Question 1. (9 points) What are the types of the following function definitions?

(a) fun pick x y = y 
   'a -> 'b -> 'b 

(b) fun pr f x y = f(x,y) 
   ('a * 'b -> 'c) -> 'a -> 'b -> 'c 

(c) fun fpair z = pr(z) (* pr defined in part (b) *) 
   ('a * 'b -> 'c) -> 'a -> 'b -> 'c 

Question 2. (6 points) Consider the following SML expressions:

   fun g a b = fn x => a(x) + b;
   val y = 10;
   val f = g(fn y => y*y) 

(a) What is the type of function g? 
   ('a -> int) -> int -> 'a -> int 

(b) What is the type of f? 
   int -> int -> int 

(c) What is the result of evaluating f 5 3? 
   14
**Question 3.** (10 points) (Hint: you may find it useful – and the graders might get some hints if partial credit is needed – if you make some notes about the values and/or bindings of the various parts of the following expressions. But be sure that we can find the answers!)

(a) Consider the following SML expressions:

```
val x = 10;
fun f y = x * y;
fun g z =
  let
  val x = 3
  in
  f(z) + x
  end;
```

What is the value of $g\ 2$?

23

(b) Consider the following SML expressions:

```
fun f x y z = x (y) + z
val y = 3
fun g z = let
  val x = fn x => x * 2
  in
  f z
  end
val h = g (fn a => a*a)
```

What is the value of $h\ 5\ 2$?

27
**Question 4.** (8 points) Write simple recursive function \( \text{nOdd \ lst} \) that calculates the number of odd integers in the list \( \text{lst} \). For example, \( \text{nOdd[]} \) should evaluate to 0, \( \text{nOdd}[1,2,3,4] \) is 2, and \( \text{nOdd}[3, 5, 2, 5, 8, 6] \) is 3.

For full credit your solution **must** use pattern matching, not the \text{hd} and \text{tl} functions or \text{if}-statements. Also, if your solution involves an auxiliary, or helper function, that function should be defined locally in \text{nOdd} and not defined externally as a top-level function.

You should assume that the list is either empty or contains only positive integer values. Your function does not need to be tail-recursive.

\[
\text{fun nOdd \ nil = 0} \\
| \ nOdd (x::xs) = (x \mod 2) + nOdd(xs)
\]

**Question 5.** (10 points) The \( n \)th Fibonacci number can be calculated with the following recursive function:

\[
\text{fun fib \ n = if \ n < 2} \\
\quad \text{then 1} \\
\quad \text{else fib(n-1) + fib(n-2)}
\]

While it produces the correct answer, this function has the unfortunate property that its running time is exponential \((O(2^n))\). However, a simple iterative function can calculate the result in linear time \((O(n))\).

Write a **tail-recursive** version of fib that calculates \( \text{fib \ n} \) in linear time. If you define any auxiliary (helper) functions as part of your solution, they should be placed inside let bindings so they are local to \text{fib} and not defined in the global environment.

Hint: You almost certainly will want an auxiliary function, and you may find it helpful to have more than one “accumulator”-like parameter. (It might help to think about how you would solve this problem with a single loop in a language with such constructs.)

Hint: Don’t worry about the linear time restriction at first. A simple iterative algorithm will likely be linear time once you’ve figured it out.

\[
\text{fun fib \ n = let} \\
\quad \text{fun aux(k, last, prev) =} \\
\quad \quad \text{if \ k >= n} \\
\quad \quad \quad \text{then last + prev} \\
\quad \quad \text{else aux(k+1, last+prev, last)} \\
\quad \text{in} \\
\quad \quad aux(2,1,0) \\
\text{end}
\]
Question 6. (11 points) If you recall from a homework assignment, a tree structure containing integer values can be defined in SML with the following type:

```sml
datatype tree = Tree of int * tree * tree
  | EmptyT
```

(a) (8 points) Write a function `treemap f t` that has two parameters: a function `f` whose type is `int->int`, and a tree `t` of type `tree`. The result of evaluating `treemap f t` should be a new tree that is a copy of the original tree `t`, except that the `int` value in each node should be calculated by applying the function `f` to the corresponding node value in the original tree. (In other words, `treemap` is a map function for trees the same way that the standard library `map` function maps a function onto a list.)

```sml
fun treemap f EmptyT = EmptyT
  | treemap f (Tree(x, left, right)) =
      Tree(f(x), (treemap f left), (treemap f right))
```

(b) (3 points) Use `treemap` to define a new function `doubletree t` that returns a copy of the tree `t` where each node in the original tree has an integer value twice that of the corresponding node in the original tree. You *may not* define any additional global bindings. Hint: partial application (e.g., Currying) and anonymous functions are both useful here.

```sml
val doubletree = treemap (fn x => x+x)
```
Question 7. (8 points) Although most of the examples we’ve seen of SML structures use a signature to specify the type of the structure, this isn’t required. If we define a structure without naming a signature, then we create a set of bindings that contain all of the items in the structure. For example, we might want to create a structure containing definitions for complex numbers and associated operations.

```plaintext
structure cpx = struct
  type complex = real*real
  fun make_complex(x,y) = (x,y): complex
  fun sum((x1,y1), (x2,y2)) = make_complex(x1+x2, y1+y2)
  fun prod((x1,y1), (x2,y2)) = make_complex(x1*x2-y1*y2, x1*y2+x2*y1)
  fun recip(x,y) = let val t = x*x + y*y
                  in  make_complex(x/t, ~y/t) end
  fun quot(x,y) = prod(x, recip y)
end
```

When SML process this definition, it reports the following inferred signature and types:

```plaintext
structure cpx : sig
  val make_complex : (real*real) -> complex
  val sum          : (real*real) * (real*real) -> complex
  val prod         : (real*real) * (real*real) -> complex
  val recip        : real*real -> complex
  val quot         : (real*real) * (real*real) -> complex
  type complex = real*real
end
```

Unfortunately, this exposes the representation details of type `complex` to code that uses the structure `cpx`. It also exposes all of the functions defined in the structure, even though we might prefer to hide some of them that are only part of the implementation. We can fix both of these problems by defining an appropriate signature and changing the first line of the structure to use that signature (e.g., `structure cpx :> complex`). Complete the definition of signature `complex`, below, so that when it is implemented by structure `cpx`, the representation details of type `complex` and the function `recip` are hidden and not visible outside the structure. (Function `sum` is specified for you below to get started; you should add specifications for the other public items.)

```plaintext
signature complex = sig
  type complex
  val make_complex: real * real -> complex
  val sum: complex*complex -> complex
  val prod: complex * complex -> complex
  val quot: complex * complex -> complex
end
```
**Question 8.** (8 points) The following two functions evaluate whether some property is true of any or all of the items in a list.

```haskell
fun exists p nil = false
  | exists p (a::x) = if p a then true else exists p x

fun all p nil = true
  | all p (a::x) = if p a then all p x else false
```

In other words, `exists p lst` returns true if `p x` is true for *any* item in the list `lst`, and `all p lst` returns true if `p x` is true for *every* item in the list `lst`. A few examples:

- `exists (fn x => x>0) [-1,2,3]` evaluates to true,
- `exists (fn x => x>0) [-1,-2,-3]` evaluates to false;
- `all (fn x => x>0) [1,2,3]` evaluates to true,
- `all (fn x => x>0) [1,2,-3]` evaluates to false.

Next, consider the following function:

```haskell
fun C x y = (y mod x = 0)
```

This function returns true if the integer `y` is a multiple of `x` and false otherwise. Examples: `C 1 3` evaluates to true; `C 2 3` evaluates to false.

Now use the functions `exists`, `all`, and `C` to write an expression that solves the following problem: Given two lists `X` and `Y` that contain integers, return true if there is some integer `x` in list `X` such that all of the integers in list `Y` are a multiple of `x`. If no such integer exists in list `X`, return false. You may assume that `X` and `Y` are non-empty lists containing positive integers.

For full credit, your solution should *not* contain recursions that directly process the elements of `X` and `Y` individually – use `exists` and `all` and appropriate functional parameters.

```haskell
exists (fn z => all (C z) Y) X
```