CIS 500 Software Foundations (Fall 2002)

Homework Assignment 1

Basic OCaml Programming

Due: Friday, September 13, 2002, by noon

Instructions:

- Solutions should be submitted in hardcopy form to Christine Metz in 556 Moore. Include your name and email address at the top of your solution set.
- Solutions to all problems are provided at the end of this handout, so that you can check your answers before turning them in. You may also peek at the solutions for help if you feel stuck, but (obviously) we do not recommend simply copying the solution.
- All the homeworks will be graded on a binary scale: you must turn in solutions to all required problems to receive credit for the assignment.

  One problem below is marked optional. You may turn in a solution for this problem or not, whichever you prefer.
- Collaboration on homework assignments is *encouraged*. You may talk with people about the problems, work together on solving them, and discuss any aspect of the problem in the newsgroup, etc.

  However, even if you have worked all the problems together with others, you must turn in your own copy of the solutions. We strongly recommend that you talk through the solutions with others (if you wish) but then go away and write them out by yourself, to make sure that you understand them completely.
- The instruction given in Chapter 2 of *Introduction to Objective Caml* (see below) should suffice for learning to interact with the OCaml compiler and "toploop" on eniac and gradient (don't worry if the version numbers don't match). If you would like to install ocaml on your own machine, binaries for various platforms as well as a source distribution are available here:

  `http://caml.inria.fr/ocaml/distrib.html`

  As noted here, some Linux distributions also come with OCaml packages ready to install, so you may want to check your CDs. OCaml is straightforward to build from sources on most UNIX-like systems if you are accustomed to doing such things, however we do not have the resources to help everyone with installing OCaml at home – it's up to you.

Reading assignment: Before beginning the programming exercises below, read Chapters 1 through 5 of Jason Hickey's *Introduction to Objective Caml*. Don't worry if you find Chapter 5 a little dense—for the moment, all you need from it is the examples of simple list processing functions.

1 Exercise The nth *Fibonacci number*, fib_n, is defined recursively as follows: $\text{fib}_0 = 0$, $\text{fib}_1 = 1$ and for all $n \geq 2$, $\text{fib}_n = \text{fib}_{n-1} + \text{fib}_{n-2}$. This function can be straightforwardly translated into an OCaml function: write such an implementation `fib`.

Try out your function on inputs 7, 20, and 33 (the results should be 13, 6765 and 3524578, repectively).

Even on a very fast machine, you should see a noticeable delay on calculating fib_{33}. Finding Fibonacci numbers much bigger than this one will take much longer than you are likely to want to wait. (To see why this is the case, consider how many additions your function must perform to compute a given Fibonacci number.)

As you can see, it's often not the case that the most obvious implementation will yield the best performance. Write a *tail-recursive* function `fib_tr` that also computes the nth Fibonacci number. Recall that a tail-recursive function is a recursive function in which there is at most a single recursive call in each control path, and that recursive call must be the final statement in that control path. To write `fib_tr`, you will need to write a tail-recursive auxiliary function that does all of the real work and that `fib_tr` just calls once. Note that only `fib_tr` will need to call this auxiliary function, so you might as well define it in a `let` block inside the body of `fib_tr` – see section 3.1.1 of Jason Hickey's notes for a simple example of this. Try `fib_tr` on 7, 20, and 33, and (subjectively) compare its performance to your prior implementation.

2 Exercise Write a function `evenmembers` that takes a list of integers as input and returns a list containing just the even members of the original list. For example:

```
# evenmembers [1;2;3;5;6;8;9];;
: int list = [2; 6; 8]
```

3 Exercise Write a function `append` that takes two lists and returns a new list containing the members of the first list followed by the members of the second list. For example:

```
# append [1;2;3] [4;5];;
: int list = [1; 2; 3; 4; 5]
```

4 Exercise [Optional]

Write functions that implement basic set operations using lists. In particular, you should write the following functions:

- `member x s` : returns `true` if `x` is in `s`; `false` otherwise
- `add x s` : returns a new set with `x` added to `s` if it is not already there
- `union s1 s2` : returns a new set that is the union of `s1` and `s2`
- `inter s1 s2` : returns a new set this is the intersection of `s1` and `s2`
- `subset s1 s2` : returns `true` is `s1` is a subset of `s2`; `false` otherwise

You may assume that `union`, `inter`, and `subset` get only sets as their arguments, i.e. that there are no repetitions. Said another way, you can assume that sets are only constructed with `add`. An example:

```
# let s1 = add 1 (add 2 (add 3 (add 4 (add 5 []))));;
val s1 : int list = [1; 2; 3; 4; 5]
# let s2 = add 6 (add 7 (add 4 (add 8 (add 9 (add 5 (add 0 []))))));;
val s2 : int list = [6; 7; 4; 8; 9; 5; 0]
# member 2 s1;;
```

```
- : bool = true
# member 2 s2;;
- : bool = false
# union s1 s2;;
- : int list = [1; 2; 3; 6; 7; 4; 8; 9; 5; 0]
# inter s1 s2;;
- : int list = [4; 5]
# subset s1 s2;;
- : bool = false
# subset (inter s1 s2) s2;;
- : bool = true
```

5 Exercise Write a function `permute` which takes a list and returns a list of all possible permutations of the input list. For example:

```
# permute [1;2;3];;
- : int list list =
[[1; 2; 3]; [2; 1; 3]; [2; 3; 1]; [1; 3; 2]; [3; 1; 2]; [3; 2; 1]]
# permute [1;2];;
- : int list list = [[1; 2]; [2; 1]]
# permute [1];;
- : int list list = [[1]]
# permute [];;
- : '_a list list = []
```

(The order of the results in your solution may vary from what is shown above.)

6 Debriefing

1. Approximately how many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. Did you work on it mostly alone, or mostly with other people?
4. How deeply do you feel you understand the material it covers (0%–100%)?
5. Any other comments?

This question is intended to help us calibrate the homework assignments. Your answers will not affect your grade.

Solutions

1.

```
let rec fib = function
  0 -> 0
| 1 -> 1
| n -> fib (n - 2) + fib  (n - 1);;


let fib_tr n =
  let rec fib_loop previous sum counter =
    match counter with
      0 -> sum
    | n -> fib_loop sum (previous + sum) (counter - 1)
  in fib_loop 1 0 n
```

2.

```
let iseven n = ((n mod 2) = 0)

let rec evenmembers = function
  [] -> []
| x::xs -> if (iseven x) then x::evenmembers xs
           else evenmembers xs
```

3.

```
let rec append l1 l2 =
  match l1 with
    [] -> l2
  | x::xs -> x::(append xs l2)
```

4.

```
let rec member x s =
  match s with
    [] -> false
  | y::ys -> (x = y) || member x ys

let add x s = if (member x s) then s else x::s

let rec union s1 s2 =
  match s1 with
    [] -> s2
  | x::xs -> add x (union xs s2)

let rec inter s1 s2 =
  match s1 with
```

```
    [] -> []
  | x::xs -> if (member x s2) then x::(inter xs s2) else inter xs s2

let rec subset s1 s2 =
  match s1 with
    [] -> true
  | x::xs -> (member x s2) && (subset xs s2)
```

5. There are several ways of listing permutations: one is below. This solution uses the OCaml standard library function `List.append` to append two lists, but you could also try using your own `append` function from exercise 3.

```
let rec percolate_right element left right =
  match right with
    [] -> [List.append left [element]]
  | rightHead::rightTail ->
        (List.append (List.append left [element]) right)
          ::(percolate_right element (List.append left [rightHead]) rightTail)

let rec insert element = function
    [] -> []
  | firstList::restLists ->
      List.append (percolate_right element [] firstList)
                  (insert element restLists);;

let rec permute = function
    []    -> []
  | [x]   -> [[x]]
  | x::xs -> insert x (permute xs)
```