CSE341 Spring '06 Due Wednesday, April 19
Assignment 2

For this homework your solutions must use pattern-matching. Don't use the functions `hd`, `tl`, `null` or anything containing the `#` character. Similarly, don't use if-then-else in places where pattern matching will suffice (although there are a small number of places where you will need if-then-else). Don't include types in function declarations unless necessary; ML's type inference should suffice in nearly all cases.

1. In the following we will represent sets as lists.

   (a) Consider the following.

   ```
   infix mem
   fun x mem [] = false
     | x mem (y::ys) = x=y orelse x mem ys
   fun newmem(x,xs) = if (x mem xs) then xs else x::xs
   ```

   Type these up and include them in your turn-in. In a comment near these functions, answer the following three questions. What is the result of `newmem(2,[1,2])`? Of `newmem("apple" ,["orange","banana"])`? Describe in your own words what these functions do, using only one sentence for each function.

   (b) Write a function `setof` that takes a list of items, possibly with duplicates, and returns a list with all the duplicates removed. For example, `setof [1,2,3,2] =>[1,2,3]`. Hint: use `newmem`. The order of items in the output list doesn't matter.

   (c) Write an infix function `union` that computes the union of two lists of items when viewed as sets. `[1,2,3] union [2,3,4]` evaluates to `[1,2,3,4]` (perhaps in a different order).

   (d) Write an infix function `isect` that computes the intersection of two lists of items when viewed as sets, evaluating `[1,2,3] isect [2,3,4]` as `[2,3]` (or `[3,2]`).

   (e) What is the main advantage of not declaring the argument types of these functions? Include the answer as a comment after your definition of `isect`.

   (f) Is your `isect` function tail-recursive? Explain why or why not in another comment. Then write an alternate version of it named `isect_alt` that is tail recursive if `isect` is not, or *vice versa*. ("Accumulator style" might be useful for one or the other of them.) In another comment, give an example (executable code) where the output order differs between the two functions, or explain why they are always the same.

2. In the rest of this assignment we'll look at Boolean expressions, such as:

   ```
   true And (false Or Not false)
   Not x And (true Or x)
   ```

   Because the first expression contains only the constants `true` and `false`, we say it is an expression over constants. It *evaluates* to `true`. The second expression contains a variable `x`; we have to *bind* a value to `x` before we can evaluate the expression. We say that `x` is *free* in the expression. We can *bind* free variables to constants, for example binding `x` to `true` in the expression `(Not x And (true Or x))` gives `Not true And (true Or true)`, which evaluates to `false`. If we bound `x` to `false`, the expression would evaluate to `true`.

   We say that an expression is *constant* if it contains no free variables. A constant expression can be evaluated without having to bind anything.

To express a Boolean expressions in ML, we will use the following data type.

```
datatype 'a expr =
         Const of bool
       | Var of 'a
       | Not of 'a expr
       | And of 'a expr * 'a expr
       | Or of 'a expr * 'a expr
exception UnboundVar
```

For this assignment, we'll use only `string expr`. Thus, the ML expression that's the same as the second example above is `And(Not(Var "x"),Or(Const true, Var "x"))`. (But in your next homework we'll generalize this to Boolean expressions over objects other than strings.)

(a) Write a function `free_vars` that takes an expression and returns a `string list` of any free variables. Variables should not be repeated even if they appear multiple times; use the set functions above.

(b) Write a function `eval` that takes a constant expression and returns its value. If `eval` is passed a function with free variables, it should raise an `UnboundVar` exception.

Note: a natural way to solve this problem will evaluate `Or(Const true,Var "x")` as `true` even though it's not really a constant expression. That's okay. The same solution will raise an exception if passed `Or(Var "x",Const true)`, even though it's the same expression as before. That's okay too. The point of this problem is to get practice in evaluating things, not figuring out the type of an expression.

(c) Write a function `bind1` that takes a variable, a Boolean value and an expression, and returns an expression with all instances of the variable bound to the truth value. It should be a curried function. For example,

```
bind1 "x" true (And(Var "x",Or(Var "y", Var "x")))
             --> And(Const true,Or(Var "y", Const true))
bind1 "x" true (And(Var "z",Or(Var "y", Var "z")))
             --> And(Var "z",Or(Var "y", Var "z"))
```

(d) `bind1` is nice, but not very general. For example, we couldn't use it to change some variable in an expression to a different variable or sub-expression. Write a function `bind` that takes a function `binder` and an expression `e`, applies `binder` to the argument of each `Var` in `e`, and leaves the rest of the expression unchanged. `binder` should take the value from a `Var` and return an expression.

Then write functions `bindvar` and `changevar` using `bind` so that `bindvar` is equivalent to `bind1`, and `changevar` is similar except that variables are changed. For example,

```
bindvar "x" true (And(Var "x",Or(Var "y", Var "x")))
             --> And(Const true,Or(Var "y", Const true))
changevar "x" "help" (And(Var "x",Or(Var "y", Var "x")))
             --> (And(Var "help",Or(Var "y", Var "help")))
```

(e) Write a function `satisfying_assignments` that takes an expression and returns a `(string*bool) list list` of all satisfying assignments, if any exist. For example,

```
satisfying_assignments (And(Not(Var "x"),Or(Const true, Var "x")))
            --> [[("x",false)]]
```

```
satisfying_assignments (And(Not(Var "x"),Or(Const false, Var "x")))
            --> nil
satisfying_assignments (Or(Var "x", Var "y"))
            --> [[("x",true),("y",true)],[("x",true),("y",false)],
                 [("x",false),("y",true)]]
satisfying_assignments (Const true)
            --> [[]]
satisfying_assignments (Const false)
            --> []
```

Hint: write a local function that takes an expression and a list of its free variables. If the list is empty, evaluate the expression; otherwise recurse, using `map` and `@` (append) to build the new bindings lists. My implementation of this function is 11 lines.

There's not that much code to write—under 100 lines in my solution—but they require careful thought. Start early. Don't just settle for your first answer. Revise, experiment, play with it.

Extra Credit

Extend your solution to handle the *quantifiers* `Exist` and `All`. E.g.,
`Exist("x",And(Not(Var "x"),Or(Const true, Var "x")))` evaluates to true, whereas
`All("y",Exist("x",And(Not(Var "x"),Or(Var "y", Var "x"))))` evaluates to false.

Type Bindings

Your solution should generate the following bindings (or their synonyms).

```
datatype 'a expr
  = And of 'a expr * 'a expr
  | Const of bool
  | Not of 'a expr
  | Or of 'a expr * 'a expr
  | Var of 'a
exception UnboundVar
infix mem union isect
val mem = fn : ''a * ''a list -> bool
val newmem = fn : ''a * ''a list -> ''a list
val setof = fn : ''a list -> ''a list
val union = fn : ''a list * ''a list -> ''a list
val isect = fn : ''a list * ''a list -> ''a list
val free_vars = fn : ''a expr -> ''a list
val eval = fn : 'a expr -> bool
val bind1 = fn : ''a -> bool -> ''a expr -> ''a expr
val bind = fn : ('a -> 'b expr) -> 'a expr -> 'b expr
val bindvar = fn : ''a -> bool -> ''a expr -> ''a expr
val changevar = fn : ''a -> ''a -> ''a expr -> ''a expr
val satisfying_assignments = fn : ''a expr -> (''a * bool) list list
```