15-150 Summer 2016
Homework 04

Out: Tuesday, 24 May 2016
Due: Friday, 27 May 2016 at 23:59 EST

1 Introduction

This assignment 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.andrew.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 (usually either 0.0 or 1.0) 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.

Your `hw04.sml` file must contain all the code that you want to have graded for this assignment, 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 Friday, 27 May 2016 at 23:59 EST. 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, write the name and type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function.
5. Provide testcases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function:

```
(*  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 720 = fact 6
```

1.5 Style

In this and future homeworks, we will also grade your code for style. If your code for a task does not adhere to the style guidelines on the course website, you will temporarily receive a score of 0 for that task. You may then fix the problems with your code style and show it to a TA within two weeks of getting your graded assignment back. If you do this, you will get the number of points you would get if your original submission had proper style. If you do not do this, you will keep the grade of 0 for that task.

2 A Quick Sort of List Problem

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
```

with the following specification:

```
(* REQUIRES: true                                              *)
(* ENSURES:  part(x, L) ==> a pair of lists (A,B) such that    *)
(*           A consists of the items in L that are less than x *)
(*       and B consists of the items in L that are             *)
(*             greater than or equal to x.                     *)
```

Another way to state this specification is:

> For all integers `x` and integer lists `L`, `part(x, L)` returns a pair of lists `(A, B)` such that `A` consists of the items in `L` that are less than `x` and `B` consists of the items in `L` that are greater than or equal to `x`.

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

```
Note: (x, L): int * int list if x: int and L: int list whereas
(L1, L2): int list * int list if L1, L2: int list
```

Task 2.1 (10 pts). Define an ML function

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

that satisfies the above specification.

We now would like to implement the quicksort algorithm on lists. Your implementation should use the `part` function which you defined in the previous task. Do not introduce additional helper functions, and do not change the types or specs.

Task 2.2 (5 pts). Give big-O bounds on the work and span for `part`.

Task 2.3 (5 pts). Using `part`, define a recursive ML function

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

such that for all int lists `L`, `quicksort(L)` returns a sorted permutation of `L`.
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. The ML function `sorted` below checks for this property.

```
(* sorted : int list -> bool
 * REQUIRES: true
 * ENSURES: sorted L ==> true if L is <-sorted
 *                   ==> false otherwise
 *)
fun sorted ([] : int list) : bool       = true
  | sorted ([x] : int list) : bool      = true
  | sorted (x::y::L : int list) : bool  =
      (Int.compare(x,y) <> GREATER) andalso sorted(y::L)
```

3 Parentheses Matching

Consider the following strings:

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

A string of parentheses is well matched if at every point in the string there are no more right parentheses than left parentheses, but every left parenthesis is eventually matched with some right parenthesis. The empty string is a well matched string of parentheses. So, the first three strings are well matched while the last three are not.

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

```
datatype paren = LPAR | RPAR
```

A string of multiple parens will be represented as a list. For example, "(())" would be represented as `[LPAR,LPAR,RPAR,RPAR]`.

Task 3.1 (12 pts). Write a function

```
pmatch_help : (paren list * int) -> bool
```

such that `pmatch_help (L,0)` evaluates to `true` if `L` is well matched.

If you are confused as to why there is an `int` included in the input type of `pmatch_help`, think about what information you will need to keep in every recursive call.

Task 3.2 (3 pts). Write a function

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

that finishes your solution to the parentheses matching problem by using `pmatch_help`. 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
```

4 Tree Size

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 4.1 (15 pts). Prove, by structural induction on trees, that for all values `t : tree`,

$$\texttt{size(t)} \cong \texttt{length(treeToList t)}$$

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

$$\texttt{length (A@B)} \cong \texttt{length(A) + length(B)}$$

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

$$\texttt{length(x :: L)} \cong \texttt{1 + length(L)}$$

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.

5 Trick Or Treet

The distance between two nodes of a tree is defined as the number of edges on the shortest path between those two nodes. There are various algorithms for finding the shortest path between nodes in a general graph, including depth first search and breadth first search. We will focus on finding the shortest path between two nodes of a binary tree.

Notice that the shortest path between two nodes in a tree can be determined by finding the *lowest common ancestor* of those two nodes. The lowest common ancestor is the root of the smallest subtree which contains both nodes. (You can draw some examples to better understand this.) If the two nodes are n_1 and n_2, and t is the lowest common ancestor, the distance between n_1 and n_2 is

$$depth(t, n_1) + depth(t, n_2)$$

where *depth* returns the depth of the given node in the tree rooted at t.

In this section, you will be given a tree where the values at all nodes are 0 except for two nonzero nodes, one of which has value 1 and another which has value 2. You will be implementing a function

```
distance : tree -> int
```

which returns the distance between the two nonzero nodes in the input tree. Before implementing this function, you will first implement some helpers.

Task 5.1 (3 pts). Define an ML function which counts the number of nodes with nonzero value in a given tree. The signature is as follows:

```
countNonZero : tree -> int
```

Remember that you can use `<>` to check for inequality.

Task 5.2 (7 pts). Define an ML function

```
lca : tree -> tree
```

which takes a tree and finds the lowest common ancestor of the two nonzero nodes. The `countNonZero` function might be useful.

Task 5.3 (10 pts). Define an ML function

```
distance : tree -> int
```

which takes a tree and finds the distance between the two nonzero nodes of that tree. You may need a helper function. Remember that your nonzero nodes have values 1 and 2.

6 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 it satisfies one of the following invariants:

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`. Note that the standard definition of heaps requires a balanced tree, but for simplicity we exclude this condition.

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 6.1 (5 pts). Define an ML function

```
treecompare : tree * tree -> order
```

that, when given two trees, returns a value of type `order`, based on which tree has a smaller value at the root node. For example:

```
val a = Node(Node(Empty, 7, Empty), 3, Empty)
val b = Node(Empty, 4, Empty)
treecompare(a, b) ==> LESS
```

(You may assume that `Empty` is smaller than every nonempty tree.)

Task 6.2 (10 pts). Define a recursive ML function

```
swapDown : tree -> tree
```

with the following specification:

```
(* REQUIRES: the subtrees of t are both maxheaps
 * ENSURES:  swapDown(t) ==> returns t if t is Empty,
 *                           otherwise returns a maxheap
 *                containing exactly the elements in t.
 *)
```

Task 6.3 (5 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.

Task 6.4 (10 pts). Give the work and span for `swapDown` and `heapify`. You should derive the work and span recurrence relations for each function. Then, using these recurrences, give a big-O estimate for the work and span of each function.

Note: Remember that there are two ways to analyze a tree.