15-150 Fall 2012
Homework 04

Out: Wednesday, 18 September 2013
Due: Tuesday, 24 September 2013 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.

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 Tuesday, 24 September 2013 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, 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 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 720 = fact 6
```

2 Fastest Fib

The Fibonacci function `fib : int -> int`, as defined below, has exponential runtime.

```
fun fib n = if n<=2 then 1 else fib(n-1) + fib(n-2)
```

In lab last week you wrote a function `fibber : int -> int` that does the same job but in linear time.

Task 2.1 (10%). Write a function

```
fastfib : int -> int
```

that does the same job but in *logarithmic* time!

Your function should be based on the facts that, for all integers $k \geq 0$,

$$\begin{aligned} \text{fib}(2k) &= \text{fib}(k)(2\text{fib}(k+1) - \text{fib}(k)) \\ \text{fib}(2k+1) &= \text{fib}(k+1)^2 + \text{fib}(k)^2 \end{aligned}$$

For example:

```
- fib 10;
val it = 55 : int
- fib(5) * (2*fib(6)-fib(5));
val it = 55 : int
- fib 11;
val it = 89 : int
- fib(5)*fib(5) + fib(6)*fib(6);
val it = 89 : int
```

The apparent runtime (in the ML interpreter) for `fib(42)` is noticeably slow.

```
- fib 42;
val it = 267914296 : int (* after a few seconds! *)
```

Your function should evaluate `fastfib(42)` much faster!

By the way, design your function so that it reproduces exactly these results. Sometimes the Fibonacci series is taken to start at `n=0` rather than `n=1`, and that would lead to a slightly different function that's always "off by 1". The function defined above has `fib(1)=1` and `fib(2)=1`, and computes the Fibonacci numbers for `n` ≥ 1.

Solution 2.1 See solution in `hw04.sml`.

3 Series Trees

Consider the below datatype

```
datatype rtree = rEmpty
               | rNode of rtree * real * rtree
```

Recall the definition of a "partial sum" from Homework 02. In that assignment we considered the partial sum of the geometric series for $\frac{1}{2}$. Now we are interested in expanding this definition to any real r, not just $\frac{1}{2}$. We denote the partial sum of the first n terms of the series to be S_n. Also, we define the behavior of $S_n(r)$ as follows:

$$S_n(r) = 1.0 \text{ if } n = 0$$

$$S_n(r) = \sum_{k=0}^{n} r^k \text{ in general (for k} \geq 0)$$

We also define the following:

$$r^0 = 1.0$$

$$r^{m+1} = r \cdot r^m$$

for all real numbers r. For example, if $r = \frac{1}{3}$,

$$S_n = \sum_{i=0}^{n} (\frac{1}{3})^i = \frac{1}{3} + \frac{1}{9} + \frac{1}{27} + \dots$$

Task 3.1 (5%). Write a recursive ML function

```
geometricTree : int * real -> rtree
```

such that for all non-negative integers n, and all reals r, `geometricTree`(n, r) returns the "complete" (or "full") binary tree of size $2^n - 1$ and depth n, with the read value at each node being $S_i(r)$, where i is the depth of that node in the tree.

Your function design should exploit symmetry: when $n > 0$, the complete binary tree with depth n has identical left- and right children, each of which is also a complete binary tree. It is possible to build `geometricTree`(n, r) with a linear number of recursive calls; you will lose credit if your function makes an exponential number of recursive calls.

Remember to use `Real.==` to test for the equality of reals.

Solution 3.1 See solution in `hw04.sml`.

Task 3.2 (5%). Derive recurrence relations for an asymptotic estimate of $W_{\texttt{geometricTree}}(n)$ and $S_{\texttt{geometricTree}}(n)$, then using these recurrences, give a big-O estimate for both $W_{\texttt{geometricTree}}(n)$ and $S_{\texttt{geometricTree}}(n)$.

Solution 3.2

$$
\begin{aligned}
W_{geometricTree}(n) &= k + W_{geometricTreeHelp}(n) \\
\\
W_{geometricTreeHelp}(0) &= k_0 \\
W_{geometricTreeHelp}(n) &= k_1 + W_{geometricTreeHelp}(n-1)
\end{aligned}
$$

As shown in Lecture 5, the above recurrence yields $W_{geometricTreeHelp}(n) \in O(n)$. Since the linear work dominates the constant k, we have $W_{geometricTree}(n) \in O(n)$.

To understand the above recurrence, observe that the work of `geometricTree` is exactly the work of its helper function on the same argument depth, plus a constant amount of work involved in calling the helper function. The work of `geometricTreeHelp` is constant in the base case $d = 0$, while in the recursive case $d > 0$ there is some constant work and a recursive call on $d - 1$.

$$
\begin{aligned}
S_{geometricTree}(n) &= k + S_{geometricTreeHelp}(n) \\
\\
S_{geometricTreeHelp}(0) &= k_0 \\
S_{geometricTreeHelp}(n) &= k_1 + S_{geometricTreeHelp}(n-1)
\end{aligned}
$$

The recurrence for the span of `geometricTree` is the same as the recurrence for its work because all work done is sequential. As before, the recurrence is linear, so $S_{geometricTree}(n) \in O(n)$.

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
```

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`), recursively sorting these two lists. The final result is obtained by combining the sorted sublists and `x`.

Task 4.1 (10%). 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 trees. Your implementation should use the `part` function which you defined in the previous task. Do not introduce additional helper functions! Also, do not mess with the types or specs!

Solution 4.1 See solution in `hw04.sml`.

Task 4.2 (10%). 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 following definition (from lectures 5 and 6) 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
 * sorted L = true iff L is <-sorted
 *)
fun sorted ([ ] : int list) : bool      = true
  | sorted ([x] : int list) : bool      = true
  | sorted (x::y::L : int list) : bool  =
      (compare(x,y) <> GREATER) andalso sorted(y::L)
```

Solution 4.2 See solution in `hw04.sml`.

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`.

In class we covered the ML function `trav` below is an implementation of an in-order traversal of a tree.

```
(* trav : tree -> int list
 * trav t = the in-order traversal of t
 *)
fun trav (t:tree) : int list =
  case t of
       Empty           => [ ]
     | Node(t1, x, t2) => trav t1 @ (x :: trav t2)
```

Task 5.1 (5%). Define an ML function

```
traver : 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,

```
traver (t, A) = trav (t) @ A
```

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

Solution 5.1 See solution in `hw04.sml`.

6 Tree Size

Task 6.1 (10%). Prove, by structural induction on trees, that for all values `t : tree`,

```
size(t) = length(trav 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 values `A : int list, B : int list`

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

2. For all values `x : int, L : int list`

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

3. For all values `x : int, y : int, z : int`

   ```
   x + (y + z) = x + y + z
   ```

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. If you use any of these, be sure to cite them in your proof.

Solution 6.1

Proof. We proceed by structural induction on `t : tree`.

Base case: `t = Empty`. Then

$$\begin{aligned} \texttt{size t} &= 0 && \text{first clause of } \texttt{size} \\ \texttt{length(trav t)} &= \texttt{length([])} && \text{first clause of } \texttt{trav} \\ &= 0 && \text{defn. of } \texttt{length} \end{aligned}$$

Hence, by transitivity of $=$, `size t = length(trav t)`.

Inductive Hypothesis Assume for some values `l : tree` and `r : tree` that `size l = length(trav l)` and `size r = length(trav r)`.

Inductive case: `t = Node(l, x, r)`. Then

$$\begin{aligned}
\texttt{size t} &= \texttt{size(l) + 1 + size(r)} && \text{clause 2 of } \texttt{size} \\
&= \texttt{length(trav l) + 1 + length(trav r)} && \text{IH} \\
&= \texttt{length(trav l) + length(x :: trav r)} && \text{lemma 2} \\
&= \texttt{length(trav l @ (x :: trav r))} && \text{lemma 2} \\
&= \texttt{length(trav t)} && \text{clause 2 of } \texttt{trav}
\end{aligned}$$

Hence, by transitivity of $=$, `size t = length(trav t)`. By structural induction `t`, the claim holds for all values `t : tree`. □

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 *minheap*

A `tree` t is a minheap if it obeys the following invariants:

1. t is `Empty`
2. t is a `Node(L, x, R)`, where `R, L` are minheaps and `value(L), value(R)` $\geq$ `x`

Here, `value` is the value at the root node of the tree.

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

Task 7.1 (5%). 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 larger value at the root node. For example:

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

Solution 7.1 See solution in `hw04.sml`.

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

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

with the following specification:

```
(* REQUIRES the subtrees of t are both minheaps
 * ENSURES swapDown(t) = if t is Empty or all of t's immediate children are
 *         empty then just return t, otherwise returns a minheap which
 *         contains exactly the elements in t.
 *)
```

Solution 7.2 See solution in `hw04.sml`.

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

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

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

Solution 7.3 See solution in `hw04.sml`.

Task 7.4 (10%). Assume that your implementation of `swapDown` is correct (it works according to spec). Using this, prove that your implementation of `heapify` is correct.

Solution 7.4

Proof. **Want to Show (Theorem): `heapify(t)` evaluates to a heap containing the same elements as `t`**

Proof by Structural Induction on t

Base Case: `t = Empty`

```
heapify Empty =>* Empty (by code step)
```

Inductive Hypothesis: Assume for some trees L, R that heapify(L) = LH, and heapify(R) = RH

Inductive Statement:

```
heapify Node(L, x, R)
 =>* swapdown(Node(heapify L, x, heapify R))          [by code step]
 =  swapdown(Node(LH, x, RH))                         [by IH]
 =  H is a heap                                       [by correctness of swapdown]
```

By the induction hypothesis, heapify(L) contains all of the elements of L, and heapify(R) contains all of the elements of R. Since swapdown does not add or remove elements any elements, the result contains all the elements of L, all the elements of R, and x. These are exactly all of the elements of T.

□

Task 7.5 (10%). Give the work and span for `swapDown` and `heapify`. You should give a recurrence relation and closed form for each.

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

Solution 7.5 **Work**

Let $W_{\text{swapDown}}(d)$ be the time for `swapDown` on a tree of depth d, and $W_{\text{heapify}}(d)$ be the time for `heapify` on a tree of depth d.

$$W_{\text{swapDown}}(d) = \begin{cases} k_0 & \text{if } d = 0 \\ k_1 + W_{\text{swapDown}}(d-1) & \text{if } d > 0 \end{cases}$$

The closed form of this recurrence is $k_0 + \sum_{i=1}^{d} k_1 = k_0 + d \cdot k_1$

From the Lecture 5 notes, we know that $W_{\text{swapDown}}(d) \in O(d)$.

$$W_{\text{heapify}}(d) = \begin{cases} k_0 & \text{if } d = 0 \\ k_1 + 2W_{\text{heapify}}(d-1) + W_{\text{swapDown}}(d) \end{cases}$$

The closed form of this recurrence is $k + (k_0 + k_3 + k_1 d)(2^{d+1} - 1) - (2 * 2^d d - 2^d + 1)$. Because we know that it is easy to create arithmetic errors when finding the closed form, if your dominating term was correct (if you gave a big O), you received full credit.

From the Lecture 5 notes, we know that $W_{\text{heapify}}(d) \in O(d2^d)$.

Span

Let $S_{\text{swapDown}}(d)$ be the time for `swapDown` on a tree of depth d, and $S_{\text{heapify}}(d)$ be the time for `heapify` on a tree of depth d.

$$S_{\text{swapDown}}(d) = \begin{cases} k_0 & \text{if } d = 0 \\ k_1 + S_{\text{swapDown}}(d-1) & \text{if } d > 0 \end{cases}$$

The closed form of this recurrence is $k_0 + \sum_{i=1}^{d} k_1 = k_0 + d \cdot k_1$

From the Lecture 5 notes, we know that $S_{\text{swapDown}}(d) \in O(d)$.

$$S_{\text{heapify}}(d) = \begin{cases} k_2 & \text{if } d = 0 \\ k_3 + S_{\text{heapify}}(d-1) + S_{\text{swapDown}}(d) \end{cases}$$

The closed form of this recurrence is

$$k_2 + \sum_{i=1}^{d} \left(k_3 + k_0 + \sum_{j=i}^{i} k_1 \right) = k_2 + d \cdot k_3 + d \cdot k_0 + \sum_{i=1}^{d} i \cdot k_1 = \frac{1}{2} d^2 k_1 + d(k_0 + k_3) + k_2$$

From the Lecture 5 notes, we know that $S_{\text{heapify}}(d) \in O(d^2)$