Sample Solution

Question 1. (8 points) What are the types of the following function definitions?

(a) `fun clone x = (x, x);`

`'a-> 'a * 'a`

(b) `fun fst(x, y) = x;`

`'a * 'b -> 'a`

(c) `fun ffst z = fst (fst z);`

`('a * 'b) * 'c -> 'a`

(d) `fun g (x, y, z) = x (y z);`

`('a -> 'b) * ('c -> 'a) * 'c -> 'b`

Question 2. (8 points) Write a *tail-recursive* function `len lst` that calculates the length of the list `lst`. For example, `len[]` should evaluate to 0, `len[1,2,3,4]` should evaluate to 4, `len[[1,2,3], 4]` should evaluate to 2. For full credit your solution **must** use pattern matching, not the `hd` and `tl` functions or if-statements. Also, if your solution involves an auxiliary, or helper function, that function should be defined locally in `len` and not defined externally as a top-level function.

```
fun len lst =
    let fun f(lst,acc) =
          case lst of
             [] => acc
           | hd::tl => f(tl,acc+1)
    in
          f(lst,0)
    end
```

(There's a bug in the question that wasn't caught during proofreading – the expression len[[1,2,3], 4] won't typecheck since [1,2,3] and 4 have different types, so it should not have been included as an example.)

Sample Solution

Question 3. (3 points) SML provides a lot of "syntactic sugar" to make it possible to use convenient notation for more basic underlying constructs. For instance, we can define a tuple e

```
val e = (123, 456, 789);
```

and reference its fields as `#1 e`, `#2 e`, `#3 e`. But this is syntactic sugar for a record datatype. How could you define `e` if the tuple syntactic sugar were not available?

```
val e = { 1 = 123, 2 = 456, 3 = 789 };
```

Question 4. (8 points) Arithmetic expressions involving integers, addition, and multiplication, can be represented as a data structure in an ML program with the following data type.

```
datatype expr =  Int of int
              | Prod of expr * expr
              | Sum of expr * expr
```

Write a recursive function `eval e:expr` that, given an expression `e`, evaluates the expression and returns its value.

```
fun eval(e:expr) =
    case e of
         Int n => n
       | Prod(x,y) => eval(x) * eval(y)
       | Sum(x,y)  => eval(x) + eval(y)
```

Sample Solution

Question 5. (6 points) For each of the following sets of expressions and definitions, write the value of the final expression.

(a)
```
val k = 17;
fun f k = k+1;
fun g n = f k;
val k = 42;
g(k+1);
```

18

(b)
```
val n = 2;
fun f x = let val y = x+1 in fn g => n+y end;
fun g x = f 4;
g 1 2;
```

7

Question 6. (8 points) Write a curried function `head` that has two parameters, an integer `k` and a list `lst`. The result of executing `head k lst` should be a list consisting of the first `k` items in `lst`. For example, `head 3 [1,2,3,4,5]` should evaluate to `[1,2,3]`. The result of evaluating `head k` should be a function that, when applied to a list, yields the first `k` items in the list. So, for example, if the result of `head 3` is applied to the list `[1,2,3,4,5]`, it should evaluate to `[1,2,3]`. If the list has fewer than `k` elements, the function `head k` (or `head k lst`) should generate a `TooFewElements` exception.

```
exception TooFewElements;

fun head k lst =
           case k of
              0 => []
            | _ => case lst of
                         [] => raise TooFewElements
                       | hd::tl => hd::(head (k-1) tl)
```

Sample Solution

Question 7. (3 points) Both of the following signatures define the interface to a complex number structure. What's the significant difference between them from the perspective of a programmer using these signatures?

```
signature COMPLEX_A =
sig
    datatype complex = Pair of real * real | Real of real
    val make_complex : real * real -> complex
    val add : complex * complex -> complex
    val print_complex : complex -> unit
end

signature COMPLEX_B =
sig
    datatype complex
    val make_complex : real * real -> complex
    val add : complex * complex -> complex
    val print_complex : complex -> unit
end
```

In the 2nd signature, `COMPLEX_B`, the representation of the `complex` type is abstract, meaning that client code can't see the `Pair` and `Real` constructors and can't directly access the components of a `complex` value.

Sample Solution

Question 8. (8 points) The ML standard library provides several higher-order functions for manipulating lists, in particular `map`, `filter`, `foldl` (fold left), and `foldr` (fold right). These are defined as follows:

`map f [x1, …, xn]` = `[f x1, …, f xn]`

`filter f [x1, …, xn]` = a list containing all elements `xi` in the original list where `f xi` evaluates to true

```
foldl f e [x1, …, xn] = f(xn, …, f(x1, e)…)
foldr f e [x1, …, xn] = f(x1, …, f(xn, e)…)
```

The fold functions apply the function f to the list elements from left to right (`foldl`) or right to left (`foldr`) to produce a single result.

(a) What are the types of these functions?

```
map      ('a -> 'b) -> 'a list -> 'b list

foldl    ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

(b) Use some combination of these functions and any anonymous functions you need to define a function `sumpos` that returns the sum of all the positive numbers in a list of integers, for example, `sumpos [3, -4, 12, 0, 5]` would evaluate to 20. You can assume that the list has type `int list` (i.e., it only contains integers). You should not use any loops or recursion in your solution – just use some combination of the higher-order functions to calculate the result – and you should not define (bind) any other top-level functions other than `sumpos`.

```
fun sumpos lst =
  foldl (fn (x,y) => (x+y)) 0
              (filter (fn x => (x>0)) lst)
```