CSE 341, Autumn 2010
Assignment #3 - ML: Curried Functions / Personality Test (100 points)
Due: Friday, October 22, 2010, 11:30 PM

This assignment focuses on higher-order functions, composition of functions, curried functions, defining operators, and anonymous functions in ML as described in Chapter 5 of the Ullman textbook. You are allowed to use any constructs from Chapters 1-5 of Ullman, subject to other restrictions on various parts of the assignment described on the following pages.

Place all of your functions and variables into two files called `hw3curry.sml` (Parts A-C) and `hw3ptest.sml` (Part D) and submit them from the class web page. If one of your files uses code defined in the other, it is fine for one file to load the other file with a `use` declaration.

In writing this problem set you will need a set of utility functions defined in a file called `utility.sml` available from the Homework section of the class web site. You will also need a file called `utility-ptest.sml` that has supporting data and functions for Part D, the personality test.

NOTE: In lecture, we wrote higher-order functions named `map`, `filter`, and `reduce`. Those functions appear in `utility.sml`, except that `map` and `filter` have been renamed to `mapx` and `filterx` so as not to conflict with with ML's built-in `map` and `filter` functions. You can still call `mapx`, `filterx`, and `reduce` to solve problems on this assignment except if a given part of the assignment forbids it.

If the function you implement to solve a given problem is useful for solving another later problem, you should call the earlier function from the later function to reduce redundancy.

Part A - Defining Operators:

Define the following operators as top-level symbols. You may use any of the ML syntax we have learned so far; you are not subject to the restrictions described in other parts of the assignment.

1. <<

```
val << = fn : int * int -> int
```

Define a "left-shift" operator << similar to the one seen in C and Java. The expression `a << b` should multiply the integer *a* times 2 to the *b* power. For example, `15 << 1` produces `30`, and `6 << 3` produces `48`, and `0 << 5` produces `0`. If the power *b* is negative, you should raise an exception of type `OutOfRange` (an that exception type you should define as a top-level symbol).

2. ~~

```
val ~~ = fn : 'a list * 'a list -> 'a list
```

Define a "list interleave" operator `~~` . The expression `list1 ~~ list2` should produce a new list whose elements come from `list1` and `list2` in alternating order. For example, `[1,2,3] ~~ [4,5,6]` should produce `[1,4,2,5,3,6]` . If the lists are not the same length, the remaining elements from the longer list should all be placed at the end of the result list. For example, `[8,6] ~~ [3,9,1,7]` should produce `[8,3,6,9,1,7]` and `[12,2,4,0] ~~ [5]` should produce `[12,5,2,4,0]`.

Part B - Higher-Order Functions:

The problems in this part of the assignment must be solved as one-line functions involving `mapx`, `filter`, and `reduce` and the infix operator `--` (all included in `utility.sml`). If you prefer, you may use ML's real versions of these functions: `map`, `List.filter`, and `List.foldl/foldr`. You are not allowed to use `if/else` expressions, `let` expressions, or pattern matching. You also should not need to use recursion in your definitions; that is, they do not need to call themselves. You should declare any helper functions as anonymous functions. You may use the standard functions `length` and `real` and all of the standard operators (`+`, `-`, `~`, `*`, `/`, `div`, `mod`, `@`, `::`, <, >, <=, >=, =, <>). You may also use the `op` keyword to convert a standard operator into a function.

3. `multiples`

```
val multiples = fn : int * int -> int list
```

Define a function `multiples` that takes two integers *k* and *n* as arguments and produces a list of the first *k* multiples of *n*. For example, the call of `multiples(5, 2)` should produce `[2,4,6,8,10]`.

4. `sumTo`

```
val sumTo = fn : int -> real
```

(You have already implemented this function in Homework 1. The point here is to re-implement it in a new way using higher-order functions and the other new material shown this week.)

Define a function `sumTo` that accepts an integer *n* and computes the sum of the first *n* reciprocals. That is:

$$\sum_{i=1}^{n} \frac{1}{i}$$

For example, `sumTo(3)` should produce (1 + 1/2 + 1/3) = 1.83333333.... You may assume that the function is not passed a value of *n* that is 0 or negative.

5. `countNegative`

```
val countNegative = fn : int list -> int
```

(You have already implemented this function in Homework 1. The point here is to re-implement it in a new way using higher-order functions and the other new material shown this week.)

Define a function `countNegative` that takes a list of integers as an argument and produces a count of the number of negative integers in the list. For example, `countNegative([3,17,~9,34,~7,2])` should produce `2`.

6. `allPairs`

```
val allPairs = fn : int * int -> (int * int) list
```

Define a function `allPairs` that takes two integers *m* and *n* as arguments and produces a list of all tuples whose first value is between 1 and *m* and whose second value is between 1 and *n*. The tuples should appear in nondecreasing order using the first value and within those groups, in increasing order by the second value. For example, `allPairs(3, 4)` should produce `[(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4),(3,1),(3,2),(3,3),(3,4)]`. You may assume that the values of both *m* and *n* will be positive integers.

Part C - Curried Functions:

These problems must be solved as one-line `val` declarations using curried functions and composition. This will allow you to practice partially instantiated functions. You are not allowed to use the `fun` keyword or anonymous function notation (`fn`). You also should not use `if-then-else` expressions, patterns, or matches. Instead define these functions with `val` declarations in terms of standard functions and standard operators (see the list above).

Use the `o` operator to compose functions. You are allowed to use ML's built-in curried functions `map`, `filter`, and `foldl`/`foldr`. But you should *not* use the "CSE 341" versions `mapx`, `filterx`, and `reduce` from `utility.sml`, because they are not in curried form. You may use other definitions from `utility.sml`, such as the operator `--` and the `curry` function that allows you to partially instantiate infix operators like `+`, `-`, `/`, `div`, `mod`, `--`, `<`, `>`, etc. For example:

```
val increment = curry op+ 1;
```

7. `timesTwoPlusThree`

```
val timesTwoPlusThree = fn : int -> int
```

Define a function `timesTwoPlusThree` that takes an integer argument *n* and produces 2*n* + 3. For example, the call of `timesTwoPlusThree 6` should produce `15`.

8. `positive`

```
val positive = fn : int -> bool
```

Define a function `positive` that takes an integer argument *n* and returns `true` if *n* is positive (strictly greater than 0) and `false` otherwise. For example, `positive 4` produces `true` and `positive ~7` produces `false`.

9. `evens`

```
val evens = fn : int -> int list
```

Define a function `evens` that takes an integer argument *n* and returns a list of the first *n* even numbers, starting with 2. For example, the call of `evens 7` produces `[2,4,6,8,10,12,14]`. You may assume that *n* is not negative.

10. `roots`

```
val roots = fn : int list -> int list
```

Define a function `roots` that takes a list of integers as an argument and produces a list of the square roots of the positive integers in the list, rounded to the nearest integer. Use ML's built-in functions such as `round`, `real`, and `Math.sqrt` to help you. Your function should ignore numbers that are not positive integers (i.e., should exclude them from the result). For example, the call of `roots [1,3,~8,14,0,7,~22,45]` should produce `[1,2,4,3,7]`.

11. `acronym`

```
val acronym = fn : string list -> string
```

Define a function `acronym` that takes a list of strings as an argument and produces a string composed of the uppercased first letters of each string in the list. Use ML's built-in functions such as `explode`, `implode`, `hd`, etc., along with the `Char.toUpper` function that returns the uppercase version of a character, to help you. For example, the call of `acronym ["what", "you", "see", "is", "what", "you", "get"]` should produce `"WYSIWYG"` and the call of `acronym ["there", "ain't", "no", "such", "thing", "as", "a", "free", "lunch"]` should produce `"TANSTAAFL"`. You may assume that there are no empty strings in the list.

12. `reverseStrings`

```
val reverseStrings = fn : string list -> string list
```

Define a function `reverseStrings` that takes a list of strings and produces the list obtained by reversing the order of the letters in each string and reversing the overall order of strings in the list. Use ML's built-in functions including the built-in function `rev` that reverses a list. For example, the call of `reverseStrings ["four", "score", "and", "seven", "years", "ago"]` should produce a result of `["oga","sraey","neves","dna","erocs","ruof"]`.

Part D - Personality Test (hw3ptest.sml):

This larger part involves working with a personality test described in detail below. Your code goes into `hw3ptest.sml`.

13. `answer`

```
(string * int list * string) list
```

The overall result you are computing for this part of the assignment is a `val` declaration to define a variable named `answer` that contains a list of triples. Each triple in your list should contain information for one person from the variable called **`data`** that is defined in `utility-ptest.sml`. Each triple should have the following three elements:

(*person's name*, *list of their four personality scores*, *personality type*)

This information constitutes a solution to the Keirsey personality test as described below. Obviously you will write many helper functions to be able to solve this task. You may use the full range of ML features, including everything we have studied so far and functions from the ML library (e.g., `Math`, `Int`, `String`, `Real`, `List`).

You can also use any functions in `utility.sml` and `utility-ptest.sml`. The second utility file contains a useful function named `quicksort`, which accepts two parameters: a "less-than" comparison function (a function that accepts two values and returns `true` if the first value comes before the second), and a list of values. For example, `quicksort(op<, [1,~5,17,2,3,~2,99,5])` produces `[~5,~2,1,2,3,5,17,99]`.

Sample output is available from the class web site. Your list must be **sorted by personality type and sub-sorted by name** (i.e., alphabetical within a given personality type). Your variable should be of the type shown above.

You can display the result by calling `displayAll(answer)`. The rest of this document describes the personality test.

The bulk of this assignment involves developing code that evaluates data obtained from the Keirsey personality test. The data will be in the form of a list of entries, one per person, where each entry is a tuple with a name (of type string) followed by a string of seventy characters (As, Bs, and dashes). You may assume all the As and Bs are in **uppercase**.

The Keirsey test measures four independent **dimensions** of personality:

- Extrovert versus Introvert (E vs I): what energizes you
- Sensation versus iNtuition (S vs N): what you focus on
- Thinking versus Feeling (T vs F): how you interpret what you focus on
- Judging versus Perceiving (J vs P): how you approach life

Individuals are categorized on one side or the other of each dimension. The corresponding letters are put together to form a personality type. For example, an extroverted, intuitive, thinking, perceiving person is referred to as an ENTP.

Remember that the Keirsey test involves 70 questions answered either A, B or - (a dash indicating no answer). You should compute a **score** in each of the four dimensions by computing the **percent of B** responses for that dimension. Some of the characters will be **dashes** to indicate that the user did not answer the question. These answers should not be included in the percent computation. The percentages should be **rounded** to the nearest integer.

These numbers are then converted into a **string** of letters. The A answers correspond to E, S, T, and J. The B answers correspond to I, N, F, and P. If a person is closer to the A side, use that letter. If they are closer to the B side, use that letter. If they are exactly in the middle, use "X".

The provided `utility-ptest.sml` has a function **`translate1`** that takes a string of 70 As, Bs or dashes, and returns a list of 4 lists of strings, one for each dimension of the personality test. The fact that this function is called `translate1` is a hint that you may want to define your own functions (`translate2`, `translate3`, etc) that perform other transformations on the data. You also have access to all of the utility functions (`quicksort`, `mapx`, `filterx`, `reduce`) and are encouraged to use them. You may also use ML's built-in curried versions of these functions if you prefer.

Your program should not depend on the specific number 70, although we assume that the answers always appear in a particular pattern that is repeated every 7 characters and we are assuming that the overall length is a multiple of 7. You also will write code specific to this personality test and its four dimensions. You may assume that it will be possible to compute a percentage for each dimension (i.e., that no user has all blank answers for any of the four dimensions).

For this part of the assignment, you are allowed to define helper functions and `val` declarations at the top level.

Extra Credit - Distance in 4-Space:

Another way to look at the personality data is to think of it as coordinates in 4-space. We can use these coordinates to compute distances between individuals. For our purposes, we will round the 4-dimensional distance to the nearest integer.

X. `bigAnswer`

```
val bigAnswer : (string * int * (string * int) list) list
```

(1 point) Define a variable `bigAnswer` that is a list of triples of the type shown above. The first element should be a person's name, the second element should be their average distance from other people (this can be the truncated average of the rounded integers), and the third element should be a list of distance tuples that have a person's name and their distance from the given individual. A person should not appear in their own list of distance tuples. The overall list should be sorted alphabetically by name. The individual distance tuple lists should be sorted by increasing distance from the individual. A sample output will be posted on the course web site.

Grading and Submission:

For reference, our solution to Parts A-C is 22 "substantive" lines long according to the class Indenter page, excluding blank lines and comments. Our solution to Part D is 24 "substantive" lines long, excluding blank lines and comments (and 46 "substantive" lines long with the extra credit). You don't need to match these numbers or even come close to them; they are just rough guidelines.

Your program should not produce any syntax errors when loaded into the ML interpreter. Your code also should not produce any warning messages, such as "non-exhaustive match." You may lose points if you name your functions or other top-level values incorrectly, even if their behavior is correct.

Your code should work for both basic and advanced cases. Make sure to perform your own testing, and always remember to test edge cases and bizarre values, such as negative numbers, empty lists, zeros, etc. unless the particular problem says you can assume that you won't encounter those.

As always, if the solution to a previous problem is useful in helping you solve a later problem, you should call the earlier function from the later function. You can include any testing code you develop (although it should be labeled as such).

You are expected to use good programming style, such as naming, indentation/spacing, and avoiding redundancy when possible. Avoid long lines of over 100 characters in length; if you have such a line, split it into multiple lines.

Favor the use of patterns rather than `if-then-else` to distinguish between base and recursive cases. Favor the `::` operator to grow lists versus the `@` operator (though the `@` operator is sometimes necessary and is not forbidden entirely).

Place a descriptive comment heading at the top of your program along with a comment header on each function, including any preconditions it assumes. Redundancy in general, such as recomputing a value unnecessarily or having unneeded recursive cases, should be avoided.