CptS355 - Assignment 2 (Standard ML)
Fall 2018

Assigned: Wednesday, September 19, 2018

Due: Monday, October 1, 2018

Weight: Assignment 2 will count for 6% of your course grade.

Your solutions to the assignment problems are to be your own work. Refer to the course academic integrity statement in the syllabus.

This assignment provides experience in ML programming. We have used both PolyML and SML of New Jersey implementations in class and you can use either for doing this assignment. You may download PolyML at http://polyml.org and SML of NewJersey at http://www.smlnj.org/ . Major Linux distributions include PolyML as an installable package (the particular version will not matter).

Turning in your assignment

All code should be developed in the file called `HW2.sml`. The problem solution will consist of a sequence of function definitions in one file. Note that this is a plain text file. You can create/edit with any text editor.

For each function, provide at least 2 test inputs (other than the given tests) and test your functions with those test inputs. Please see the section " Testing your functions" for more details. Include your test functions at the end of the file. Make sure that debugging code is removed before you submit your file (i.e. no print statements other than the test functions). Also, include your name as a comment at the top of the file.

To submit your assignment, turn in your file by uploading on the Assignment2 (ML) DROPBOX on Blackboard (under AssignmentSubmisions menu). You may turn in your assignment up to 4 times. Only the last one submitted will be graded.

The work you turn in is to be your own personal work. You may not copy another student's code or work together on writing code. You may not copy code from the web, or anything else that lets you avoid solving the problems for yourself.

Important rules

- Unless directed otherwise, you must implement your functions using recursive definitions built up from the basic built-in functions. You must program "functionally" without using ref cells (which we have not, and will not, talk about in class).
- If the problem asks for a non-recursive solution, then your function should make use of the higher order functions we covered in class (map, fold, or filter.) For those problems, your main functions can't be recursive. If needed, you may define helper functions which are recursive. However, some functions may require the helper functions to be non-recursive as well.
- The type of your functions should match with the type specified in each problem. Each problem specifies the number of arguments the functions should take and the type of each argument. You function definition should comply with that specification. Otherwise you will be deducted points (around 40%).

- Make sure that your function names match the function names specified in the assignment specification. **Also, make sure that your functions work with the given test cases.** However, the given test inputs don't cover all boundary cases. You should generate other test cases covering the extremes of the input domain, e.g. maximum, minimum, just inside/outside boundaries, typical values, and error values.
- Some questions require the solution to be tail recursive. Make sure that your function is tail recursive for those problems.
- You will call `fold, map, or filter` in several problems. Copy the definitions of `fold, map,` and `filter` functions from the lecture slides and include them in the beginning of your file.
- When auxiliary functions are needed, make them local functions (inside a `let` block). In this homework you will lose points if you don't define the helper functions inside a let block. The only exceptions to this are the `fold,map,` and `filter` functions.
- The assignment will be marked for good programming style (indentation and appropriate comments), as well as clean compilation and correct execution. ML comments are placed inside properly nested sets of opening comment delimiters, (*, and closing comment delimiters.*).

Problems

1. `numbersToSum and numbersToSumTail` **- 15%**

(a) (7%) Write a function `numbersToSum` that takes an `int` value (called `sum` - which you can assume is positive), and an `int list` (called `L` - which you can assume contains positive numbers) and returns an `int list`. The returned list should include the first `n` elements from the input list `L` such that the first `n` elements of the list add to less than `sum`, but the first `(n + 1)` elements of the list add to `sum` or more. Assume the entire list sums to more than the passed in `sum` value.
The type of `numbersToSum` should be `int -> int list -> int list.`

Examples:
```
> numbersToSum 100 [10, 20, 30, 40]
[10, 20, 30]
> numbersToSum 30 [5, 4, 6, 10, 4, 2, 1, 5]
[5, 4, 6, 10, 4]
> numbersToSum 1 [2]
[]
> numbersToSum 1 []
[]
```

(b) (8%) Re-write the `numbersToSum` function from part (a) as a tail-recursive function. Name your function `numbersToSumTail`.
The type of `numbersToSumTail` should be `int -> int list -> int list.`

2. partition - 10%

`partition` function takes a predicate function (F) and a list (L) as input, and returns a 2-tuple `(pos,neg)` as output where `pos` is the list of the elements (e_i) in L for which `F` e_i evaluated to `true`, and `neg` is the list of those elements in L for which `F` e_i evaluated to `false`. The elements of `pos` and `neg` retain the same relative order they possessed in L.
Your function shouldn't need a recursion but should use the function "`filter`". You may need to define additional helper function(s), which are not recursive.
The type of the `partition` function should be:
`('a -> bool) -> 'a list -> 'a list * 'a list`

Examples:
```
> partition (fn x => (x<=4)) [1,7,4,5,3,8,2,3]
([1,4,3,2,3],[7,5,8])

> partition null [[1,2],[1],[],[5],[],[6,7,8]]
([[],[]],[[1,2],[1],[5],[6,7,8]])

fun exists n [] = false
  | exists n (x::rest) = if n=x then true else (exists n rest)
> partition (exists 1) [[1,2],[1],[],[5],[],[6,7,8]]
([[1,2],[1]],[[],[5],[],[6,7,8]])

> partition (fn x=> (x<=4)) []
([],[])
```

3. areAllUnique - 10%

areAllUnique function takes a list as input and returns true if every element in the list appears only once (i.e. unique in the list). areAllUnique returns false otherwise. Your function shouldn't need a recursion but should use functions "map"and "filter". You may need to define additional helper functions, which can be recursive. (Hint: Use the countInList from Assignment1 as a helper function)
The type of the areAllUnique function should be ''a list ->bool.

Examples:
```
> areAllUnique [1,3,4,2,5,0,10]
true
> areAllUnique [[1,2],[3],[4,5],[]]
true
> areAllUnique [(1,"one"),(2,"two"),(1,"one")]
false
> areAllUnique []
true
> areAllUnique [1,2,3,4,1,7]
false
```

2. sum, sumOption, and sumEither

(a) **sum - 5%**
Function sum is given a list of int lists and it returns the sum of all numbers in all sublists of the input list. Your function shouldn't need a recursion but should use functions "map" and "fold". You may define additional helper functions which are not recursive.
The type of the sum function should be int list list -> int.
Examples:
```
> sum [[1,2,3],[4,5],[6,7,8,9],[]]
45
> sum [[10,10],[10,10,10],[10]]
60
> sum [[]]
0
> sum []
0
```

(b) **sumOption - 10%**
Function sumOption is given a list of int option lists and it returns the sum of all int option values in all sublists of the input list. Your function shouldn't need a recursion but should use functions "map" and "fold". You may define additional helper functions which are not recursive.
The type of the sumOption function should be:
int option list list -> int option.

(Note: To implement sumOption, change your sum function and your helper function in order to handle int option values instead of int values. Assume the integer value for NONE is 0.)

Examples:

```
> sumOption [[SOME(1),SOME(2),SOME(3)],[SOME(4),SOME(5)],[SOME(6),NONE],[],[NONE]]
SOME 21
> sumOption [[SOME(10),NONE],[SOME(10), SOME(10), SOME(10),NONE,NONE]]
SOME 40
> sumOption [[NONE]]
NONE
> sumOption []
NONE
```

(c) **sumEither - 15%**

Define the following ML datatype:

```
datatype either  = IString of string | IInt of int
```

Define an ML function `sumEither` that takes a list of `either lists` and it returns an `IInt` value which is the sum of all values in all sublists of the input list. The parameter of the `IString` values should be converted to integer and included in the sum. You may use the following function to convert a string value to integer.

```
fun str2int s = valOf(Int.fromString(s))
```

Your `sumEither` function shouldn't need a recursion but should use functions `"map"` and `"fold".` You may define additional helper functions which are not recursive. The type of the `sumEither` function should be:

```
either list list -> either
```

Examples:

```
> sumEither [[IString "1",IInt 2,IInt 3],[IString "4",IInt 5],[IInt 6,IString
"7"],[],[IString "8"]]
IInt 36
> sumEither [[IString "10" , IInt 10],[],[IString "10"],[]]
IInt 30
> sumEither [[]]
IInt 0
```

5. depthScan, depthSearch, addTrees

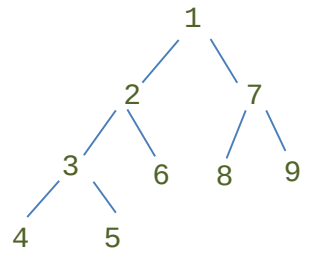
In SML, a polymorphic binary tree type with data both at the leaves and interior nodes might be represented as follows:

```
datatype 'a Tree = LEAF of 'a | NODE of 'a * ('a Tree) * ('a Tree)
```

(a) depthScan - 10%

Write a function `depthScan` that takes a tree of type `'a Tree` and returns a list of the `'a` values stored in the leaves and the nodes. The order of the elements in the output list should be based on the depth-first post-order traversal of the tree.

The type of the `depthScan` function should be: `'a Tree -> 'a list`



on depth-first post-order traversal:
`[4,5,3,6,2,8,9,7,1]`

Examples:

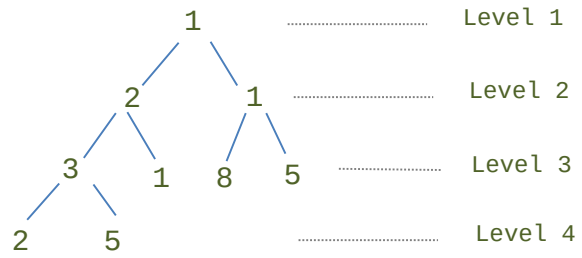
```
> depthScan (NODE("Science",NODE ("and",LEAF "School", NODE("Engineering", LEAF
"of",LEAF "Electrical")),LEAF "Computer"))
["School","of","Electrical","Engineering","and","Computer","Science"]

> depthScan (NODE(1, NODE (2, NODE(3, LEAF 4 ,LEAF 5),LEAF 6), NODE(7,LEAF 8,LEAF 9)))
[4,5,3,6,2,8,9,7,1]

> depthScan (LEAF 4)
[4]
```

(b) **`depthSearch` – 15%**

Write a function `depthSearch` takes a tree of type `'a Tree` and an `'a` value and returns the level of the tree where the value is found. If the value doesn't exist in the tree, it returns `~1`. The tree nodes should be visited with depth-first post-order traversal and the level of the first matching node should be returned. The type of the `depthSearch` function should be: `''a Tree -> ''a -> int`



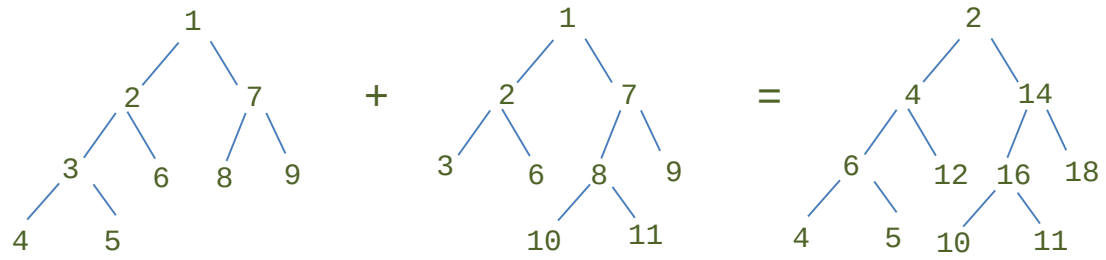

```
> val myT = NODE(1, NODE (2, NODE(3, LEAF 2 ,LEAF 5),LEAF 1), NODE(1,LEAF 8,LEAF 5))

> depthSearch myT 1
3
> depthSearch myT 5
4
> depthSearch myT 8
3
> depthSearch myT 4
~1
```

(a) **`addTrees` – 15%**

Write a function `addTrees` takes two `int Tree` values and returns an `int Tree` where the corresponding nodes from the two trees are added. The trees might have different depth. You

should copy particular branches/nodes of the trees as it is if the other tree doesn't have that branch/node. See the example below.
The type of the `addTrees` function should be `int Tree -> int Tree -> int Tree`



We can create the above `int Tree(s)` as follows:
left :

```
val T1 = NODE(1, NODE (2, NODE(3, LEAF 4 ,LEAF 5),LEAF 6), NODE(7,LEAF 8,LEAF 9))
```

right:

```
val T2 = NODE(1, NODE (2, LEAF 3, LEAF 6), NODE(7, NODE(8, LEAF 10 ,LEAF 11),LEAF 9))
```

And `addTrees T1 T2` will return the rightmost `int Tree` which is equivalent to the following:

```
NODE (2,NODE (4,NODE (6,LEAF 4,LEAF 5),LEAF 12),
     NODE (14,NODE (16,LEAF 10,LEAF 11),LEAF 18))
```

(c) **Testing your tree functions:**
Create three trees of type `'a Tree` ('a will be substituted by the type of the values stored in the tree). The height of all three trees should be at least 4. Test your functions `depthScan, depthSearch, addTrees` with those trees. The trees you define should be different than those that are given. You don't need to write additional test functions for `depthScan, depthSearch,` and `addTrees.`

Both SML of NJ doesn't display the full content of a tree value. To be able to display the full depth tree, you need to increase the print depth parameter. In the beginning of your file, include the following (you may also run this on the command line):

```
Control.Print.printDepth := 100;
```

Here is some additional test data.

```
> val L1 = LEAF "1"
> val L2 = LEAF "2"
> val L3 = LEAF "3"
> val L4 = LEAF "4"
> val N1 = NODE("5",L1,L2)
> val N2 = NODE("6",N1,L3)
> val t1 = NODE("7",N2,L4)

> depthScan t1
["1","2","5","3","6","4","7"]
```

Testing your functions

For each problem , write a test function that compares the actual output with the expected (correct) output. Below is an example test function for `partition`:

```
fun partitionTest () =
let
    fun exists n [] = false | exists n (x::rest) = if n=x then true else (exists n rest)
    val partitionT1 = ( (partition (fn x => (x<=4)) [1,7,4,5,3,8,2,3]) = ([1,4,3,2,3],[7,5,8]) )
    val partitionT2 = ( (partition null [[1,2],[1],[],[5],[],[6,7,8]]) =
([[],[]],[[1,2],[1],[5],[6,7,8]]) )
    val partitionT3 = ( (partition (exists 1) [[1,2],[1],[],[5],[],[6,7,8]]) =
([[1,2],[1]],[[],[5],[],[6,7,8]]) )
in
   print ("partition:-------------------- \n   test1: " ^ Bool.toString(partitionT1) ^  "\n" ^
            "   test2: " ^ Bool.toString(partitionT2) ^ "\n" ^
            "   test3: " ^ Bool.toString(partitionT3) ^ "\n")
end
val _ = partitionTest()
```

Make sure to test your functions for at least 2 additional test cases (in addition to the provided examples).
Note that you don't need to write test functions for `depthScan, depthSearch,` and `addTrees.`

Hints about using files containing ML code

In order to load files into the ML interactive system you have to use the function named `use`.

The function `use` has the following syntax assuming that your code is in a file in the current directory named `HW2.sml`: You would see something like this in the output:

```
> use "HW2.sml";
[opening file "HW2.sml"]
...list of functions and types defined in your file
[closing file "HW2.sml"]
> val it = () : unit
```

The effect of use is as if you had typed the content of the file into the system, so
each `val` and `fun` declaration results in a line of output giving its type.
If the file is not located in the current working directory you should specify the full or relative path-name for the file. For example in Windows for loading a file present in the users directory in the C drive you would type the following at the prompt. Note the need to use double backslashes to represent a single backslash.

```
- use "c:\\users\\example.sml";
```

Alternatively you can change your current working directory to that having your files before invoking the ML interactive system.

You can also load multiple files into the interactive system by using the following syntax

```
- use "file1"; use "file2";...; use "filen";
```

How to quit the ML environment

Control-Z followed by a newline in Windows or Control-D in Linux will quit the interactive session.

ML resources:

- Standard ML of New Jersey
- Moscow ML
- Prof Robert Harper's CMU SML course notes