Higher-Order ML Functions and Function Arity

(a) Write an ML function, curry, that takes a function of two arguments and returns the curried version of that function. These examples may help to guide your intuition:

```ml
fun curry f = ...
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
fun cons (x, []) = [x] | cons (x, xs) = x::xs;
val cons = fn : 'a * 'a list -> 'a list
val cons = fn : 'a * 'a list -> 'a list
val it = [5] : int list
val cons 5 nil; (* error *)
val (curry cons) 5 nil;
val it = [5] : int list
```

(b) Write an ML function, uncurry, that takes a curried function as an argument and returns the equivalent function of two arguments:

```ml
fun uncurry f = ...
val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
fun append (ls) (x) = ls@x;
val append = fn : 'a list -> 'a list -> 'a list
val l1 = [1,2,3]; val l2 = [4,5];
val append l1 l2;
val it = [1,2,3,4,5] : int list
val append (l1,l2); (* error *)
val (uncurry append) (l1,l2);
val it = [1,2,3,4,5] : int list
```

(c) (strictly optional: for personal edification and extra credit) It seems natural to want to bind a function created with curry, such as (curry cons), to a variable so that we can call it later in a more compact form. However, the attempt to do so will produce the following errors:

```ml
val consC = curry cons;
stdin:23.1-23.23 Warning: type vars not generalized because of
value restriction are instantiated to dummy types (X1,X2,...)
val consC = fn : ?.X1 -> ?.X1 list -> ?.X1 list
val consC 5 nil;
stdin:23.1-23.12 Error: operator and operand don’t agree [literal]
operator domain: ?.X1
operand:     int
in expression:
    consC 5
```

Explain why the ML type inference system forbids this definition.
2.  

ML type inference

Give the types of each of the following ML functions. For each problem, show the equations used to derive the function’s type. (It is not sufficient to simply type these in to ML and report the result of the type inference system. On the other hand, doing so can provide a useful reality check).

(a) fun map (p, nil) = nil
    | map (p,x::xs) = map (p,xs);

(b) fun f p nil = true
    | f p (x::xs) = if (not (p x))
        then false
        else (f p xs);

(c) The implementation of map has a bug—what is it? How might knowing the type of this function help the programmer to find the bug?

3.  

Static and Dynamic Scope

Consider the following program fragment, written both in ML and in pseudo-C:

```plaintext
1 let x = 3 in { int x = 3; { 
2     let fun f(y) = x + y in int f (int y) { return x + y; } ( 
3     let x = 5 in int x = 5; { 
4         x + 
5         f(x) f(x); 
6     end 
7     end } 
8 end; }
```

The C version would be legal if C had nested functions; it’s just there for reference if you find the ML hard to decipher.

(a) Under static scoping, what value does the above code return? During the execution of this code, the value of x is needed three different times (on lines 2, 4, and 5). For each line where x is used, state what numeric value is used when the value of x is requested and explain why these are the appropriate values under static scoping.

(b) Under dynamic scoping, what value does the above code return? For each line where x is used, state which value is used for x and explain why these are the appropriate values under dynamic scoping.
4. ........................................ Lambda calculus and scope

Consider the following ML expression:

```ml
fun foo(x:int) = 
    let fun bar(f) = fn x => f (f (x)) 
    in 
        bar(fn y => y + x) 
    end;
```

In $\beta$-reduction on lambda terms, the function argument is substituted for the formal parameter. In this substitution, it is important to rename bound variables to avoid capture. This question asks about the connection between names of bound variables and static scope. Using a variant of substitution that does not rename bound variables, we can investigate dynamic scoping.

(a) The following lambda term is equivalent to the function $\text{foo}$:

$$
\lambda x. ( (\lambda f. \lambda x.f(x)) (\lambda y.y + x) )
$$

Use $\beta$-reduction to reduce this lambda term to normal form.

(b) Using the example reduction from part (a), explain how renaming bound variables provides static scope. In particular, say which variable above ($x$, $f$ or $y$) must be renamed, and how some specific variable reference is therefore resolved statically.

(c) Under normal ML static scoping, what is the value of the expression $\text{foo}(3)(2)$?

(d) Give a lambda term in normal form that corresponds to the function that the expression in part (a) would define under dynamic scope. Show how you can reduce the expression to get this normal form by not renaming bound variables when you perform substitution.

(e) Under dynamic scoping, what is the value of the expression $\text{foo}(3)(2)$?

(f) In the usual statically scoped lambda calculus, $\alpha$-conversion (renaming bound variables) does not change the value of an expression. Use the example expression from part (a) to explain why $\alpha$-conversion may change the value of an expression if variables are dynamically scoped. (This is a general fact about dynamically-scoped languages, not a peculiarity of lambda calculus.)