ICS 313 Homework 3

Functional Programming in Ocaml

1. Goals and Overview

The goal of this assignment is to reinforce your knowledge of functional programming, introducing you to ocaml, and giving you additional experience using recursion with lists.

Success in this assignment requires ability to recursively process lists, and the ability to use ocaml to create, represent, and use functions.

2. Using Ocaml

For this class, you are welcome to install ocaml on a system you control, or to use the ocaml available on uhunix.

After starting ocaml, you will get a pound sign (#) as a prompt. You can now type expressions which ocaml will compile and evaluate. Remember to put two semicolons (;;) after everything you want evaluated.

Abundant documentation for ocaml is available from the manual.

The keyword `let` is used to declare global variables and functions. Recursive functions are declared with `let rec`.

3. Functional Programming in Ocaml

In general, functional programming includes higher-order functions, which are functions that take functions as parameters, return functions as results, or both.

An example of a higher-order function is compose, which takes as parameters two other single-argument functions, and returns a function that combines the two. Here is an implementation of compose in ocaml:

```
let compose f1 f2 = (fun x -> f1 (f2 x));;
```

This function first applies f2 to its parameter, then applies f1 to the result. For example,

```
compose List.rev List.hd [[1; 2; 3]; [4]; [5; 6]];;
```

takes the head (first element) of the argument, and then reverses it, yielding the value [3; 2; 1].

Notice that in Ocaml all list elements have to have the same type, so [[1; 2; 3]; 4; 5; 6] is not a valid list because the first element is an int list, and the remaining elements are just int.

I can also define conventional map and reduce functions directly in ocaml:

```
let rec map f = function
      | [] -> []
      | first :: rest -> f first :: map f rest;;
let rec reduce f init = function
      | [] -> init
      | first :: rest -> f first (reduce f init rest);;
```

In this definition, reduce takes three arguments, a function, an initial value (used when the list is nil), and a list.

Also note that the first argument to map takes one parameter, and the first argument to reduce takes two parameters.

Here are simple examples of using these functions:

```
# map (fun x -> x + 1) [5; 7; 9];;
- : int list = [6; 8; 10]
# reduce (fun x y -> x + y) 0 [1; 2; 3];;
- : int = 6
```

4. Assignment

4.1 mapreduce

Create a mapreduce function which maps a single-parameter function to every element of a linked list, then reduces the resulting list using a two-parameter function. The third argument to mapreduce is the initial value for reduce. For example,

```
# mapreduce (fun x -> x + 1) (fun x y -> x + y) 0  [1; 2; 3];;
- : int = 9
```

For this section, you **must** implement mapreduce by using compose (defined above) to combine reduce with map. To be clear,

- the first argument to mapreduce is the function to be used with map, the second argument is the function to be used with reduce, and the third argument is the initial value to be used with reduce
- map processes the list first, and reduce reduces the result of the call to map (this is the normal sequence of mapreduce).

Summary: use compose to write the code of the mapreduce function which takes two functions (one with a single parameter, the other a binary operator) along with an initial reduce value and a list of values to mapreduce, and returns a single value.

4.2 Dot product with mapreduce

The inner product or dot product of two vectors (a1 a2 ... an) and (b1 b2 ... bn) is the sum of the products of the paired elements, so a1*b1 + a2*b2 + ... + an*bn. We are representing the two vectors as lists of numbers that can be used with mapreduce.

The dot product can be easily expressed as a mapreduce problem. Implement a function dot-product of two arguments (the two lists). You first **must** implement a compose2 and a map2 functions that take two list parameters instead of one, and then you **must** implement the corresponding map2reduce function using compose2 and map2.

So for example,

```
# map2 (fun x y -> x * y) [1; 2; 3] [4; 5; 6];;
- : int list = [4; 10; 18]
# map2reduce (fun x y -> x * y) (fun x y -> x + y) 0 [1; 2; 3] [4; 5; 6];;
- : int = 32
```

Your code will be simpler if you use pattern matching to detect the end of a linked list. You can pattern match on two or more lists by joining them in a tuple (in this case, a two-element tuple, also known as a pair). For example:

```
match (list1, list2) with
  | ([], _) -> ...
  | (_, []) -> ...
  | (first1 :: rest1, first2 :: rest2) -> ...
```

Summary: change the mapreduce from step 1 to map to two lists instead of one, and use map2reduce to write the inner-product of two vectors.

4.3. Mapping multiple functions to the same list

Now expand map to a function mapf which takes as its first parameter a list of functions, rather than a single function, and maps each of those functions to the list. The result is a list of lists. For example,

```
# let even x = (x mod 2) = 0;;
val even : int -> bool =
# let odd x = (x mod 2) = 1;;
val odd : int -> bool =
# mapf [even; odd] [0; 1; 2; 3; 4; 5; 6; 7];;
- : bool list list =
[[true; false; true; false; true; false; true; false];
 [false; true; false; true; false; true; false; true]]
```

You might find it convenient to use map in implementing mapf.

Because in ocaml all elements of a list have to have the same type, the parameter types and return types of all the functions in the list must be the same.

4.4 Dot product of a pair of lists with mappairreduce

Since ocaml provides tuples, we have a different way to implement the dot product of a single parameter, a pairs of lists, instead of two separate parameters, each of them a list. For this section, create a mappairreduce that you can use as follows:

```
mappairreduce (fun (x, y) -> x * y) (fun (x, y) -> x + y) 0 ([1; 2; 3], [4; 5; 6]);;
```

Note that the fourth argument to mappairreduce is a pair of lists. In the same way, the arguments to each of the map and reduce functions are pairs of values.

You may adapt any of the code in previous sections to implement mappairreduce, or write it from scratch, as you prefer. You may use a version of compose, but for this section you don't have to, so if you prefer you are welcome to write mappairreduce recursively, or in any way that works in ocaml.

5. Turning in your assignment

Use Laulima to turn in one file with all your code.

Once you log into Laulima and select the ICS 313 site, on the left-hand side will be the assignments tab.