COS 326 Midterm Exam, Princeton University, Fall 2022

Andrew W. Appel

This was a **closed book, 100-minute** exam.

Problem 1

In this program, what are the free variables of function g?

```
let f (x: int) (y: foo) =
   let u = h y x in
   let g (z: int) =
     let x = u+z in
       h y x
    in g
```

Problem 2

In answering this question you may use the OCaml List library if you want to.

This datatype describes the result of a comparison:

```
type order = LESS | EQUAL | GREATER
```

A. Consider a "partition" function that you might use in a Quicksort. It is given a *single-argument* "test" function of type **`element->order`**, and it is given a list of elements, and it applies the test function to each element, returning three lists as a result: the elements that test "less", those that test "equal", and those that test "greater". But the partition function should be polymorphic, i.e. work on any element type. What is the type of the partition function?

B. Now implement the partition function. The three result lists, if you were to concatenate them together, should be a permutation of the original list, but within each of the result lists it doesn't matter what order they're in.

You can work on Parts C,D even if you don't do Part B

C. Consider a function that is given a *nonempty* list of elements, and returns the middle element—that is, an item about halfway down the list (which has nothing to do with the *value* of the item). Choose a name for this function and write down its type. It's inconceivable that the input list is empty, but do something appropriate for inconceivable cases.

D. Now implement the function (from Part C) in OCaml. Hint: One parameter of your recursive function should walk down the list twice as fast as another parameter.

You can work on Parts E,F,G even if you don't do Part D

E. Quicksort, in choosing a pivot element, would be perfectly correct if it just chose the first element of the list; and that would certainly be easier than using the function you defined in part C. Why would it be a bad idea to simply choose the first element as pivot?

F. Write down the type of a polymorphic **`quicksort`** function that takes a comparison function as an argument (comparing two elements and telling LESS, EQUAL, or GREATER).

G. Now write the recursive quicksort function. It should be polymorphic on the element type, and it should take the comparison function as an argument. The algorithm is, of course: find some arbitrary "pivot" element that's in the list; partition the list into the less, equal, and greater sublists; sort the "less" and "greater" lists; and concatenate back together.

Problem 3

Prove this theorem: for all u, `addup` u `= foldr (+) (translate` u`) 0`.

```
type one_two = One of int | Two of int * one_two * int

let rec addup (s: one_two) : int =
match s with
| One i -> i
| Two (i,r,j) -> i + addup r + j

let rec translate (s: one_two) : int list =
match s with
| One i -> i::[]
| Two (i,r,j) -> i :: j :: translate r

let rec foldr (f: 'a -> 'b -> 'b) (al: 'a list) (z: 'b) : 'b =
match al with
| h::tl -> f h (foldr f tl z)
| [] -> z
```