions from Q1.3

2: Code Comprehension (25%)

Consider and run the following three functions

```
let rec foo =
    function
    | [x] -> [x]
    | x::y::xs when x > y -> y :: (foo (x::xs))
    | x::xs               -> x :: foo xs

let rec bar =
    function
    | [x]          -> true
    | x :: y :: xs -> x <= y && bar (y :: xs)

let rec baz =
    function
    | []               -> []
    | lst when bar lst -> lst
    | lst              -> baz (foo lst)
```

Question 2.1

- What are the types of functions `foo`, `bar`, and `baz`?
- What do functions `bar`, and `baz` do (not `foo`, we admit that it is a bit contrived)? Focus on what they do rather than how they do it.
- What would be appropriate names for functions `foo`, `bar`, and `baz`?

Question 2.2

The functions `foo` and `bar` generate a warning during compilation: `Warning: Incomplete pattern matches on this expression.`.

- Why does this happen, and where?
- For these particular three functions will these incomplete pattern matches ever cause problems for any possible execution of `baz`? If yes, why; if no, why not.

Using the following function `baz2` in stead of `baz`

```
let rec baz2 =
  function
  | lst when bar2 lst -> lst
  | lst               -> baz2 (foo2 lst)
```

Write functions `foo2` and `bar2` such that no function generates these warnings and that `baz` and `baz2` behave the same for all possible inputs.

Hint: You cannot use `failwith` to solve this.

Question 2.3

Consider this alternative definition of `foo`.

```
let rec foo3 =
  function
  | [x]                 -> [x]
  | x::xs               -> x :: foo3 xs
  | x::y::xs when x > y -> y :: (foo3 (x::xs))
```

Do the functions `foo` and `foo3` produce the same output for all possible inputs? If yes, why; if no why not and provide a counter example.

Question 2.4

Using higher-order function(s) from the list library, create a function `bar3` that behaves the same as `bar2` (it must generate no compilation warnings and never fail).

Hint: An accumulator may store information that is needed for the computation, but which you can discard at the end.

Question 2.5

One of the functions `foo` and `baz` is not tail-recursive. Why? To make a compelling argument you should evaluate a function call of the function, similarly to what is done in Chapter 1.4 of HR, and reason about that evaluation. You need to make clear what aspects of the evaluation tell you that the funciton is not tail recursive. Keep in mind that all steps in an evaluation chain must evaluate to the same value (`(5 + 4) * 3 --> 9 * 3 --> 27`, for instance).

Create a tail-recursive version of `foo` or `baz`, whichever is not tail recursive, called `fooTail` or `bazTail` respectively, using continuations (not an accumulator), that does exactly the same thing as the original function except that it does not generate any warnings and never fails.

3: Big integers

Integers are typically stored by the computer in 32- or 64-bit registers and have a natural upper bound to how high they can go. In this assignment we will create a small library for big positive integers with support for addition, multiplication, and the factorial function.

For this assignment you may use any library from the standard library **except** any library that handles big numbers of any kind like the `BigInt` library. Using this library provides no credit.

Question 3.1

Create a type `bigInt` to store big integers. There must be no upper bound (short of the program running out of memory) to how big the numbers can be.

Important: The rest of this assignment hinges on chosing a good representation for big integers so read through Q3.1-3.3 to make sure you know what is required. If you get stuck, move on to Q4 which requires less code and all types are provided to you.

Create a function `fromString : string -> bigInt` that given a string `nums` consisting only of numbers between `0` and `9` creates the corresponding `bigInt`. You may assume that:

- there are no leading zeroes in `nums` (`"000"`, `"010"`, etc)
- `nums` is not equal to the empty string `""`

Your function does not have to handle these cases at all and we will not test for them.

Create a function `toString : bigInt -> string` that given a big integer `x` returns a string corresponding to that number

Hint: A useful way to turn a charecter `c`, if you know it is a number between `0` and `9`, to its corresponding integers is `int c - int '0'`.

.**Examples:**

```
> "0" |> fromString |> toString;;
- val it : string = "0"

> "120" |> fromString |> toString;;
- val it : string = "120"

> "12345689123456789" |> fromString |> toString;;
- val it : string = "12345689123456789"
```

Question 3.2

Create a function `add : bigInt -> bigInt -> bigInt` that given two big integers `x` and `y` returns the big integer `x + y`.

Use standard long addition as you would on paper. If you need to freshen up on the algorithm then you can do so here: https://www.mathsisfun.com/numbers/addition-column.html

Examples:

```
> add (fromString "15") (fromString "39") |> toString;;
> val it : string = "54"

> add (fromString "9995") (fromString "8") |> toString;;
- val it : string = "10003"

> add (fromString "123456789123456789")
      (fromString "987654321987654321") |> toString;;
- val it : string = "1111111111111111110"
```

Question 3.3

Create a function `multSingle : bigInt -> int -> bigInt` that given a big integer `x` and a standard integer `y` between `0` and `9` (your function does not have to work otherwise and we will not test against any other numbers) returns the big integer `x * y`.

Use standard long multiplication as you would on paper, but bearing in mind that you are only multiplying against a single number between `0` and `9`. If you need to freshen up on the algorithm you can do so here: https://www.mathsisfun.com/numbers/multiplication-long.html

Examples:

```
> multSingle (fromString "4") 8 |> toString;;
- val it : string = "32"

> multSingle (fromString "424") 0 |> toString;;
- val it : string = "0"

> multSingle (fromString "123456789123456789") 9 |> toString;;
- val it : string = "1111111102111111101"
```

Question 3.4

Create a function `mult : bigInt -> bigInt -> bigInt` that given two big integers `x` and `y` returns the big integer `x * y`.

Use standard long multiplication as you would on paper, but this time you are multiplying with larger numbers.

Hint: Use `add` and `multSingle` from Q3.2 and Q3.3 respectively.

Examples:

```
> mult (fromString "12") (fromString "2") |> toString;;
- val it : string = "24"

> mult (fromString "45") (fromString "956") |> toString;;
- val it : string = "43020"

> mult (fromString "123456789123456789")
       (fromString "987654321987654321") |> toString;;
- val it : string = "121932631356500531347203169112635269"
```

Question 3.5

Create a factorial function `fact : int -> int -> bigInt` that given two standard integers `x` and `numThreads` returns `!x` by calculating the result in `numThreads` threads in parallel.

Recall that the definition of `!x` is `x * (x - 1) * (x - 2) * ... * 1` and that `!0` is `1`. So, for instance, if you were to calculate `!10` in two threads you would:

- Have one thread multiply `1 * 2 * 3 * 4 * 5 = 120`
- Have one thread multiply `6 * 7 * 8 * 9 * 10 = 30240`
- Collect the results from the two threads and multiply their results `30240*120 = 3628800`

On top of this you may assume that:

- `numThreads > 0` and that `x % numThreads = 0` - the number you are calculating will always create an equal amount of work for all threads.
- `x >= 0`

Hint: It is easier if you treat `!0` as a special case as no computation is required.

Examples:

```
> fact 0 1 |> toString;;
- val it : string = "1"

> fact 10 1 |> toString;;
- val it : string = "3628800"

> fact 10 2 |> toString;;
- val it : string = "3628800"

> fact 10 5 |> toString;;
- val it : string = "3628800"

> fact 10 10 |> toString;;
- val it : string = "3628800"

> fact 20 10 |> toString;;
- val it : string = "2432902008176640000"
```

4: Lazy lists

Lazy lists delay the computation of the individual elements of the list until they are actually needed. We will be working with infinite lists only which means that computing the entire list ahead of time will never be possible as that computation would run forever.

More precisely, we define lazy lists in the following manner

```
type 'a llist =
| Cons of (unit -> ('a * 'a llist))
```

The type `llist` only has one single constructor containing a unit function that produces the head and the tail of the lazy list.

For instance, the following example produces an infinite lazy list consisting only of zeroes.

```
let rec llzero = Cons (fun () -> (0, llzero))
```

The unit guard ensures that the next element of the list is not evaluated when the list is created, but rather when the list is accessed by calling the function for individual elements of the list.

Important: You do **NOT** need mutable state for this assignment.

Assignment 4.1

Create a function `step : 'a llist -> ('a * 'a llist)` that given a lazy list `ll` returns a pair containing the head and the tail of the list.

Examples:

```
> let (hd, tl) = step llzero;;
- val tl : int llist = Cons <fun:llzero@0>
  val hd : int = 0

> let (hd1, tl1) = step tl;;
- val tl1 : int llist = Cons <fun:llzero@0>
  val hd1 : int = 0

> let (hd2, tl2) = step tl1;;
- val tl2 : int llist = Cons <fun:llzero@0>
  val hd2 : int = 0
```

Create a function `cons : 'a -> a llist -> 'a llist` that given an element `x` and a lazy list `ll` returns `ll` with `x` added to the head of the list.

Examples:

```
> let (hd, tl) = step (cons 42 llzero);;
- val tl : int llist = Cons <fun:llzero@0>
  val hd : int = 42

> let (hd1, tl1) = step tl;;
- val tl1 : int llist = Cons <fun:llzero@0>
  val hd1 : int = 0
```

Assignment 4.2

Create a function `init : (int -> 'a) -> 'a llist` that given a function `f` of type `int -> 'a` returnes a lazy list of the form `[f 0; f 1; f 2; ...]`. Constructing the list should be done in constant time, while accessing the next element, e.g. using `step`, depends on the complexity of `f`.

Examples:

```
> let (hd, tl) = step (init (fun x -> x % 3));;
- val tl : int llist = Cons <fun:aux@0-1>
  val hd : int = 0

> let (hd1, tl1) = step tl;;
- val tl1 : int llist = Cons <fun:aux@0-1>
  val hd1 : int = 1

> let (hd2, tl2) = step tl1;;
- val tl2 : int llist = Cons <fun:aux@0-1>
  val hd2 : int = 2

> let (hd3, tl3) = step tl2;;
- val tl3 : int llist = Cons <fun:aux@0-1>
  val hd3 : int = 0
```

Assignment 4.3

Create a function `map : ('a -> 'b) -> 'a llist -> 'b llist` that given a function `f` and a lazy list `[e0; e1; e2; ...]` returns the lazy list `[f e0; f e1; f e2; ...]`. Your function should run in constant time, but accessing the elements will depend on the complexity of `f`.

Examples:

```
> let (hd, tl) = init id |> map (fun x -> x % 2 = 0) |> step;;
- val tl : bool llist = Cons <fun:map@0>
  val hd : bool = true

> let (hd1, tl1) = step tl;;
- val tl1 : bool llist = Cons <fun:map@0>
  val hd1 : bool = false

> let (hd2, tl2) = step tl1;;
- val tl2 : bool llist = Cons <fun:map@0>
  val hd2 : bool = true
```

Assignment 4.4

Create a function `filter : ('a -> bool) -> 'a llist -> 'a llist` that (similarly to the filter function for regular lists) given a function `f` and a lazy list `ll` returns a lazy list that contains all elements of `e` of `ll` where `f e = true` and where the order of the elements is preserved.

Important: This function should run in constant time, but when using the `step` function to get the next element of a filtered list you can end up in an infinite loop if no element matching the function `f` is present in the list. For instance, the command `step (filter (fun _ -> false) ll)` will not terminate for any lazy list `ll`.

```
> let (hd, tl) = init id |> filter (fun x -> x % 2 = 0) |> step;;
- val tl : int llist = Cons <fun:filter@0>
  val hd : int = 0

> let (hd1, tl1) = step tl;;
- val tl1 : int llist = Cons <fun:filter@0>
  val hd1 : int = 2

> let (hd2, tl2) = step tl1;;
- val tl2 : int llist = Cons <fun:filter@0>
  val hd2 : int = 4
```

Assignment 4.5

Create a function `takeFirst : int -> 'a llist -> ('a list * 'a llist)` that given a number `x` and a lazy list `ll` returns a pair containing a standard list containing the first `x` elements of `ll` and the lazy list containing the rest of the elments from `ll`.

Important: Your function must be linear -- you must make sure to only call the function for every element you are interested in at most once as these functions can potentially be very expensive. You must also make sure that your function does not overflow the stack.

Examples:

```
> let (hdlst, tl) = init id |> takeFirst 10;;
- val tl : int llist = Cons <fun:aux@0-1>
  val hdlst : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]

> let (hd1, tl1) = step tl;;
  val tl1 : int llist = Cons <fun:aux@0-1>
  val hd1 : int = 10
```

Assignment 4.6

Create a function `unfold : ('state -> ('a * 'state)) -> 'state -> 'a llist` that, simlirly to `Seq.unfold`, takes a function `generator`, which given a state returns an element of the lazy list and the next state, and an initial state `st`, returns the lazy list created by sequentially applying `generator` in the following way:

- `generator st = (x, st')`
- `generator st' = (y, st'')`
- `generator st'' = (z, st''')`
- ...

and then returns the lazy list `[x; y; z; ...]`.

Examples:

```
> let (hd, tl) = step (unfold (fun st -> (st, st + 5)) 0);;
- val tl : int llist = Cons <fun:unfold@0>
  val hd : int = 0

> let (hd1, tl1) = step tl;;
- val tl1 : int llist = Cons <fun:unfold@0>
  val hd1 : int = 5

> let (hd2, tl2) = step tl1;;
- val tl2 : int llist = Cons <fun:unfold@0>
  val hd2 : int = 10
```

Recall that the Fibonacci sequence **0 1 1 2 3 5 8 13 21 ...** where each element in the sequence is the sum of the previous two.

Consider the following two implementations of Fibonacci sequences `fibll1` and `fibll2`:

```
let fib x =
    let rec aux acc1 acc2 =
        function
        | 0 -> acc1
        | x -> aux acc2 (acc1 + acc2) (x - 1)

    aux 0 1 x


let fibll1 = init fib
let fibll2 = unfold (fun (acc1, acc2) -> (acc1, (acc2, acc1 + acc2))) (0, 1)
```

Both `fibll1` and `fibll2` correctly calculate a lazy list of Fibonacci numbers. Which of these two lazy lists is the most efficient implementation and why?