CSC 402/502 Homework 5 Instructor: Jeff Ward
Covers: F#: Finite trees (Chapter 6, Sections 2-4,7) 40 pts

For this assignment create a file called HW5.fs. All of your code for this assignment should be placed in this .fs file.

If you are using JetBrains Rider as your IDE then you can create the file and runnable project as you did for HW4: Click File->New... . Select “Console Application” under .NET Core. Give the solution a name (e.g. “HW5” without the quotes). The dialog box will fill in the same name (e.g. “HW5”) as the name of the project. Click “Create”. Rider will create a file named Program.fs under the HW5 project.

Once again you will submit your work to Mimir. To prepare for this, perform the following three additional steps:

(1) Right-click on Program.fs file , select Edit->Rename..., and change the file’s name to HW5.fs.

(2) Put the declaration **`module HW5`** at the top of HW5.fs. Also **`open System`** there, as well.

(3) At the bottom of HW5.fs file delete the `[<EntryPoint>]` directive and all of the code that constitutes the main function.

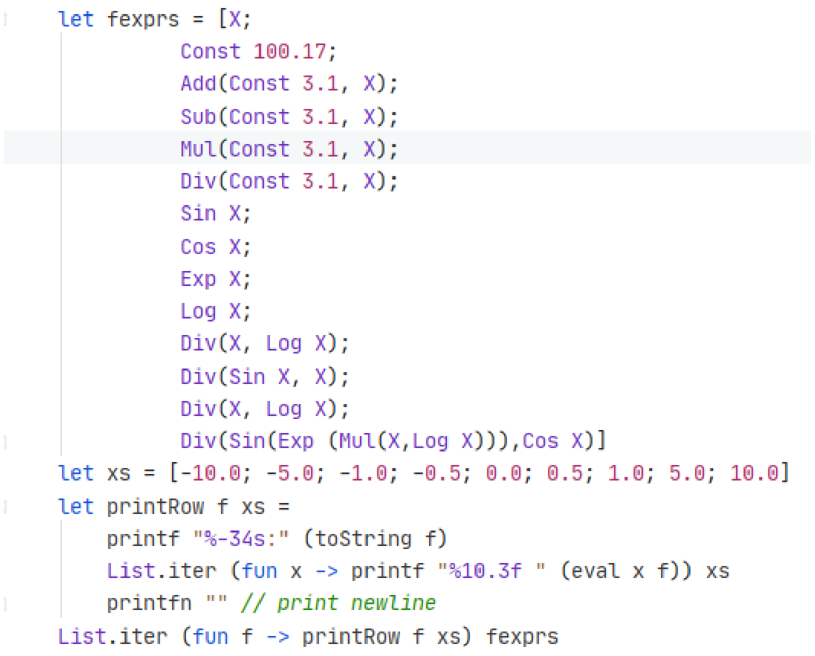
From Canvas you can download HW5Test.fs and include it in your project in order to test your code. (Make sure that HW5Test.fs appears *after/below* HW5.fs in your project explorer.)

Problem 1 (Evaluating expressions – 8 pts) Consider the following type and function definitions from Section 6.2:

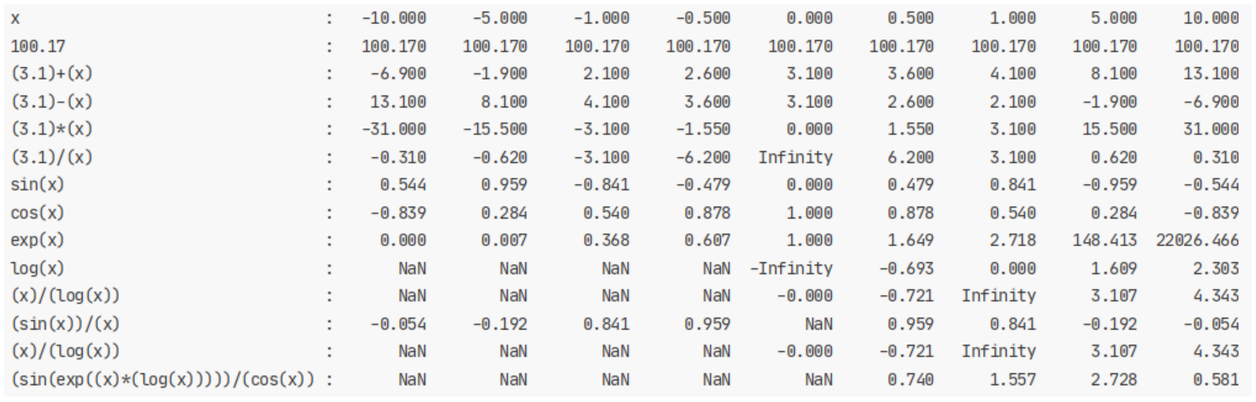
```
type Fexpr =
    | Const of float
    | X
    | Add of Fexpr * Fexpr
    | Sub of Fexpr * Fexpr
    | Mul of Fexpr * Fexpr
    | Div of Fexpr * Fexpr
    | Sin of Fexpr
    | Cos of Fexpr
    | Log of Fexpr
    | Exp of Fexpr

let rec toString = function
    | Const x      -> string x
    | X            -> "x"
    | Add(fe1,fe2) -> "(" + (toString fe1) + ")" + "+" + "(" + (toString fe2) + ")"
    | Sub(fe1,fe2) -> "(" + (toString fe1) + ")" + "-" + "(" + (toString fe2) + ")"
    | Mul(fe1,fe2) -> "(" + (toString fe1) + ")" + "*" + "(" + (toString fe2) + ")"
    | Div(fe1,fe2) -> "(" + (toString fe1) + ")" + "/" + "(" + (toString fe2) + ")"
    | Sin fe       -> "sin(" + (toString fe) + ")"
    | Cos fe       -> "cos(" + (toString fe) + ")"
    | Log fe       -> "log(" + (toString fe) + ")"
    | Exp fe       -> "exp(" + (toString fe) + ")"
```

Define a function `eval` of type `float -> Fexpr -> float`. The task of `eval` is to evaluate an Fexpr at a particular value for x. (Hint: This function very similar to `toString`, except you have an additional parameter of type `float`, and the function returns a `float`.)

```
> eval 3.14159 (Sin (Div(X, Const 2.0)));;
val it : float = 1.0
```



```
let fexprs = [X;
              Const 100.17;
              Add(Const 3.1, X);
              Sub(Const 3.1, X);
              Mul(Const 3.1, X);
              Div(Const 3.1, X);
              Sin X;
              Cos X;
              Exp X;
              Log X;
              Div(X, Log X);
              Div(Sin X, X);
              Div(X, Log X);
              Div(Sin(Exp (Mul(X,Log X))),Cos X)]
let xs = [-10.0; -5.0; -1.0; -0.5; 0.0; 0.5; 1.0; 5.0; 10.0]
let printRow f xs =
    printf "%-34s:" (toString f)
    List.iter (fun x -> printf "%10.3f " (eval x f)) xs
    printfn "" // print newline
List.iter (fun f -> printRow f xs) fexprs
```

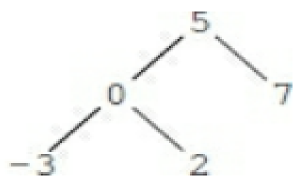


```
x                                 :    -10.000     -5.000     -1.000     -0.500      0.000      0.500      1.000      5.000     10.000
100.17                            :    100.170    100.170    100.170    100.170    100.170    100.170    100.170    100.170    100.170
(3.1)+(x)                         :     -6.900     -1.900      2.100      2.600      3.100      3.600      4.100      8.100     13.100
(3.1)-(x)                         :     13.100      8.100      4.100      3.600      3.100      2.600      2.100     -1.900     -6.900
(3.1)*(x)                         :    -31.000    -15.500     -3.100     -1.550      0.000      1.550      3.100     15.500     31.000
(3.1)/(x)                         :     -0.310     -0.620     -3.100     -6.200   Infinity      6.200      3.100      0.620      0.310
sin(x)                            :      0.544      0.959     -0.841     -0.479      0.000      0.479      0.841     -0.959     -0.544
cos(x)                            :     -0.839      0.284      0.540      0.878      1.000      0.878      0.540      0.284     -0.839
exp(x)                            :      0.000      0.007      0.368      0.607      1.000      1.649      2.718    148.413  22026.466
log(x)                            :        NaN        NaN        NaN        NaN  -Infinity     -0.693      0.000      1.609      2.303
(x)/(log(x))                      :        NaN        NaN        NaN        NaN     -0.000     -0.721   Infinity      3.107      4.343
(sin(x))/(x)                      :     -0.054     -0.192      0.841      0.959        NaN      0.959      0.841     -0.192     -0.054
(x)/(log(x))                      :        NaN        NaN        NaN        NaN     -0.000     -0.721   Infinity      3.107      4.343
(sin(exp((x)*(log(x)))))/(cos(x)) :        NaN        NaN        NaN        NaN        NaN      0.740      1.557      2.728      0.581
```

Problem 2 (Tree traversal – 8 pts) Consider the following type and function definitions from Section 6.4:

```
type BinTree<'a> =
    | Leaf
    | Node of BinTree<'a> * 'a * BinTree<'a>

let rec preOrder = function
    | Leaf    -> []
    | Node(tl,x,tr) -> x::(preOrder tl) @ (preOrder tr)

let rec inOrder = function
    | Leaf          -> []
    | Node(tl,x,tr) -> (inOrder tl) @ [x] @ (inOrder tr)

let rec postOrder = function
    | Leaf          -> []
    | Node(tl,x,tr) -> (postOrder tl) @ (postOrder tr) @ [x]
```

Define a function `levelOrder` of type `BinTree<'a> -> 'a list`. The task of `levelOrder` is to return a list of the tree elements according to a level-order traversal. This involves traversing the tree level-by-level, starting at the root, and proceeding left-to-right along each level. For example, recall the following tree *t4* from Figure 6.11 in the textbook:



```
t4;;
val it : BinTree<int> =
  Node
    (Node (Node (Leaf, -3, Leaf), 0, Node (Leaf, 2, Leaf)), 5,
     Node (Leaf, 7, Leaf))

levelOrder t4;;
val it : int list = [5; 0; 7; -3; 2]
```

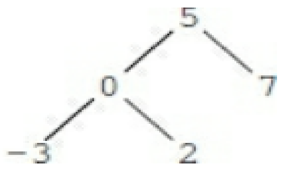
Tip - The following pseudocode from Wikipedia (https://en.wikipedia.org/wiki/Tree_traversal) gives an algorithm for level-order traversal:

```
levelorder(root)
  q := empty queue
  q.enqueue(root)
  while not q.empty do
    node := q.dequeue()
    visit(node)
    if node.left ≠ null then
      q.enqueue(node.left)
    if node.right ≠ null then
      q.enqueue(node.right)
```

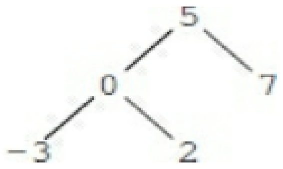
Note – The above pseudocode uses a couple of assignment statements. However, we have not covered assignment statements in F#. They exist in F# but are not covered until Chapter 8 of the textbook. Also, assignment statements involve changing the value of a variable. This is not considered part of *pure* functional programming. We are trying to learn to program in a functional style here and, in fact, any computable function can be defined in a functional style. So do this problem without assignment statements. We can translate the idea behind the above pseudocode into concise, purely functional code if we use recursion rather than a loop.

Suggestion – Use a recursive helper function to define `levelOrder`. I'll call the helper `levelOrderHelper`. The type of the helper can be `BinTree<'a> list -> 'a list`, in which case it takes a list of BinTrees and returns the elements of the BinTrees listed in level order. So a full level order computation on t4 might proceed roughly like this:

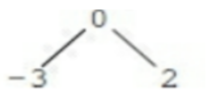
```
levelOrder
```



```
= levelOrderHelper [                                ]
```



```
= 5 :: levelOrderHelper [                ; 7]
```



```
= 5 :: 0 :: levelOrderHelper[7 ; -3 ; 2]

= 5 :: 0 :: 7 :: levelOrderHelper[-3 ; 2]

= 5 :: 0 :: 7 :: -3 :: levelOrderHelper[2]

= 5 :: 0 :: 7 :: -3 :: 2 :: levelOrderHelper[]

= 5 :: 0 :: 7 :: -3 :: 2 :: []

= [5;0;7;-3;2]
```

It's okay to put Leaf values into your list/queue as you go into recursion as long as you do not put them into your result when you take them off the queue. If you proceed in the way described above then you will need to consider three cases in your helper function: (1) when the parameter is an empty list, (2) when the parameter is a non-empty list with a Leaf value at the front, and (3) when the parameter is a non-empty list with a Node value at the front. You can use the :: and @ operators to implement the right-hand side of your patterns. (Review the textbook's code for the preOrder, inOrder, and postOrder traversals, which I copied and pasted into the beginning of this problem description.)

Problem 3 (Creating a SizedBinTree – 8 points) Consider the following type and function definitions:

```
type SizedBinTree<'a> =
    | SizedLeaf
    | SizedNode of SizedBinTree<'a> * 'a * SizedBinTree<'a> * int

let size = function
    | SizedLeaf -> 0
    | SizedNode(tl,x,tr,sz) -> sz
```

A SizedBinTree is supposed to be the same as a BinTree except that each node in the tree has an integer that gives the number of elements in that node's subtree.

Write function `binTree2SizedBinTree` of type `BinTree<'a> -> SizedBinTree<'a>` that takes a `BinTree` and returns the corresponding `SizedBinTree`. The following evaluation uses the example `BinTree` t4 shown in Problem 2.

```
binTree2SizedBinTree t4;;
val it : SizedBinTree<int> =
  SizedNode
    (SizedNode
       (SizedNode (SizedLeaf, -3, SizedLeaf, 1), 0,
        SizedNode (SizedLeaf, 2, SizedLeaf, 1), 3), 5,   <== Note: this 5 is an element
     SizedNode (SizedLeaf, 7, SizedLeaf, 1), 5)          <== Note: this 5 is a size
```

Problem 4 (Searching a SizedBinTree – 8 points) The List module has a List.item function that can be used the get the i-th element from a list. Of course, the indexing starts at 0.

```
List.item;;
val it : (int -> 'a list -> 'a)

List.item 3 ['a';'b';'c';'d';'e'];;
val it : char = 'd'
```

If the index is out of range the function throws an exception.

```
List.item 5 ['a';'b';'c';'d';'e'];;
System.ArgumentException: The index was outside the range of elements in the list.
(Parameter 'index')
   at Microsoft.FSharp.Collections.ListModule.Item[T](Int32 index, FSharpList`1 list)
in F:\workspace\_work\1\s\src\fsharp\FSharp.Core\list.fs:line 141
   at <StartupCode$FSI_0006>.$FSI_0006.main@()
Stopped due to error
```

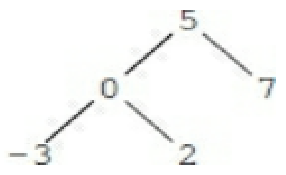
We can easily implement our own custom versions of this function. The following version returns an option:

```
let rec getOptionFromList index = function
    | [] -> None
    | x::xs -> if index = 0
               then Some x
               else getOptionFromList (index - 1) xs
val getOptionFromList : index:int -> _arg1:'a list -> 'a option

getOptionFromList 3 ['a';'b';'c';'d';'e'];;
val it : char option = Some 'd'

getOptionFromList 5 ['a';'b';'c';'d';'e'];;
val it : char option = None
```

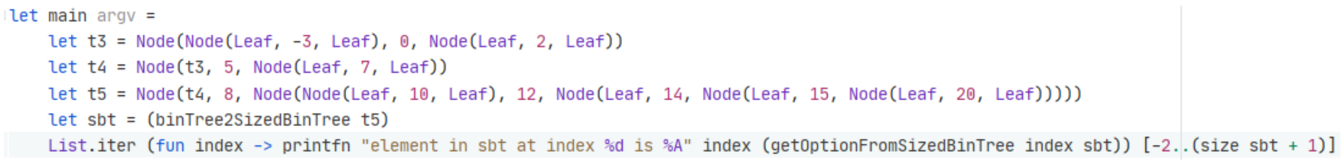
In Problem 3 you wrote a function to create `SizedBinTrees`. Now write a function `getOptionFromSizedBinTree` to return the i-th element in a `SizedBinTree`.

```
val getOptionFromSizedBinTree :
   index:int -> _arg1:SizedBinTree<'a> -> 'a option
```

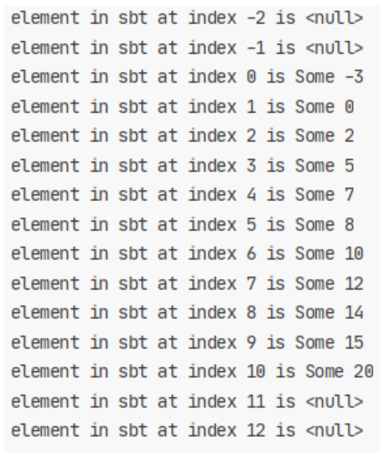
In the following examples *t4* is again the example from Figure 6.11:



```
getOptionFromSizedBinTree 1 (binTree2SizedBinTree t4);;
val it : int option = Some 0

getOptionFromSizedBinTree 5 (binTree2SizedBinTree t4);;
val it : int option = None
```

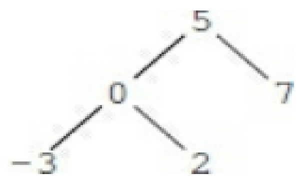


```
let main argv =
    let t3 = Node(Node(Leaf, -3, Leaf), 0, Node(Leaf, 2, Leaf))
    let t4 = Node(t3, 5, Node(Leaf, 7, Leaf))
    let t5 = Node(t4, 8, Node(Node(Leaf, 10, Leaf), 12, Node(Leaf, 14, Node(Leaf, 15, Node(Leaf, 20, Leaf)))))
    let sbt = (binTree2SizedBinTree t5)
    List.iter (fun index -> printfn "element in sbt at index %d is %A" index (getOptionFromSizedBinTree index sbt)) [-2..(size sbt + 1)]
```



```
element in sbt at index -2 is <null>
element in sbt at index -1 is <null>
element in sbt at index 0 is Some -3
element in sbt at index 1 is Some 0
element in sbt at index 2 is Some 2
element in sbt at index 3 is Some 5
element in sbt at index 4 is Some 7
element in sbt at index 5 is Some 8
element in sbt at index 6 is Some 10
element in sbt at index 7 is Some 12
element in sbt at index 8 is Some 14
element in sbt at index 9 is Some 15
element in sbt at index 10 is Some 20
element in sbt at index 11 is <null>
element in sbt at index 12 is <null>
```

Your `getOptionFromSizedBinTree` function should run in O(h) time, where h is the height of the tree. So it *should not* in general traverse the entire tree. Instead, do a binary search for the appropriate element. This is easy if you take advantage of the size property of the nodes. For example, suppose you are trying to get an element from *t4*:



The root Node has size=5 (besides the fact that it has element 5). So if the index that you are searching is negative or ≥ 5, then you know immediately that the search will fail. But suppose that the index is in the range [0,...,4]. Then look at the left child. It has a size=3. So if the index that you are searching is < 3, you know that the element that you want is in the left tree (so recurse left). If the index that you are searching is exactly 3, then you know that the root element is the one that you want. However, if the index that you are searching is > 3, then the element is in the right tree (recurse right). Note that if you recurse right in this circumstance then, in the recursion, you should reduce the index by 3+1=4 because by recursing right you are bypassing 4 elements.

Problem 5 (Sum of circuit components – 4 pts) Recall the following definitions from the *Tree Recursion* subsection of Section 6.7 *Electrical circuits*:

```
type Circuit<'a> =
     | Comp of 'a
     | Ser of Circuit<'a> * Circuit<'a>
     | Par of Circuit<'a> * Circuit<'a>

let rec circRec (c,s,p) = function
     | Comp x -> c x
     | Ser (c1,c2) ->
         s (circRec (c,s,p) c1) (circRec (c,s,p) c2)
     | Par (c1, c2) ->
         p (circRec (c,s,p) c1) (circRec (c,s,p) c2)

let count circ = circRec ((fun _ -> 1), (+), (+)) circ : int


let resistance =
    circRec (
              (fun r -> r),
              (+),
              (fun r1 r2 -> 1.0/(1.0/r1 + 1.0/r2)))
```

In the above, a `Circuit` is defined as a tree. The interesting part is `circRect`, which is a higher-order generic function that can be used to iterate through circuit to accumulate a result. The `circRec` function takes two parameters: a triple and a circuit. The triple consists of three functions: c, s, and p, which are used to accumulate the result when the recursion reaches a `Comp`, `Ser`, or `Par`, respectively. The `count` and `resistance` functions above show particular applications of `circRec`: `count` returns the number of components in a circuit and `resistance` returns the resistance of a circuit.

For example, if `cmp` is the circuit from Figure 6.16 in the textbook,

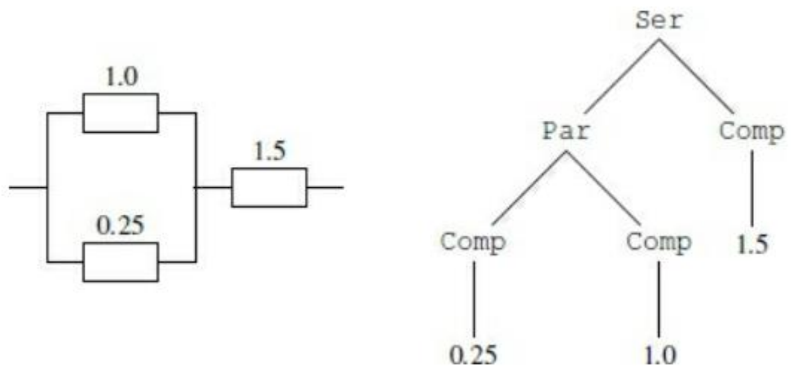


Figure 6.16 Circuit and corresponding tree

then we have:

```
let cmp = Ser(Par(Comp 0.25, Comp 1.0), Comp 1.5);;
val cmp : Circuit<float> = Ser (Par (Comp 0.25, Comp 1.0), Comp 1.5)

count cmp;;
val it : int = 3

resistance cmp;;
val it : float = 1.7
```

Use `circRec` to define a function `sum` that returns the sum of the resistances in the individual components of a circuit.

```
sum;;
val it : (Circuit<float> -> float) = <fun:it@3-11>

sum cmp;;
val it : float = 2.75
```

Problem 6 (Sum along cheapest path – 4 pts) Repeat the previous problem, but this time your function should return the sum of components along the cheapest path that starts at a component and ends at the root. Call it `sumAlongCheapestPath`.

```
sumAlongCheapestPath;;
val it : (Circuit<float> -> float) = <fun:it@3-10>

sumAlongCheapestPath cmp;;
val it : float = 1.75
```

How to submit your code for automated grading:

The submission process is the same as for HW4. Visit “Assignments”->”HW5: F# trees” and click the “Load HW5: F# trees” button at the bottom of the page. This will take you to Mimir. Upload your HW5.fs file there. Mimir will run it on the test cases from HW5Test.fs and show your score. If your program fails any test cases then you can click on those cases to see the problem. You will be able to see your program’s output was and the expected output. You can revise your program and resubmit, if necessary.

My final checks of your code:

Within a week or so of the assignment deadline I will take a look at the code that you submitted through Mimir to check that you followed the directions. For example, in Problem 2, did you use a functional style (no assignment statements)? In Problems 5 and 6, did you actually use the provided higher-order generic function (`circRec`)? Once I have checked everyone’s code I will release the grades to Canvas.