15-150 Fall 2015
Homework 04

Out: Wednesday, 23 September 2015
Due: Tuesday, 29 September 2015 at 23:59 EDT

1 Introduction

This homework will focus on lists, trees, sorting, and work-span analysis.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository, as usual.

1.2 Submitting The Homework Assignment

Submissions will be handled through Autolab, at `https://autolab.cs.cmu.edu`

In preparation for submission, your `hw/04` directory should contain a file named exactly `hw04.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/04` directory (that contains a `code` folder and a file `hw04.pdf`). This should produce a file `hw04.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw04.tar` file via the "Handin your work" link.

The Autolab handin script does some basic checks on your submission: making sure that the file names are correct; making sure that no files are missing; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number corresponding to the "check" section of your latest handin on the "Handin History" page. **If this number is 0.0, your submission failed the check script; if it is 1.0, it passed.**

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

All the code that you want to have graded for this assignment should be contained in `hw04.sml`, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Due Date

This assignment is due on Tuesday, 29 September 2015 at 23:59 EDT. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester.

1.4 Methodology

You must use the five step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall the five step methodology:

1. In the first line of comments, specify the type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments to be passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns when applied to an argument that satisfies the assumptions in `REQUIRES`).
4. Implement the function.
5. Provide testcases, generally in the format

```
val <return value> = <function> <argument value>.
```

For example, for the factorial function presented in lecture:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n! *)
fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)
val 1 = fact 0
val 6 = fact 3
val 720 = fact 6
```

2 Parentheses Matching

Consider the following strings:

"","(())()(())", "()()(())", ")(())", "(()()", "(()))"

Strings are tedious to manipulate, so we will represent parentheses with the following datatype, in which `Left` represents the string "(", and `Right` represents the string ")".

```
datatype paren = Left | Right
```

A string of multiple parentheses will be represented as a list. For example, "(())" would be represented as `[Left,Left,Right,Right]`.

Formally we say a value `A : paren list` is balanced if and only if either one of the following holds:

1. `A = [ ]`
2. There are balanced lists `M1 : paren list` and `M2 : paren list` such that `A = (Left :: M1) @ (Right :: M2)`

Now consider the following function:

```
(* is_balanced : (paren list * int) -> bool
 * REQUIRES: true
 * ENSURES: is_balanced (L, n) evaluates to true if L is balanced and false otherwise.
 *)
  fun is_balanced (([], n) : paren list * int) : bool = (n = 0)
   |     is_balanced (Left::xs, n) = is_balanced (xs, n + 1)
   |     is_balanced (Right::xs, n) = if n > 0 then is_balanced (xs, n - 1) else false
```

The way `is_balanced (L, n)` determines if `L` has balanced parenthesis is that it keeps track of the unmatched `Left` parenthesis in `L` in this way:

1. If `L` is the empty list, then `L` is balanced if `n`, the number of unmatched `Left` parenthesis, is zero.
2. If `L` is `Left::xs` for some `xs : paren list`, then we increment `n`, the number of unmatched `Left` parenthesis, and check if `xs` is balanced.
3. If `L` is `Right::xs` for some `xs : paren list`, first we check `n`, the number of unmatched `Left` parenthesis.
   (a) If `n > 0`, then that means there is at least one `Left` parenthesis we can match the current `Right` parenthesis with. Therefore, in this case we decrement `n` since we've matched one `Left` parenthesis with the current `Right` parenthesis, and then check if `xs` is balanced.

(b) If `n <= 0`, then that means there are no unmatched `Left` parenthesis we can match the current unmatched `Right` parenthesis with. Therefore, the parenthesis list we are examining is not balanced, so the function evaluates to `false` in this case.

Task 2.1 (15 pts). Prove that for all values `A : paren list, n : int,` such that `n >= 0` and `B : paren list`, if `A` is balanced, then

```
is_balanced(A @ B, n) = is_balanced(B, n).
```

You may cite the following lemmas without proof if you need them:

1. The function `@` is associative, i.e. for all lists `L1, L2, L3`,
   `(L1 @ L2) @ L3 = L1 @ (L2 @ L3)`.
2. The function `@` is total, meaning for all lists `L1, L2`,
   `L1 @ L2` evaluates to a value.
3. For all lists `L1`,
   `L1 @ [] = L1` and `[] @ L1 = L1`.
4. For all ints `x` and for all lists `L1, L2`,
   `x :: (L1 @ L2) = (x :: L1) @ L2`.

Hints: You should use strong induction on the length of `A`. Use the definition of a balanced parenthesis list and the code from `is_balanced` to help you.

Task 2.2 (2 pts). Given the result that you have proven, show that if `L : paren list` is a balanced list, then `is_balanced(L, 0) = true`.

Hint: This does not need any inductive argument - it follows easily!

Task 2.3 (3 pts). Write a function

```
pmatch : string -> bool
```

that determines if a string of parenthesis is balanced. Your implementation should use `is_balanced`. In addition, we have provided you with a helper function `string_to_parens : string -> paren list` in order to help convert between the representations.

Here is an example of what your code should look like in action!

```
- pmatch "";
val it = true : bool
- pmatch "()";
val it = true : bool
- pmatch "((hello))(there)";
val it = true : bool
- pmatch "())";
val it = false : bool
- pmatch "((()";
val it = false : bool
```

3 Full Trees

The tree datatype can be defined as the follows:

```
datatype tree =
    Empty
  | Node of tree * int * tree
```

Consider the following function:

```
(* skeleton : tree -> tree
 * REQUIRES: true
 * ENSURES: skeleton T = the tree obtained by replacing each node value of T
 *                       by its depth in T, counting the root node as depth 1
 *)

 fun skeleton T =
   let
       fun shape (Empty, _) = Empty
       | shape (Node(t1, _, t2), d) = Node(shape(t1, d+1), d, shape(t2, d+1))
   in
     shape(T, 1)
   end
```

Task 3.1 (2 pts). What is the type of `shape`?

Suppose that `T` is a full binary tree of height (or depth) `m` and `n` is a non-negative integer.

Task 3.2 (4 pts). What is the work to evaluate `shape(T, n)`? Show your work recurrence and give the big-O class.

Task 3.3 (4 pts). What is the span to evaluate `shape(T, n)`? Show your span recurrence and give the big-O class.

Task 3.4 (6 pts). In `hw04.sml`, write the function `census : int * tree -> int * int` such that `census(x, T) = (n, m)` where `n` is the number of integers in `T` that are strictly less than `x` and `m` is the number of integers in `T` that are strictly greater than `x`.

Tip: We have included a `treeEqual` function that you may use for testing.

4 Quicksorting a List

The *quicksort* algorithm for sorting lists of integers can be implemented in ML as a recursive function

```
quicksort : int list -> int list
```

that uses a helper function

```
part : int * int list -> int list * int list * int list
```

with the following specification:

```
(* REQUIRES: true
 * ENSURES:  part(x, L) ==> (L0, L1, L2)
 * such that L0 consists of the items in L that are less than x
 *           L1 consists of the items in L that are equal to x
 *           L2 consists of the items in L that are greater than x
*)
```

Another way to state this specification is:

> For all integers `x` and all integer lists `L`, `part(x, L)` returns a tuple of lists `(L0, L1, L2)` such that `L0` consists of the items in `L` that are less than `x`, `L1` consists of the items in `L` that are equal to `x`, and `L2` consists of the items in `L` that are greater than `x`.

The key idea is that one can sort a non-empty list `x::L` by partitioning `L` into three lists (the items less than `x`, the items equal to `x`, and the items greater than `x`), and then recursively sorting these three lists. The final result is obtained by combining the sorted sublists and `x`.

Task 4.1 (10 pts). Define an ML function

```
part : int * int list -> int list * int list * int list
```

that satisfies the above specification. You are allowed to use `Int.compare` but no other helper functions.

Now consider the following implementation of the function `quicksort`:

```
(* quicksort : int list -> int list *)
fun quicksort [] = []
  | quicksort (x::L) =
      let
        val (L0, L1, L2) = part(x,L)
      in
        (quicksort L0) @ L1 @ (quicksort L2)
      end
```

Recall the definition from lecture of sortedness on lists: A list of integers is $<$-*sorted* if each item in the list is $\leq$ all items that occur later in the list.

Task 4.2 (5 pts). Assuming that the function `part` satisifes its specifications (given above), try to prove the following proposition:

For all integer lists `L`, `quicksort L` evaluates to a $<$-sorted permuation of `L`.

Where does the proof break down? Why?

Task 4.3 (5 pts). In `hw04.sml` fix the function definition of `quicksort` such that the above proposition is true.

5 Tree Traversal

Recall that the in-order traversal of a tree `t` visits all of the nodes in the left subtree of `t`, then the node at `t`, and then all of the values in the right subtree of `t`.

The function `treeToList` below computes an in-order traversal of a tree.

```
(* treeToList : tree -> int list
 * REQUIRES: true
 * ENSURES: treeToList t ==> a list representing the in-order traversal of t
 *)
fun treeToList (Empty : tree) : int list = []
  | treeToList (Node(t1, x, t2)) = treeToList t1 @ (x :: treeToList t2)
```

Task 5.1 (5 pts). Define an ML function

```
traversal : tree * int list -> int list
```

that, when given a tree `t` and int list `A`, produces a list which represents the in-order traversal of that tree appended onto `A`. In other words,

```
traversal (t, A) = treeToList (t) @ A
```

Your implementation of `traversal` should not use `@` (or equivalents, or `treeToList`) and should run in linear work on the size of the tree (where size is the number of nodes in the tree).

6 Tree Size

Task 6.1 (14 pts). Prove, by structural induction on trees, that for all values `t : tree`,

```
size(t) = length(treeToList t)
```

Where the `size` function is defined as below:

```
fun size(Empty : tree) : int = 0
  | size(Node(l, x, r)) = size(l) + 1 + size(r)
```

You may use the following lemmas:

1. For all expressions `A : int list, B : int list` such that `A` and `B` reduce to values

   ```
   length (A@B) = length(A) + length(B)
   ```

2. For all expressions `x : int, L : int list` such that `x` and `L` reduce to values

   ```
   length(x ::  L) = 1 + length(L)
   ```

and you can use the definition of the `length` function for lists, as given in class, as well as basic properties of the list operations as used in class and the tree functions as defined in this assignment. You can also use basic properties of algebra such as commutativity and associativity. In addition you may assume that `treeToList` is total. If you use any of these, be sure to cite them in your proof, and justify that the preconditions are satisfied.

7 Heaps and Heaps of Heaps

You've seen one definition of a sorted tree, where all elements in the left subtree of a node are less than or equal to the node's value, and all elements in the right subtree of the node are greater than or equal to the node's value, and the subtrees are sorted. Consider the definition of sorted which defines a tree as a *maxheap*:

A `tree T` is a maxheap if one of the following holds:

1. `T` is `Empty`
2. `T` is a `Node(L, x, R)`, where `R, L` are maxheaps, `value(L)` $\leq$ `x, and value(R)` $\leq$ `x`

Here, `value(L)` is the value at the root node of the tree `L`; similarly for `R`.

In this section you will implement the ML function `heapify`, which, given an arbitrary `tree T` returns a maxheap with exactly the elements of `T`.

Task 7.1 (5 pts). Define an ML function

```
insert : (int * tree) -> tree
```

that, when `T` is a maxheap, `insert(x,T)` evaluates to a maxheap containing `x` and the elements of `T`.

Task 7.2 (10 pts). Define a recursive ML function

```
join: (tree * tree) -> tree
```

Such that when `T1 : tree` and `T2 : tree` are maxheaps, `join(T1, T2)` evaluates to a maxheap containing the elements from `T1` and `T2`.

Task 7.3 (10 pts). Define a recursive ML function

```
heapify : tree -> tree
```

which, given an arbitrary `tree T`, evaluates to a maxheap with exactly the elements of `T`.

You must use your previously defined functions `insert` and `join` to write `heapify`. The structure of the `tree` datatype should give you hints on how to make use of these two helper functions when writing `heapify`.

Tip: We have included `treeEqual` and `isHeap` functions that you may use for testing.