Assignment 4: ML Warmup

CS - 456 – Programming Languages

October 25, 2014

General Information

- Due Date: 11/03/2014 by midnight.
- Submission: See the instructions at the end of the document. Multiple submissions are possible before the deadline. Only the last one will be considered.
- All assignments are *individual*.
- Use Piazza for Q&A.

Programming Exercises

Solve the following exercises using ML Language. Your submission will be evaluated using the Moscow ML (mosml) implementation of SML. You are encouraged to use the Standard ML Basis Library.

Aside: Using mosml

Calling mosml as follows `mosml -P full` loads all the libraries that you might need from the standard library. The recommended usage is then:

```
rlwrap mosml -P full
```

The command `help "";` provides information within the REPL of `mosml`. `help "lib";` provide comprehensive information about the standar library. You can get more information about the standar library at `http://sml-family.org/Basis/`.

Exercises

1. Pattern Matching:

   a) Write a function `fib` that implements the fibonacci function. You cannot use `if` nor imperative features.

   b) Write a function `firstVowel` that takes a list of lower-case letters and returns true if the first character is a vowel (aeiou) and false if the first character is not a vowel or if the list is empty. Use the wildcard symbol `_` to match uninteresting patterns whenever possible, and avoid `if`. (The ML character syntax is `#"x"`.)

   c) Write the function `null`, which when applied to a list tells whether the list is empty. Avoid `if`, and make sure the function takes constant time. Make sure your function has the same type as the `null` in the Standard Basis.

2. Lists:

   a) `foldl` and `foldr` are predefined with type

      `('a * 'b -> 'b) -> 'b -> 'a list -> 'b`

      They are like the μScheme versions except the ML versions are Curried. Implement `rev` ("reverse") using `foldl` or `foldr`.

   b) Implement `minlist`, which returns the smallest element of a non-empty list of integers. Your solution can fail (e.g., by `raise Match`) if given an empty list of integers. Use `foldl` or `foldr`. Do not use recursion in any of your solutions.

   c) Implement `foldl` and `foldr` using recursion. Do not create unnecessary cons cells. Do not use if.

   d) Write the function

      `zip: 'a list * 'b list -> ('a * 'b) list`

      that takes a pair of lists (of equal length) and returns the equivalent list of pairs. If the lengths don't match, raise the exception `Mismatch`, which you will have to define.

   e) Define a function

      ```
      val pairfoldr : ('a * 'b * 'c -> 'c) -> 'c -> 'a list * 'b list -> 'c
      ```

      that applies a three-argument function to a pair of lists of equal length, using the same order as foldr. Use pairfoldr to implement zip.

   f) Define a function

      ```
      val unzip : ('a * 'b) list -> 'a list * 'b list
      ```

      that turns a list of pairs into a pair of lists. This one is tricky; here's a sample result:

      ```
      - unzip [(1, true), (3, false)];
      > val it = ([1, 3], [true, false]) : int list * bool list
      ```

      Hint: Try defining an auxiliary function, using `let`, that takes one or more accumulating parameters. You can use `rev`.

g) Define a function

```
val flatten : 'a list list -> 'a list
```

which takes a list of lists and produces a single list containing all the elements in the correct order. For example,

```
- flatten [[1], [2, 3, 4], [], [5, 6]];
> val it = [1, 2, 3, 4, 5, 6] : int list
```

3. Exceptions:

a) Write a (Curried) function

```
val nth : int -> 'a list -> 'a
```

to return the nth element of a list. (Starting from 0.) If nth is given arguments on which it is not defined, raise a suitable exception. You may define one or more suitable exceptions or you may choose to use an appropriate one from the initial basis. (If you have doubts about what's appropriate, be safe and define an exception of your own.) Do not use `List.nth`.

b) Environments. Define a type `'a env` and functions

```
type 'a env = (* you fill in this part *)
exception NotFound of string
val emptyEnv : 'a env = (* ... *)
val bindVar : string * 'a * 'a env -> 'a env = (* ... *)
val lookup  : string * 'a env -> 'a = (* ... *)
```

i. Define `bindVar`, `emptyEnv`, `env`, `lookup`, `NotFound`, such that you can use `'a env` for a type environment or a value environment. On an attempt to look up an identifier that doesn't exist, raise the exception `NotFound`.

ii. Do the same, except make type `'a env = string -> 'a`, and let `fun lookup (name, rho) = rho name`.

iii. Write a function `val isBound : string * 'a env -> bool` that works with both representations of environments. That is, write a single function that works regardless of whether environments are implemented as lists or as functions. You will need imperative features, like sequencing (the semicolon). Don't use `if`.

iv. Write a function

```
val extendEnv : string list * 'a list * 'a env -> 'a env
```

that takes a list of variables and a list of values and adds the corresponding bindings to an environment. It should work with both representations. Do not use recursion. Hint: you can do it in two lines using the higher-order list functions defined above.

4. Algebraic Data Types:

   a) Search trees. ML can easily represent binary trees containing arbitrary values in the nodes:

   ```
   datatype 'a tree = NODE of 'a tree * 'a * 'a tree
                    | LEAF
   ```

   Defines tree. To make a search tree, we need to compare values at nodes. The standard idiom for comparison is to define a function that returns a value of type order. Order is predefined by

   ```
   datatype order = LESS | EQUAL | GREATER     (* do not include *)
   ```

   Defines order. Because order is predefined, if you include it in your program, you will hide the predefined version (which is in the initial basis) and other things may break mysteriously. So don't include it.

   We can use the order type to define a higher-order insertion function by, e.g.,

   ```
   fun insert cmp =
       let fun ins (x, LEAF) = NODE (LEAF, x, LEAF)
             | ins (x, NODE (left, y, right)) =
                 (case cmp (x, y)
                   of LESS    => NODE (ins (x, left), y, right)
                    | GREATER => NODE (left, y, ins (x, right))
                    | EQUAL   => NODE (left, x, right))
       in  ins
       end
   ```

   Defines insert. This higher-order insertion function accepts a comparison function as argument, then returns an insertion function. (The parentheses around case aren't actually necessary here, but I've included them because if you leave them out when they are needed, you will be very confused by the resulting error messages.)

   We can use this idea to implement polymorphic sets in which we store the comparison function in the set itself. For example,

   ```
   datatype 'a set = SET of ('a * 'a -> order) * 'a tree
   fun nullset cmp = SET (cmp, LEAF)
   ```

   Defines nullset, set. Write a function
   `val addelt : 'a * 'a set -> 'a set`
   that adds an element to a set.

      i. Write a function
      `val treeFoldr : ('a * 'b -> 'b) -> 'b -> 'a tree -> 'b`
      that folds a function over every element of a tree, rightmost element first. Calling `treeFoldr op :: [] t` should return the elements of t in order.

ii. Write a similar function

   `val setFold : ('a * 'b -> 'b) -> 'b -> 'a set -> 'b`

   The function setFold should visit every element of the set exactly once, in an unspecified order.

b) Sequences with constant-time append. Consider the following informal definition of a sequence, by cases: A sequence is either empty, or it is a singleton sequence containing one element x, or it is one sequence followed by another sequence.

   i. Define an algebraic datatype `'a seq` that corresponds to this definition of sequences. Your definition should contain one value constructor for each alternative.

   ii. Define a function

   `scons : 'a * 'a seq -> 'a seq`

   that adds a single element to the front of a sequence, using constant time and space.

   iii. Define a function

   `ssnoc : 'a * 'a seq -> 'a seq`

   that adds a single element to the back of a sequence, using constant time and space. Note: The order of arguments is the opposite of the order in which the results go in the data structure. That is, in `ssnoc (x, xs)`, `x` follows `xs`. (The arguments are the way they are so that ssnoc can be used with `foldl` and `foldr`.)

   iv. Define a function

   `sappend : 'a seq * 'a seq -> 'a seq`

   that appends two sequences, using constant time and space.

   v. Define a function

   `listOfSeq : 'a seq -> 'a list`

   that converts a sequence into a list containing the same elements in the same order. Your function must allocate only as much space as is needed to hold the result.

   vi. Without using explicit recursion, define a function

   `seqOfList : 'a list -> 'a seq`

   that converts an ordinary list to a sequence containing the same elements in the same order.

5. Immutable, persistent alternative to linked lists

   Define your own representation of a new abstraction: the list with finger. A list with finger is a nonempty sequence of values, together with a “finger” that points at one position in the sequence. The abstraction provides constant-time insertion and deletion at the finger. For this problem you will have to create your own data structure. Finding a good representation is half of the problem. Once you have a good representation, the code is easy: over half the functions can be implemented in one line each, and no function requires more than two lines of code.

a) Define a representation for type `'a flist`. (Before you can define a representation, you will want to study the rest of the parts of this problem, plus the test cases.) Document your representation by saying, in a short comment, what sequence is meant by any value of type `'a flist`.

b) Define function

   ```
   val singletonOf : 'a -> 'a flist
   ```

   which returns a sequence containing a single value, whose finger points at that value.

c) Define function

   ```
   val atFinger : 'a flist -> 'a
   ```

   which returns the value that the finger points at.

d) Define functions

   ```
   val fingerLeft  : 'a flist -> 'a flist
   val fingerRight : 'a flist -> 'a flist
   ```

   Calling `fingerLeft xs` creates a new list that is like `xs`, except the finger is moved one position to the left. If the finger belonging to `xs` already points to the leftmost position, then `fingerLeft xs` should raise the same exception that the Basis Library raises for array access out of bounds. Function `fingerRight` is similar. Both functions must run in constant time and space. Please think of these functions as "moving the finger", but remember no mutation is involved. Instead of changing an existing list, each function creates a new list.

e) Define functions

   ```
   val deleteLeft  : 'a flist -> 'a flist
   val deleteRight : 'a flist -> 'a flist
   ```

   Calling `deleteLeft xs` creates a new list that is like `xs`, except the value `x` to the left of the finger has been removed. If the finger points to the leftmost position, then `deleteLeft` should raise the same exception that the Basis Library raises for array access out of bounds. Function `deleteRight` is similar. Both functions must run in constant time and space. As before, no mutation is involved.

f) Define functions

   ```
   val insertLeft  : 'a * 'a flist -> 'a flist
   val insertRight : 'a * 'a flist -> 'a flist
   ```

   Calling `insertLeft (x, xs)` creates a new list that is like `xs`, except the value `x` is inserted to the left of the finger. Function `insertRight` is similar. Both functions must run in constant time and space. As before, no mutation is involved. (These functions are related to ”cons”.)

g) Define functions

```
val ffoldl : ('a * 'b -> 'b) -> 'b -> 'a flist -> 'b
val ffoldr : ('a * 'b -> 'b) -> 'b -> 'a flist -> 'b
```

which do the same thing as `foldl` and `foldr`, but ignore the position of the finger. Here is a simple test case, which should produce a list containing the numbers 1 through 5 in order. You can use `ffoldr` to confirm.

```
  val test = singletonOf 3
  val test = insertLeft  (1, test)
  val test = insertLeft  (2, test)
  val test = insertRight (4, test)
  val test = fingerRight test
  val test = insertRight (5, test)
```

You'll want to test the delete functions as well.

General Evaluation Criteria

The evaluation criteria is similar to previous assignments. Write clear code, you can use the following style guide for instance `http://www.cs.cornell.edu/courses/cs312/2005sp/handouts/style.htm`. Test your own implementation. Test cases are not provided. You can provide your test cases in your submission.

Submission

- Submit a single file with the following naming structure {lastname1}-{lastname2}-hw4.sml
- **Important:** Respect the naming of the definitions that you are required to provide. If you change the names of the functions, they will not be evaluated by the grader scripts.