CSE341 Spring '07 Due Friday, April 27

Assignment 2

For this homework your solutions must use pattern-matching. Avoid 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. Use curried functions in preference to tuple parameters unless there's a good reason not to.

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

   (a) Consider the following.

   ```
   Control.Print.printDepth := 99; (* deep structs for debugging. yes, := *)
   infix mem
   fun x mem [] = false
     | x mem (y::ys) = x=y orelse x mem ys
   fun addmem x xs = if (x mem xs) then xs else x::xs
   ```

   Type or copy these and include them in your turn-in. ("Infix" is explained in section 9.1.4.) In a comment near these functions, answer the following.

   What is the result of `addmem 2 [1,2]`?

   Of `addmem "apple" ["orange","banana"]`?

   Of `addmem "apple"`?

   Describe in your own words what these functions do, using only one sentence for each function.

   (b) Write a function `remmem` that (if necessary) removes its first argument from the list given by its second argument. E.g., `remmem "doubt" ["no", "doubt"]` should return `["no"]`.

   (c) 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 `addmem`. The order of items in the output list doesn't matter.

   (d) 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).

   (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 `union`.

   (f) Write a (non-infix) function `isect_1` that computes the intersection of two lists of items when viewed as sets. E.g., `isect_1([1,2,3], [2,3,4]) --> [2,3]` (or `[3,2]`).

   (g) Is your `isect_1` function tail-recursive? Explain why or why not in another comment. Then write an alternate version of it named `isect_2` that is tail recursive if `isect_1` 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.

   (h) Use `foldl` or `foldr` to write `isect_folded`, a (non-recursive) 3rd version of set intersection.

   (i) Use `List.filter` to write `isect_filtered`, a 4th version of intersection, also non-recursive.

(j) Bind your favorite of the above 4 versions of intersection to the name `isect`. In a nearby comment say why you think this is the "best" of the 4 ways to write it, in terms of simplicity and clarity of the code. (A few sentences, at most.)

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)
```

For extra precision, I'm going to rewrite this in a more (but not fully) ML-ish syntax as:

```
And(T,Or(F,Not(F)))
And(Not(x),Or(T,x))
```

Because the first expression contains only the constants `true/T` and `false/F`, 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 `And(Not(x),Or(T,x))` gives `And(Not(T),Or(T,T))`, which evaluates to `false`. If we bound `x` to `false`, the expression would evaluate to `true`.

For many applications it is useful to introduce bindings directly into the Boolean expressions in the form of "universal" (∀, "for all") and "existential" (∃, "there exist") *quantifiers*. For example, the quantified Boolean expression below expresses the (true) assertion that for every possible binding of `x` to a Boolean constant, either `x` or its negation must be true.

```
All(x,Or(x,Not(x)))
```

With quantified Boolean formulas, it's important to understand the notions of *free* versus *bound* variables and *scope of quantifiers*. Specifically, the first parameter of `All` and `Exist` give the variable being bound, and the scope of that binding is the entirety of its second parameter, *excluding* the scope of any nested quantifier using the same variable name. For example, in the following:

```
And(x,All(x,Or(x,Exist(y,Or(x,Or(y,Exist(x,Or(Not(x),z))))))))
```

the `z` and the first `x` are free variables, the middle two `x`'s are bound by the `All x` quantifier, and the last `x` is bound by the `Exist x` quantifier. Just as with scope of variable names in most programming languages, the inner `Exist x` doesn't conflict with the outer `All x`, it just supercedes it throughout the inner scope. E.g., if we replace `Exist(x,Or(Not(x),z))` by `Exist(w,Or(Not(w),z))`, the meaning is the same.

We say that an expression is *constant* if it contains no free variables. A constant expression can be evaluated without having to bind anything (other than the bindings implicit in any quantifiers within the formula).

As with ML itself, we can speak of an *environment* as a set of bindings, or equivalently as a mapping from variable names to Boolean values. I define an *assignment* for an expression `e` to be an environment in which all of `e`'s free variables, and only its free variables, are bound. If `e` has $n \geq 0$ free variables, it will have exactly 2^n assignments. An assignment is a *satisfying* assignment if the formula is true given that binding, and an expression is *satisfiable* if it has at least one satisfying assignment. The last formula above is satisfied, (e.g. by the assignment `[(x,T),(z,F)]`, hence is satisfiable.

To represent quantified Boolean expressions in ML, we will use the following data type.

```
datatype expr = Const of bool
              | Var   of string
              | Not   of expr
              | And   of expr * expr
              | Or    of expr * expr
              | All   of string * expr
              | Exist of string * expr
exception UnboundVar
```

Thus, the ML expression `And(Not(Var "x"),Or(Const true, Var "x"))` is the same as the second example above. (But note that defining

```
val T = Const true
val F = Const false
val x = Var "x"
```

makes `And(Not(x),Or(T,x))` equivalent to this, which may be a convenient shorthand for generating your test cases. Also note that All/Exist take `string*expr`, not `expr*expr`; i.e., you need quotes around the first but not the second `x` in `All("x",Or(T,x))`, even after giving the 3 shorthand `val` bindings.)

(a) Write a function `free_vars` that takes an expression and returns a `string list` of its *free* variables. Variables should not be repeated even if they appear multiple times, but order is irrelevant; use the set functions above. You might want to review the "exp" example in lecture 4. (But now that you know more about pattern matching, you could do the "eval" function much more elegantly than given there, couldn't you?)

(b) As above, we will represent an environment by a `(string * bool) list`. Write a function `getenv x env` that will return the binding of variable `x` in environment `env`. E.g., `getenv "z" [("x",true),("z",false),("z",true)] --> false`. (Having newer bindings, nearer the front of the list, hide older ones is a natural implementation; no need to remove them.) Raise an `UnboundVar` exception if `x` is not bound in `env`.

(c) Write a function `eval f env` that takes an expression `f` and an environment `env` and evaluates the expression in that environment. If the formula's value depends on free variables unbound in `env`, it should raise an `UnboundVar` exception.

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

(d) There are several kinds of manipulations that we might want to do to Boolean expressions. For starters, write a function `fix1` that takes a variable, a Boolean value and an expression, and returns an expression with all free instances of that variable fixed to the given truth value. It should be a curried function. For example,

```
fix1 "x" true (And(Var "x", Or(All("x", Var "x"), Var "x")))
          --> And(Const true, Or(All("x", Var "x"), Const true))
fix1 "x" true (And(Var "z", Or(Var "y", Var "z")))
          --> And(Var "z", Or(Var "y", Var "z"))
```

(e) `fix1` 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, or to swap the names of two free variables. Write a function `fix` that takes a function `fixer` and an expression `e`, applies `fixer` to the argument of each free `Var` in `e`, and leaves the rest of the expression unchanged. `fixer` should take the value from a `Var` and return an expression. (Hint: this one is a little more subtle than `fix1`. E.g., `fixer` might want to change both x and y, but inside `e` of `All(x,e)`, any `y`'s should still be changed, but no `x`'s. How will you keep track?)

Then write functions `fixvar`, `changevar` and `swapvar` using `fix` so that `fixvar` is equivalent to `fix1`, `changevar` is similar except that variables are changed, and `swapvar` interchanges two variable names. For example,

```
fixvar "x" true (And(Var "x",Or(All("x",And(Var "x",Var "y")), Var "x")))
    --> And (Const true,Or (All ("x",And (Var "x",Var "y")),Const true))
changevar "x" "help" (And(Var "x",Or(All("x",And(Var "x",Var "y")), Var "x")))
    --> And (Var "help",Or (All ("x",And (Var "x",Var "y")),Var "help"))
swapvar "x" "y" (And(Var "x",Or(Var "x",Var "y")));
    --> And (Var "y",Or (Var "y",Var "x"))
```

(f) `swapvar` only changes free variables. Or does it? In a comment after these definitions, give an example (executable code) of a case where the result of applying `swapvar` to a formula `f` has fewer free variables than `f` does. This is known as *capture*.

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

Hint: write a local function that takes a list of free variables and an environment. If the list is empty, evaluate the expression in that environment; otherwise recurse, extending the environment with a new binding and using `@` (append) to collect resulting satisfying assignments. My implementation is about 8 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

The following is not too difficult and *strongly recommended* as additional practice with functions. Make liberal use of map, fold, curry, compose, etc. No recursion (or loops) allowed. Extra challenge:

it's not really good style, but try to define your function in one (somewhat long) line.

Suppose you are given a Boolean formula `f` together with an environment `pay` binding its free variables x_i to ints p_i, reflecting *payoffs*: setting x_i to true earns p_i dollars; setting it to false costs $-p_i$ dollars. Write a function `maxpay f payoff` which will find the total value of `f`'s highest-paying satisfying assignment.

Type Bindings

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

```
infix mem union
val mem = fn : ''a * ''a list -> bool
val addmem = fn : ''a -> ''a list -> ''a list
val remmem = 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
datatype expr
  = All of string * expr
  | And of expr * expr
  | Const of bool
  | Exist of string * expr
  | Not of expr
  | Or of expr * expr
  | Var of string
exception UnboundVar
val free_vars = fn : expr -> string list
val fix1 = fn : string -> bool -> expr -> expr
val fix = fn : (string -> expr) -> expr -> expr
val fixvar = fn : string -> bool -> expr -> expr
val changevar = fn : string -> string -> expr -> expr
val swapvar = fn : string -> string -> expr -> expr
val getenv = fn : ''a -> (''a * 'b) list -> 'b
val eval = fn : expr -> (string * bool) list -> bool
val satisfying_assignments = fn : expr -> (string * bool) list list
```