COMP 105 Homework: Core ML

Due Thursday, March 3, at 11:59 PM.

The purpose of this assignment is to help you get acclimated to programming in ML, which you will do by writing many small exercises. By the time you complete this assignment, you will be ready to tackle serious programming tasks in core ML. It will help you to study Chapter 5 carefully and to compare the ML code in Chapter 5 with the C code in Chapter 3.

You will be more effective if you are aware of the Standard ML Basis Library. Jeff Ullman's text (Chapter 9) describes the 1997 basis, but today's compilers use the 2004 basis, which is a standard. You will find a few differences in I/O, arrays, and elsewhere; the most salient difference is in `TextIO.inputLine`.

The most convenient guide to the basis is the Moscow ML help system; type

```
- help "";
```

at the `mosml` interactive prompt. The help file is badged incorrectly, but as far as I know, it is up to date.

Guidelines

For all the problems in this homework, **use function definition by pattern matching**. In particular, do not use the functions `null`, `hd`, and `tl`; use patterns instead. Some useful list patterns include these patterns, to match lists of *exactly* 0, 1, 2, or 3 elements:

```
<patterns>= [D->]
[]
[x]
[x, y]
[a, b, c]
```

and also these patterns, which match lists of *at least* 0, 1, 2, or 3 elements:

```
<patterns>+= [<-D]
l
x::xs
x1::x2::xs
a::b::c::l
```

When using these patterns, remember that **function application has higher precedence than any infix operator!** This is as true in patterns as it is anywhere else.

Do not define auxiliary functions at top level. Use `local` or `let`. *Do not use `open`*; if needed, use one-letter abbreviations for common structures. **Do not use any imperative features unless the problem explicitly says it is OK**.

Feel free to use the standard basis extensively. Moscow ML's `help "lib";` will tell you all about the library. And if you use

```
ledit mosml -P full
```

as your interactive top-level loop, it will automatically load almost everything you might want from the standard basis.

All the sample code we show you is gathered in one place online.

As you learn ML, this table may help you transfer your knowledge of µScheme:

µScheme	ML

`val`	`val`
`define`	`fun`
`lambda`	`fn`

Put all your solutions in one file: `warmup.sml`. (If separate files are easier, combine them with `cat`.) At the start of each problem, please label it with a short comment, like

```
(***** Problem A *****)
```

To receive credit, your `warmup.sml` file must compile and execute in the Moscow ML system. For example, we must be able to compile your code *without warnings or errors*:

```
% /usr/sup/bin/mosmlc -c warmup.sml
%
```

Please remember to **put your name and the time you spent in the `warmup.sml` file**.

The homework problems

Solve the following problems:

Higher-order programming

A. Here's a function that is somewhat like `fold`, but it works on binary operators.

1. Define a function

   ```
   compound : ('a * 'a -> 'a) -> int -> 'a -> 'a
   ```

   that ``compounds'' a binary operator `rator` so that `compound rator n x` is `x` if `n=0`, `x rator x` if `n = 1`, and in general `x rator (x rator (... rator x))` where `rator` is applied exactly `n` times. `compound rator` need not behave well when applied to negative integers.
2. When `rator` is associative, it is not necessary to apply it so many times. Define a function `acompound` that has the same type as `compound`, and for an associative `rator` computes the same results, but such that `acompound rator n x` requires only O(log n) applications of `rator` to compute.
3. Use the `acompound` function to define a function for integer exponentiation

   ```
   exp : int -> int -> int
   ```

   so that, for example, `exp 3 2` evaluates to 9. *Hint: take note of the description of `op` in Ullman S5.4.4, page 165.*

Don't get confused by infix vs prefix operators. Remember this:

- Fixity is a property of an identifier, not of a value.
- If `<$>` is an infix identifier, then `x <$> y` is syntactic sugar for `<$>` applied to a pair containing `x` and `y`, which can also be written as `op <$> (x, y)`.

Patterns

B. Consider the pattern `(x::y::zs, w)`. For each of the following expressions, tell whether the pattern matches the value denoted. If the pattern matches, say what values are bound to the four variables `x`, `y`, `zs`, and `w`. If it does not match, explain why not.

1. `([1, 2, 3], ("COMP", 105))`
2. `(("COMP", 105), [1, 2, 3])`
3. `([("COMP", 105)], (1, 2, 3))`
4. `(["COMP", "105"], true)`

5. `([true, false], 2.718281828)`

(Put your answers into `warmup.sml` as comments.)

C. Using patterns, write a recursive Fibonacci function that does not use `if`.

D. Write a function 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 _ whenever possible, and avoid `if`. *Remember that the ML character syntax is `#"x"`, as decribed in Ullman, page 13.*

E. 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.

Lists

F. `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.

1. Implement `length` using `foldl` or `foldr`.
2. Implement `rev` using `foldl` or `foldr`.
3. Implement `minlist`, which returns the smallest element of a non-empty list of integers. Your solution should work regardless of the representation of integers (e.g., it should not matter how many bits are used to represent 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.

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

H. Implement queues using no side effects.

1. For a first cut, try the following representation:

```
<sample>= [D->]
exception Empty
type 'a queue = 'a list
val put : 'a queue * 'a -> 'a queue
val get : 'a queue -> 'a * 'a queue
```

Defines `Empty`, `get`, `put`, `queue` (links are to index).

Implement `put` and `get`. `get` should raise the exception `Empty` if the queue is empty.

2. The representation shown above is unpleasant in that either `put` or `get` must take O(n) time. Using *a pair of lists* to represent a queue, implement `put` and `get` that take constant ``amortized'' time. (That is, a combination of N puts and gets, in any reasonable order, can be expected to take O(N) time total, instead of possibly O(N-squared) as above.) *Hint: think about the tricks we used in class to come up with a cheap list-reversal function.*

Bonus! If you have taken COMP 40 during a Fall semester, give the invariant that relates the pair of lists to a sequence in the world of ideas. You may choose to do this by defining a function `contents` that maps the pair of list into a single sequence which is enqueued at the end and dequeued from the beginning.

Bonus! If you have taken COMP 160 during a Fall or Spring semester, give a brief *amortized* analysis of the costs associated with this representation of queues.

I. Write a 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.

J. Define a function

```
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`.

K. Define a function `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:

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

Hint: Try defining an auxiliary function that uses the method of accumulating parameters, and be prepared to use `rev`.

L. Define a function `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,

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

To get full credit for this problem, your function should use no unnecessary cons cells.

Strings

For this section it may help you to be aware of the built-in functions `implode`, `explode`, and `size`.

M. The goal of this problem is to help you explore why the Standard ML Basis Library has a built-in function `concat`. The problem has four parts:

1. *Without* using `concat`, define a function `sflatten : string list -> string`, which takes a list of strings and produces a single string containing all the original strings concatenated in the correct order. *Make your function as simple as you can.*
2. Explain how much space is used by `sflatten`. Try do to an exact calculation, but if you can't, you can fall back on Big-*O* notation.
3. Still without using `concat`, define a version of `sflatten` that uses as little space as possible. How does its space usage compare with that of your original definition?
4. A built-in, primitive function like `concat` can be written in C or in assembly language, and so can do things that an ML function cannot. It it possible that the primitive `concat` uses even less space than your version from part 3? If so, how much less? Is it worth having `concat`? Why?

Exceptions

N. Write a (Curried) function `nth : int -> 'a list -> 'a` to return the *n*th element of a list. (Number elements 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, play it safe and define an exception of your own.)

I expect you to implement `nth` yourself and not simply call `List.nth`.

O. Environments

1. Define a type `'a env` and functions

<sample>+= [<-D->]
```
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 = (* ... *)
```

Defines `bindVar`, `emptyEnv`, `env`, `lookup`, `NotFound` (links are to index).

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`. Don't worry about efficiency.

2. Do the same, except make `type 'a env = string -> 'a`, and let

<sample>+= [<-D->]

```
fun lookup (name, rho) = rho name
```

Defines `lookup` (links are to index).

3. Write a function `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`.
4. Write a function `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.*

Discriminated unions (`datatype`)

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

<sample>+= [<-D->]

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

Defines `tree` (links are to index).

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`. As discussed in Ullman, page 325, `order` is *predefined* by

<sample>+= [<-D->]

```
datatype order = LESS | EQUAL | GREATER
```

Defines `order` (links are to index).

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

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

<sample>+= [<-D->]

```
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` (links are to index).

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,

```
<sample>+= [<-D]
datatype 'a set = SET of ('a * 'a -> order) * 'a tree
fun nullset cmp = SET (cmp, LEAF)
```

Defines `nullset`, `set` (links are to index).

- Write a function `addelt` of type `'a * 'a set -> 'a set` that adds an element to a set.
- Write a function `treeFoldr` of type `('a * 'b -> 'b) -> 'b -> 'a tree -> 'b` that folds a function over every element of a tree, rightmost element first. `treeFoldr op :: [] t` should return the elements of `t` in order. Write a similar function `setFold` of type `('a * 'b -> 'b) -> 'b -> 'a set -> 'b`.

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

Extra credit

There are two extra-credit problems: **FIVES** and **VARARGS**.

FIVES

Recall the following problem from the Scheme homework:

> *Consider the class of well-formed arithmetic computations using the numeral 5. These are expressions formed by taking the integer literal 5, the four arithmetic operators +, -, *, and /, and properly placed parentheses. Such expressions correspond to binary trees in which the internal nodes are operators and every leaf is a 5. Write a µScheme program to answer one or more of the following questions:*
>
> - *What is the smallest positive integer than cannot be computed by an expression involving exactly five 5's?*
> - *What is the largest prime number that can computed by an expression involving exactly five 5's?*
> - *Exhibit an expression that evaluates to that prime number.*

Write an ML function `reachable` of type

```
('a * 'a -> order) * ('a * 'a -> 'a) list -> 'a -> int -> 'a set
```

such that `reachable (Int.compare, [op +, op -, op *, op div]) 5 5` computes the set of all integers computable using the given operators and exactly five 5's. (You don't have to bother giving the answers to the questions above, since they're easy to get with `setFold`.) My solution is under 20 lines of code, but it makes heavy use of the `setFold`, `nullset`, `addelt`, and `pairfoldr` functions defined earlier.

Hints:

- In order to be able to use `Int.compare`, you will either have to run `mosml -P full` or else tell Moscow ML interactively to `load "Int";`
- Begin your function definition this way:

  ```
  fun reachable (cmp, operators) five n =
    (* produce set of expressions reachable with exactly n fives *)
  ```
- Use dynamic programming.
- Create a list of length *k-1* in which element *i* is a set containing all the integers that can be computed using exactly *i* elements. Now compute the *k*th element of the list by combining 1 with *k-1*, 2 with *k-2*, etcetera.
- Try doing the above by passing a list and its reverse, then use `pairfoldr` with a suitable function.

- The initial list contains a set with exactly one element (in the example above, 5).
- Make sure your solution has the completely general type given above, so you could use it with different operations and with different representations of numbers.

VARARGS

Extend μScheme to support procedures with a variable number of arguments. Do so by giving the name `...` (three dots) special significance when it appears as the last formal parameter in a lambda. For example:

```
-> (val f (lambda (x y ...)) (+ x (+ x (foldl + 0 ...))))
-> (f 1 2 3 4 5) ; inside f, rho = { x |-> 1, y |->, ... |-> '(3 4 5) }
15
```

In this example, it is an error for `f` to get fewer than two arguments. If `f` gets at least two arguments, any additional arguments are placed into an ordinary list, and the list is used to initialize the location of the formal parameteter associated with `...`.

1. Implement this new feature inside of `mlscheme.sml`. I recommend that you begin by changing the definition of `lambda` on page 187 to

   ```
   and lambda = name list * { varargs : bool } * exp
   ```

   The type system will tell you what other code you have to change. For the parser, you may find the following function useful:

   ```
   fun newLambda (formals, body) =
      case rev formals
        of "..." :: fs' => LAMBDA (rev fs', {varargs=true},  body)
         | _            => LAMBDA (formals, {varargs=false}, body)
   ```

   The type of this function is

   ```
   name list * exp -> name list * {varargs : bool} * exp;
   ```

   thus it is designed exactly for you to adapt old syntax to new syntax; you just drop it into the parser wherever `LAMBDA` was used.
2. As a complement to the varargs lambda, write a new `call` primitive such that

   ```
   (call f '(1 2 3))
   ```

   is equivalent to

   ```
   (f 1 2 3)
   ```

   Sadly, you won't be able to use `PRIMITIVE` for this; you'll have to invent a new kind of thing that has access to the internal `eval`.
3. Demonstrate these utilities by writing a higher-order function `cons-logger` that counts cons calls in a private variable. It should operate as follows:

   ```
   -> (val cl (cons-logger))
   -> (val log-cons (car cl))
   -> (val conses-logged (cdr cl))
   -> (conses-logged)
   0
   -> (log-cons f e1 e2 ... en) ; returns (f e1 e2 ... en), incrementing
                                ; private counter whenever cons is called
   -> (conses-logged)
   99  ; or whatever else is the number of times cons is called
       ; during the call to log-cons
   ```

4. Rewrite the APPLY-CLOSURE rule to account for the new abstract syntax and behavior. To help you, simplified LaTeX for the original rule is online.

Avoid common mistakes

Here is a list of common mistakes to avoid:

- If you **redefine a type** that is already in the initial basis, code will fail in very mysterious ways.
- If you redefine a function at the top-level loop, this is fine, **unless that function captures one of your own functions in its closure**. Example:

  ```
  fun f x = ... stuff that is broken ...
  fun g (y, z) = ... stuff that uses 'f' ...
  fun f x = ... new, correct version of 'f' ...
  ```

  You now have a situation where **`g` is broken, and the resulting error is *very* hard to detect**. Stay out of this situation; instead, **load fresh definitions from a file using the `use` function**.
- **Never put a semicolon after a definition**. I don't care if Jeff Ullman does it, but don't you do it—it's wrong! You should have a semicolon only if you are deliberately using imperative features.
- It's a common mistake to become very confused by **not knowing where you need to use `op`**. Ullman covers `op` in Section 5.4.4, page 165.
- It's a common mistake to **include redundant parentheses in your code**. To avoid this mistake, follow the directions in the course supplement to Ullman.

What to submit

Submit the files `warmup.sml`, and optionally `varargs.sml` or `fives.sml`, using the script `submit105-ml`. In comments at the top of your `warmup.sml` file, please include your name, the names of any collaborators, and the number of hours you spent on the assignment.

Index and cross-reference