CIS 500 Software Foundations

Homework Assignment 3

OCaml, Functional Programming

Due: Monday, October 3, 2005, by noon

Submission instructions:

You must submit your solutions electronically, in ASCII format. Electronic solutions should be submitted following the same instructions as last time; these can be found at `http://www.seas.upenn.edu/~cis500/homework.html`. Do **not** email your solutions to us.

IMPORTANT: Each exercise includes directions on how to submit that exercise. In particular, please pay attention to file names.

1 Exercise The $3n + 1$ problem. Please submit all your code and answers for this exercise in a file called `threen.ml`.

1. Write a function

   ```
   string_of_list : ('a -> string) -> 'a list -> string
   ```

   that returns a `string` representation of a list, given a function that produces a `string` representation of a list item. The representation should be similar to OCaml's printed representation in the top level, i.e., it should use brackets to enclose the list and a semi-colon to separate items. (This function may help you with the subsequent programming tasks.)

2. Write a function

   ```
   until : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a list
   ```

   such that `until p f x` returns a list of the results of successive applications of `f` to `x` until a value is reached that satisfies the predicate `p`. (This value should be included in the list.)

   In other words, the first item in the resulting list is equal to `x`. Each subsequent item is equal to the application of `f` to the previous item in the list. All but the last item in the list should return `false` when supplied to `p`.

3. Using the function `until` that you wrote in part 2, write a function

   ```
   range : int -> int -> int list
   ```

   such that `range m n` returns a list of all the integers between `m` and `n`, inclusive, in ascending order. If `n` < `m`, then any result is acceptable but the function should still terminate. You may not use any local, global, or anonymous helper functions to define `range`; you may only use function applications.

   To accomplish this, you will need to note that a binary infix operator can be used, syntactically, as the name of a two-argument, curried function when it is enclosed in parentheses, e.g., the successor function could be defined as

   ```
   let succ = (+) 1
   ```

   although `succ` is, in fact, already defined in OCaml as a built-in function.

4. The $3n + 1$ algorithm does the following on input i (where $1 \leq i$):

   - If $i = 1$, then stop.
   - If i is even, then let $i/2$ be the new value of i. Repeat.

- If i is odd, then let $3n+1$ be the new value of i. Repeat.

It is conjectured that for any input i, this algorithm halts. (Don't worry, it halts on every input we ask you to run this algorithm on.)

For example, on input 3, the algorithm computes the following sequence of values: $3, 10, 5, 16, 8, 4, 2, 1$. The algorithm took 7 steps here, and the highest intermediate value reached is 16.

We can implement one step of this algorithm as follows:

```
let rec step (n : int) : int =
   if n mod 2 = 0 then
      n / 2
   else
      3*n + 1
```

Write a function

```
step_count : int -> int list
```

that, given a positive integer `n`, returns a list of `n` integers. The ith integer in the list (counting the leftmost item as 1, not 0) should be the number of steps taken by the $3n+1$ algorithm when started using i as the initial input. You may not use any local, global, or anonymous helper functions to do this; you may only use function applications. However, you may use any functions defined above, any built-in OCaml functions (i.e., in the `Pervasives` module), and any functions from the `List` module. Be careful to avoid an off-by-one error in your calculation!

5. Now write a function

```
step_max : int -> int list
```

that, given a positive integer `n`, returns a list of `n` integers. The ith integer in the list (counting the leftmost item as 1, not 0) should be the maximum value achieved by the $3n+1$ algorithm when started using i as the initial input. Again, you may not use any local, global, or anonymous helper functions; you may only use function applications. Hint: Pay special attention to the two sorts of folds in the `List` module.

6. What is the median number of steps taken by they $3n+1$ algorithm when given inputs ranging from 1 to 777? Give the expression you used to compute your answer, and put the answer itself in a comment in the source file. (Here you *may* use any helper functions, anonymous functions, built-in OCaml functions, or functions from the `List` module to compute the value.)

7. What is the maximum value ever achieved during the course of running the $3n+1$ algorithm on any input less than or equal to 20,000? Which inputs achieve this maximum value? Give the expressions you used to compute your answers, and put the answers in a comment in the source file. (Again, you *may* use any helper functions, anonymous functions, built-in OCaml functions, or functions from the `List` module to compute the values.)

8. What is the smallest value that does *not* ever appear during the course of running the $3n+1$ algorithm on any input less than or equal to 20,000? Give the expressions you used to compute your answer, and put the answer in a comment. (Again, you *may* use any helper functions, anonymous functions, built-in OCaml functions, or functions from the `List` module to compute the values.)

2 Exercise TAPL 4.2.2. In order to do this exercise, first download `arith.tar.gz` from the homeworks page. This tarball can be unpacked using `tar zxvf arith.tar.gz`, and you should see a new directory called `arith` after doing this. You only need to change the definition of the function `eval` in the file `core.ml`; there is a `TODO` comment right before this function. The datatype that implements the syntax of the Arith language is called `term`; it is defined in `syntax.ml` and you may find it useful to refer to the definition when writing your `match` expression.

To compile the interpreter, simply run `make` in the `arith` directory. The final executable will be called `f`. If you're in the `arith` directory, you can run it by executing `./f some_file`. We have provided a file `test.f` with some simple examples.

For this exercise, please submit only `core.ml`; we do not need any other files from the interpreter.

3 Debriefing

1. How many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. Did everyone in your study group participate?
4. How deeply do you feel you understand the material it covers (0%–100%)?

If you have any other comments, we would like to hear them; please send them `cis500@cis.upenn.edu`.