Problem Set 2: Folding and Datatypes

Due Thursday, February 16, 2012 at 23:59pm

Important notes about grading:

1. **Compile errors:** All programs you submit must compile. **Programs that do not compile will likely receive an automatic zero.** If you are having trouble getting your assignment to compile, please visit consulting hours. If you run out of time, it is better to comment out the parts that do not compile and hand in a file that compiles, rather than handing in a more complete file that does not compile.
2. **Missing functions:** We will be using an automatic grading script, so it is **crucial** that you name your functions and order their arguments according to the instructions, and that you place each function in the correct file.
3. **Imperative features:** You must **not** use any imperative features of OCaml (such as the `Hashtbl` module) to complete this assignment.
4. **Code style:** Please pay attention to style. Refer to the CS 3110 style guide and lecture notes. Ugly code that is functionally correct may still lose points. Take the extra time to examine each problem and find the most elegant solutions before coding them up.
5. **Wrong versions:** Please double-check that you are submitting the correct versions of your files to CMS. If you accidentally submit the wrong files, we will treat your submission as late.

Part 1: List Folding (30 points)

Folding is one of the most useful tools in functional programming. You will use folds many times the semester. The following exercises are designed to get you comfortable with folds that perform useful computations on lists. Note: All of these functions can be written without folding, but the point of the exercise is to get used to folding, so every function must include either `List.fold_left` or `List.fold_right`. Note that `List.fold_left` is tail recursive and `List.fold_right` is not. **For full credit, your solution must not contain the `rec` keyword.**

1. *(5 points)* Write a function `min2 : int list -> int` that returns the second-smallest element of a list, when put into sorted order. Note that if list contains duplicates, the second-smallest element and the smallest element may be identical; your code should return it.

   Example: `min2 [2110; 4820; 3110; 4120] = 3110`.

   For full credit, use a fold function and do not sort the list. If the list contains `0` or `1` elements, raise an exception (with a helpful message).

2. *(5 points)* Write a function `consec_dedupe : ('a -> 'a -> bool) ->'a list -> 'a list` that removes *consecutive* duplicate values from a list. More specifically,

the **consec_dedupe** function takes two arguments: a function **equiv** representing an equivalence relation, and a list **lst**. It returns a list containing the same elements as **lst**, but without any duplicates, where two elements are considered equal if applying **equiv** to them yields **true**.

Example: **consec_dedupe (=) [1; 1; 1; 3; 4; 4; 3] = [1; 3; 4; 3]**.

3. *(5 points)* Write a function **prefixes: 'a list -> ('a list) list** that returns a list of all non-empty prefixes of a list, ordered from shortest to longest.

   Example: **prefixes [1;2;3;4] = [[1]; [1;2]; [1;2;3]; [1;2;3;4]].**

4. *(7 points)* Write a function **k_sublist : int list -> int -> int list**. Given a list of integers **lst** and an integer **k**, the function **k_sublist** computes the contiguous sublist of length **k** whose elements have the largest sum.

   Example: **k_sublist [1; 2; 4; 5; 8] 3 = [4; 5; 8]**.

5. *(8 points)* The *powerset* of a set A is the set of all subsets of A. Write a function **powerset : 'a list -> 'a list list** such that **powerset s** returns the powerset of **s**, interpreting lists as unordered sets. You may assume there are no duplicate elements in **s**. More precisely, for every subset **T** of **s**, there is a list in **powerset s** that contains exactly the elements of **T**.

   Example: **powerset [1; 2; 3] = [[]; [1]; [2]; [3]; [1; 2]; [1; 3]; [2; 3]; [1; 2; 3]]**.

   The order of the sublists you return as well as the order of the elements in each sublist do not matter. Your program may return the elements in a different order from the example above.

Part 2: Stack Language (40 points)

In this part you will use higher-order functions to implement an interpreter for a simple stack language. The commands in this language are:

start	Initializes the stack to empty. This is always the first command in a program and never appears again in a program.
(push n)	Pushes the integer **n** onto the top of the stack. This command is always parenthesized with its argument.
pop	Removes the top element of the stack.
add	Pops the two top elements of the stack, computes their sum, and pushes the result back onto the stack.
mul	Pops the two top elements of the stack, computes their product, and pushes the result back onto the stack.

npop	Pops the top element of the stack **n**, then pops **n** more elements if **n** is positive.
dup	Pushes a duplicate copy of the top element onto the stack.
halt	Terminates the execution of the program. This is always the last command.

We can implement each of the commands as OCaml functions, in a way that lets us write a series of stack-language commands directly as OCaml expressions. The value of the OCaml expression should be the final stack configuration.
For example, the following series of commands: **`start (push 2) (push 3) (push 4) mul mul halt`** evaluates to a stack containing a single element, **`24`**.

1. *(25 points)* Your task is to implement the above commands in this language. To help you get started, the **`start`** and **`halt`** commands are already done for you in the starter code provided on CMS. You should represent the stack as a list, such that the previous example returns the stack [24]. Your code should fail with **`StackException`** if a command cannot be performed on the current stack. For example, the execution of the command add in the program **`start add halt`** should raise a **`StackException`**.
2. *(15 points)* Add support for storing and loading variables to your interpreter. The new interpreter should support the commands:

`(store "str")`	Stores the value current value on top of the stack in memory under the string **`"str"`**. This command is always parenthesized with its argument.
`(load "str")`	Loads the value associated with the string **`"str"`** in memory and puts it on top of the stack. This command is always parenthesized with its argument.

   To implement this, you will need to modify your above functions (including the provided stubs), as well as **`start`** and **`halt`**. For **`start`**, assume the memory is empty, and for **`halt`** you can throw away anything in memory (that is, just return the stack). If done correctly, programs without the new commands should behave identically, with no additional syntax required.
   If you have trouble with **`store`** and **`load`** working, but have the first part working correctly, submit the working version for the first part and a commented-out version of what you did for the second part. This way we can test your first part before you modified them for the second part, and then award partial credit for the second part.

Part 3: Search Engine (30 points)

In this problem, you will write a function that implements a very basic search engine, using term frequency-inverse document frequency weighting to rank the similarity between documents. Your function will take a search query **`q`** as input and a database of documents db and returns the identifier of the document most similar to the terms appearing in the query.

First, a little background. The *term frequency* (TF) of a term **`t`** in a document **`d`** is defined as the number of times that **`t`** appears in **`d`**. We will use TF to give more weight to documents that contain many occurrences of a given term mentioned in the search query.

The *inverse document frequency* (IDF) of a term **a** is defined as **N/df** where **N** is the total number of documents, and **df** is the number of documents that **a** occurs in. We will use IDF to reduce the weight of terms which are very common and appear in many documents (e.g., "the") and hence are of less use.

In addition, we will weight these basic statistics by a logarithmic factor. The *log-weighted TF* is $1+\log_{10}$**tf** if **tf** is greater than **0** and **0** otherwise, where **tf**is the statistic described above. Likewise, the *log-weighted IDF* is $\log_{10}$**N/df**.

To compute the similarity between a term and a document, we multiply the log-weighted TF of the term by its log-weighted IDF. To compute the similarity between the search query and a document, we sum up the similarities computed for each term in the query.

1. *(5 points)* Write a function **tf : string list -> string -> float** that takes in a document as a string list, and a term as a string, and returns the log-weighted TF as described above.

   Examples: **tf ["3110"; "is"; "fun"; "or"; "is"; "it"] "is" = 1.30102999566398125**

   tf ["3110"; "is"; "fun"; "or"; "is"; "it"] "fun" = 1.0

2. *(5 points)* Write a function **idf:string list list -> string -> float** that takes in the database (list of documents) and a term and returns the log-weighted IDF as described above.

   Examples: **idf [["Ocaml"; "is"; "fun"];["because";"fun";"is"; "a"]; ["keyword"]] "fun" = 0.176091259055681237**

   idf [["Ocaml"; "is"; "fun"];["because";"fun";"is"; "a"; "keyword"]] "fun" = 0.0

3. *(20 points)* Write a function **doc_rank: (int * string list) list -> string list -> int * float** that takes a database of documents, represented as a list of pairs of integers (representing document identifiers) and strings (representing document contents); a search query, represented as a list of strings; and returns the identifier of the document with the highest similarity, as well as its similarity.

   To test your solution we have provided you a function **load_documents** and several test databases. The **load_documents** function takes a database and returns an **(int*string list) list**. For the sake of consistency sake, if two or more documents have the same score, your code should return the first one in the database.

   Example: **doc_rank (load_documents "test1.txt") ["the"; "Bahia"] = (1, 1.28863187775142785)**