Name:____

CSE341 Autumn 2017, Midterm Examination
October 30, 2017

Please do not turn the page until 2:30.

Rules:

- The exam is closed-book, closed-note, etc. except for *one* side of one 8.5x11in piece of paper.
- **Please stop promptly at 3:20.**
- There are **100 points**, distributed **unevenly** among **6** questions (all with multiple parts):
- **The exam is printed double-sided.**

Advice:

- Read questions carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what is your final answer.
- The questions are not necessarily in order of difficulty. Skip around. Make sure you get to all the questions.
- If you have questions, ask.
- Relax. You are here to learn.

Name:____

1. (**20** points) This problem uses this datatype binding, where an `exp` is a simple arithmetic expression like we studied in class except instead of negations and multiplications, we have doubling and (integer) division.

```
datatype exp = Constant of int
             | Double of exp
             | Add of exp * exp
             | Divide of exp * exp
```

   (a) Write a function `eval_exp` of type `exp -> int` that returns the "answer" for "executing" the arithmetic expression. Some notes on division:

   - Use integer division, which in ML is done with the infix operator `div`. For example, in ML, `6 div 4` is `1`.
   - Division by zero will raise an exception, which is fine.

   (b) Give an example of a value of type `exp` where:

   - Calling `eval_exp` with your expression causes a division-by-zero exception, but ...
   - ... no use of the `Divide` constructor has `Constant 0` as its second argument.

   (c) Write a function `no_literal_zero_divide` of type `exp -> bool` that returns true if and only if no use of the `Divide` constructor has `Constant 0` as its second argument. Notes:

   - So, `no_literal_zero_divide` applied to your answer to the previous question would evaluate to `true`.
   - You should *not* use `eval_exp` — this question has nothing to do with evaluating expressions.

Solution:

(a)
```
fun eval_exp e =
  case e of
      Constant i => i
    | Double e => 2 * eval_exp e
    | Add(e1,e2) => (eval_exp e1) + (eval_exp e2)
    | Divide (e1,e2) => (eval_exp e1) div (eval_exp e2)
```

(b) Many possible answers such as
```
Divide(Constant 4, Double(Constant 0))
Divide(Constant 4, Add(Constant 0, Constant 0))
```

(c)
```
fun no_literal_zero_divide e =
  case e of
      Constant _ => true
    | Double e => no_literal_zero_divide e
    | Add(e1,e2) => no_literal_zero_divide e1 andalso no_literal_zero_divide e2
    | Divide(_,Constant 0) => false
    | Divide(e1,e2) => no_literal_zero_divide e1 andalso no_literal_zero_divide e2
```

Name:____

2. (**20** points) This problem uses this somewhat silly function:

```
          fun f (xs,ys) =
            case (xs,ys) of
(* 1 *)         ([],[]) => SOME 0
(* 2 *)       | (x::[], y::[]) => SOME (x+y)
(* 3 *)       | (x1::x2::[], y1::y2::[])  => SOME (x1 + x2 + y1 + y2)
(* 4 *)       | (x1::x2::xs', y1::y2::ys')  => f (xs',ys')
(* 5 *)       | _ => NONE
```

(a) What is the type of `f`?

(b) What does `f([3],[10])` evaluate to?

(c) What does `f([3,4],[10,11])` evaluate to?

(d) What does `f([3,4,5],[10,11,12])` evaluate to?

(e) What does `f([3,4,5,6],[10,11,12,13])` evaluate to?

(f) Describe in at most 1 English sentence *all* the inputs to `f` such that the result of `f` is `NONE`.

(g) Yes or no: Is `f` tail-recurisve?

For each of the remaining questions, give one of these answers (just the letter is enough):

A. The result no longer type-checks.
B. The result type-checks but gives different answers for some inputs.
C. The result type-checks and gives the same answer for all inputs.

Also, ignore the syntax detail that the first branch has no `|` character and the others do — assume that is fixed appropriately.

(h) What happens if we move branch 2 of `f` to be the first pattern in the case expression?

(i) What happens if we move branch 3 of `f` to be the first pattern in the case expression?

(j) What happens if we move branch 4 of `f` to be the first pattern in the case expression?

(k) What happens if we move branch 5 of `f` to be the first pattern in the case expression?

Solution:

(a) `int list * int list -> int option`
(b)-(e) `SOME 13`, `SOME 28`, `SOME 17`, `SOME 36`
(f) Any pair of lists of ints where the lists have different lengths
(g) Yes
(h)-(k) C, C, A, A

Name:____

3. (**12** points) In this problem, we ask you to give *good* error messages for why a short ML program does *not* type-check. A *specific* phrase or short sentence is plenty.

For example, for the program,

```
fun f1 (x,y) = if x then y + 1 else x
```

a fine answer would be, "the then-branch-expression and the else-branch-expression do not have the same type."

Give good error messages for each of the following:

(a)
```
fun f2 g xs =
  case xs of
      [] => []
    | x::xs' => (g x) :: f2 xs'
```

(b)
```
fun f3 xs =
  case xs of
      [] => NONE
    | x::[] => SOME 1
    | x::xs' => SOME (1 + (f3 xs'))
```

(c)
```
datatype t = A of int | B of (int * t) list
fun f4 x =
  let
      fun aux ys =
        case ys of
            [] => []
          | (i,j)::ys => (i+1,j)::(aux ys)
  in
      case x of
          A i => x
        | B ys => B (aux x)
  end
```

(d)
```
exception Foo
fun f5 x = if x > 3 then x else raise Foo
fun f6 y = (f5 (y+1)) handle _ => false
```

Solution:

(a) Recursive call is missing its first argument, probably want `f2 g xs'`. (We also allowed answers indicating that `f2` applied to one argument is a function so it cannot be the second argument to `::`.)

(b) The result of the recursive call is an option, so you can't add it. (`valOf (f2 xs')`) would type-check.)

(c) The (non-recursive) call to `aux` passes a `t` but `aux` expects a list.

(d) In `e1 handle _ => e2`, `e1` and `e2` need the same type, but here they have types `int` and `bool`.

Name:____

4. (**21** points)

   (a) Without using any helper functions (except `::`) write a function `zipWith` of type `('a * 'b -> 'c) -> 'a list -> 'b list -> 'c list` as follows:

   - It takes three arguments in curried form.
   - The length of the result is the length of the shorter of the second or third argument.
   - The i^{th} element of the output is the first argument applied to the i^{th} elements of the second and third arguments.

   (b) Use a `val` binding and a partial application of `zipWith` to define a function `first_bigger` of type `int list -> int list -> bool list` where, for example,
   `first_bigger [1,7,9] [0,10,9,4,2] = [true, false, false]`

   (c) Here are two ML library functions:

   - `List.map : ('a -> 'b) -> 'a list -> 'b list`
     map as discussed in class, with curried arguments
   - `ListPair.zip : 'a list * 'b list -> ('a * 'b) list`
     equivalent to `zipWith (fn pr => pr)` except takes its arguments as a pair

   Reimplement `zipWith` in one line using these two library functions and a `fun` binding.

   (d) How many times does `zipWith (fn _ => true) [1,2,3] [7,8,9]` call the `::` *function* (so do not count uses of the `::` *pattern*) if `zipWith` is your answer to part (a)?

   (e) How many times does `zipWith (fn _ => true) [1,2,3] [7,8,9]` call the `::` *function* (so do not count uses of the `::` *pattern*) if `zipWith` is your answer to part (c)?

Solution:

(a)
```
fun zipWith f xs ys =
  case (xs,ys) of
      ([],_) => []
    | (_,[]) => []
    | (x::xs',y::ys') => (f(x,y)) :: zipWith f xs' ys'
```

(b) `val first_bigger = zipWith (fn (x,y) => x > y)`

(c) `fun zipWith f xs ys = List.map f (ListPair.zip (xs,ys))`

(d) 3

(e) 6 (we gave a little partial credit for 0 but calling a function that calls `::` should definitely "count")

Name:____

5. (**8** points) Here is a definition of `flat_map` as shown in section (recall `@` is list append):

```
fun flat_map f xs =
  case xs of
      [] => []
    | x::xs' => (f x) @ flat_map f xs'
```

(a) Reimplement a curried `map` of type `('a -> 'b) -> 'a list -> 'b list` in one line using a `fun` binding and `flat_map`.

(b) Reimplement a curried `filter` of type `('a -> bool) -> 'a list -> 'a list` in one line using a `fun` binding and `flat_map`.

Solution:

(a) `fun map f = flat_map (fn x => [f x])`

(b) `fun filter f = flat_map (fn x => if f x then [x] else [])`

Name:____

6. (**19** points) This problem considers an ML module `RBNum1` for numbers in the range 0–999 that also have a "color" of blue or red. The structure definition is on a separate page you will *not* turn in.

(a) Complete this signature definition so that clients of `RBNum1` can use all the function bindings in `RBNum1` but are not able to make "bad" values like `Red ~7` or `Blue 2000`.

```
signature RBNUM =
sig
    val max_value : int
    exception OutOfRange







end
```

(b) Complete this structure definition so that it also has signature `RBNUM` and is equivalent to `RBNum1` from any client's perspective. You need to add four bindings — *put them in the left column of the table below.*

```
structure RBNum2 :> RBNUM =
struct
type t = int
exception OutOfRange
val max_value = 999
fun red_num  i = if i > max_value orelse i < 0 then raise OutOfRange else i
fun blue_num i = if i > max_value orelse i < 0 then raise OutOfRange else i+1000
(* ... part (b) ... *)
end
```

(c) For each of the bindings you added in part (b), what are their types *inside* the `RBNum2` module? *Put your answers in the middle column of the table.*

(d) For each of the bindings you added in part (b), is it possible for the client to implement an equivalent function outside the module? *Put your yes/no answers in the right column of the table.*

part (b)	part (c)	part (d)

Solution:

```
signature RBNUM =
sig
    val max_value : int
    exception OutOfRange
    type t
    val red_num : int -> t
    val blue_num : int -> t
    val is_blue : t -> bool
    val is_red : t -> bool
    val is_max_blue : t -> bool
    val to_int : t -> int
end
```

part (b)	part (c)	part (d)
`fun is_blue x = x >= 1000`	`int -> bool`	No
`fun is_red  x = x < 1000`	`int -> bool`	No
`fun is_max_blue x = x = 1999`	`int -> bool`	Yes
`fun to_int x = if x >= 1000 then x - 1000 else x` `fun to_int x = x mod 1000`	`int -> int`	No

Notes:

- In part (c), the intended answers are those above, but since inside the module we have `type t = int`, any `int` above can be replaced with `t` and we allowed such answers.
- In part (d), an external client could implement `is_blue` as `not o is_red` and vice versa — we gave most of the credit for answers that explained this unexpected "trick."

Name:____

Here is an extra page in case you need it. If you use it for a question, please write "see also extra sheet" or similar on the page with the question.

Here is `RBNum1` on a separate page. Do *not* turn in this page, so do not write answers on it.

```
structure RBNum1 :> RBNUM =
struct

val max_value = 999

exception OutOfRange

datatype t = Red of int | Blue of int

fun red_num  i = if i > max_value orelse i < 0 then raise OutOfRange else Red i

fun blue_num i = if i > max_value orelse i < 0 then raise OutOfRange else Blue i

fun is_blue x = case x of Red _ => false | Blue _  => true

fun is_red  x = case x of Red _ => true  | Blue _  => false

fun is_max_blue x = case x of Red _ => false | Blue i => i = 999

fun to_int x = case x of Red i => i | Blue i => i

end
```