15-150 Fall 2014
Homework 04

Out: Wednesday, 17 September 2014
Due: Tuesday, 23 September 2014 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, 23 September 2014 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 to be passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes when applied to an argument that satisfies the assumptions specified in the `REQUIRES` statement.
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)
```

Task 2.1 (10 pts). 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 $k \geq 0$,

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

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, you should 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.

Also, if you were to open SML's `IntInf` library (don't do this in the file you hand in, it will cause you to fail the check script and you will lose points), you could compute even larger fibonacci numbers which would go beyond the range that SML's ints support.

3 Tree Traversal

There are three common traversal orders of a tree: pre-order, in-order, and post-order traversals. The prefix is in reference to where in the traversal the current node is visited.

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

The function `trav` below computes the pre-order traversal of a tree.

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

Task 3.1 (10 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 pre-order traversal of that tree appended onto `A`. In other words,

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

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

Task 3.2 (5 pts). Define an ML function

```
chop : int * int list -> int list * int list
```

such that, when given an integer `i` and a list `L`, `chop(i,L)` returns a pair of lists `(L1, L2)` where `L = L1 @ L2` and `length L1 = i`. You may assume that `i` is less than length of `L`.

Task 3.3 (10 pts). Define an ML function

```
list2tree : int list -> tree
```

that, when given an int list `L`, `list2tree L` produces a balanced tree for which the pre-order traversal list is `L`. You may find `chop` to be useful in your implementation of `list2tree`.

Task 3.4 (5 pts). Give the work and span of `trav T`, assuming that `T` is a balanced, full tree. Your answer should contain a recurrence relation and a big-O bound.

Task 3.5 (5 pts). Give the work and span of `trav T`, assuming that `T` is the worst case input. Your answer should contain a recurrence relation and a big-O bound.

Remember that `L1@L2` takes work O(`length L1`) when `L1` and `L2` are values. Also, note that `trav T` is a list of length `size T`.

4 Tree Size

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

$$\texttt{size(t)} = \texttt{length(trav t)}$$

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`

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

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

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

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

5 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 the helper functions

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

where `part` has the following specification:

```
(* part : int * int list -> int list * int list * int list
 * REQUIRES: true
 * ENSURES: part(x, L) =>* a 3-tuple of lists (A, E, B) such that
 *          A consists of the items in L that are less than x,
 *          E consists of the items in L that are equal to x,
 *          and B consists of the items in L that are greater than x.
 *)
```

Another way to state this specification is:

> For all integers `x` and integer lists `L`, `part(x, L)` returns a 3-tuple of lists `(A, E, B)` such that `A` consists of the items in `L` that are less than `x`, `E` consists of the items in `L` that are equal to `x`, and `B` consists of the items in `L` that are greater than `x`.

Example: `part(2, [1,2,1,3,2]) = ([1,1],[2,2],[3])`.

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 the first and the last lists. The final result is obtained by combining the sorted sublists and `x`.

Task 5.1 (10 pts). Define an ML function

```
part : int * int list -> 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, as well as the `pivot` function which we have provided. Do not introduce additional helper functions, and do not change the types or specs.

We have provided the `pivot` function for you which satisfies the following specification:

```
(* pivot : int list -> int
 * REQUIRES: the input list is non-empty
 * ENSURES: pivot(L) =>* an element of the list L
 *)
```

You must use the `pivot` function for providing the first argument to `part`.

Task 5.2 (10 pts). Using `part` and `pivot`, 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 7 and 8) 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 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)
```

Task 5.3 (5 pts). Give the work and span of `part L`. Your answer should contain a recurrence relation and a big-O bound for each.

Task 5.4 (5 pts). What type of list is the worst case input to `quicksort`? What type of list is the best case? Explain your answers by reasoning about the evaluation of `quicksort` and how it behaves on the type of list that you answered.

Task 5.5 (5 pts). Give the worst case work and span of `quicksort` assuming that the implementation of `pivot` that you are given has constant work and span $(W_{\texttt{pivot}}, S_{\texttt{pivot}} \in O(1))$, but returns the first element of the input list. Your answer should contain a recurrence relation and a big-O bound for each.

Task 5.6 (5 pts). Give the work and span of `quicksort` assuming that the implementation of `pivot` that you are given has linear work and span $(W_{\texttt{pivot}}, S_{\texttt{pivot}} \in O(n))$, but always returns the median element of the input list. Your answer should contain a recurrence relation and a big-O bound for each.