CIS 500 — Software Foundations

Homework Assignment 1

Basic OCaml

Due: Monday, September 11, 2006, by noon

Instructions:

- Solutions must be submitted electronically (in ascii, postscript, or PDF format). Follow the instructions here:

  `http://www.seas.upenn.edu/~cis500/homework.html`

  Submit all of your solutions in a single ML source file. Put your name(s) in a comment at the top.

- Collaboration on this homework assignment is *strongly encouraged.* If you work on the assignment with others, please turn in a single set of solutions bearing all of your names. Everyone will receive the same grade.

- The instructions given in Chapter 2 of *Introduction to Objective Caml* (see below) should suffice for learning to interact with the OCaml compiler and "top loop" on SEAS machines. (Don't worry if the compiler 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`

  Many Linux distributions and other package managers (Fink on OSX, CygWin on Windows) offer OCaml packages ready to install. OCaml is also 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 Write a function `oddmembers` that takes a list of integers as input and returns a list containing just the odd members of the original list. For example:

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

2 Exercise Write a function `interleave2` that takes two lists and returns a new list containing the members of the first and second lists in alternating order. For example:

```
# interleave2 [1;2;3;4] [5;6];;
: int list = [1; 5; 2; 6; 3; 4]
```

3 Exercise Write a function `interleave3` that takes *three* lists as arguments and returns a new list containing the members of the first, second, and third lists in alternating order. For example:

```
# interleave3 [1;2;3;4] [5;6] [7;8;9];;
: int list = [1; 5; 7; 2; 6; 8; 3; 9; 4]
```

4 Exercise Write a function `interleaven` that takes a *list* of lists as an argument and returns a new list containing the members of the sub-lists of this list in alternating order. For example:

```
# interleaven [ [1;2;3;4]; [5;6]; [7;8;9] ];;
: int list = [1; 5; 7; 2; 6; 8; 3; 9; 4]
```

5 Exercise The nth *Fibonacci number*, fib_n, is defined recursively as follows: $\mathrm{fib}_0 = 0$, $\mathrm{fib}_1 = 1$ and for all $n \geq 2$, $\mathrm{fib}_n = \mathrm{fib}_{n-1} + \mathrm{fib}_{n-2}$. Write an OCaml function `fib` that implements this algorithm. 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.

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. Try `fib_tr` on 7, 20, and 33.

6 Exercise A *multiset* (sometimes called a *bag*) is a set where each element may appear any finite number of times. Write functions that implement basic multiset operations using lists. In particular, you should write the following functions:

- `add x s` : returns a new multiset with `x` added to `s` (so that the count of `x` is increased by 1)
- `count x s` : returns the number of times that `x` appears as an element of `s`
- `member x s` : returns `true` if `x` appears at least once in `s`; `false` otherwise
- `subset s1 s2` : returns `true` if the count of each element in `s1` is less than or equal to its count in `s2`, and `false` otherwise
- `union s1 s2` : returns a new multiset that is the union of `s1` and `s2` (i.e., where the count of each element is the maximum of its counts in `s1` and `s2`)
- `inter s1 s2` : returns a new multiset that is the intersection of `s1` and `s2` (i.e., where the count of each element is the minimum of its counts in `s1` and `s2`)

For example:

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

The order in which the elements of a multiset are stored does not matter, so don't worry if the lists in your implementation are permutations of the ones here. Also, you do not need to worry about the efficiency of your solution.

7 Exercise (Optional) Implement your favorite list sorting function in OCaml, using just the features we have discussed in class. (For example, try mergesort or quicksort.)

8 Debriefing

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

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