CSE 341 | Section 3

Q1: `datatype`s and pattern-matching

Colors are often represented based on red, green, and blue values (RGB), where the value for each of these three components is an integer between 0 and 255 (inclusive). The RGB value for Red is (255, 0, 0), the RGB value for Green is (0, 255, 0) and the RGB value for Blue is (0, 0, 255). We can represent other colors with a mix of RGB values. Suppose we have the following datatype defined to represent a color conveniently as one of the three primary colors, or a custom color:

```
datatype color = Red | Green | Blue | RGB of (int * int * int);
```

Write a function `invert_color` that takes a `color` argument and returns its inverse as a new RGB representation (as a `color` datatype), resulting from subtracting each original r, g, b value from 255. For example, the call `invert_color(RED)` should return the RGB value (0, 255, 255) and `invert_color(RGB(0, 25, 155))` should return the RGB value (255, 220, 100). Assume that if the color argument was constructed with the RGB constructor, the three integers are between 0 and 255 (inclusive). *Hint: use a case expression in your solution.*

Q2:

The `color` datatype is an example of a "one-of" datatype as discussed in lecture/readings. One alternative is to represent colors with a record:

```
{ color_name : string, r : int, g : int, b : int }
```

a. What is one advantage of using a datatype for our color representation?

b. What is an example where it would be more appropriate to represent a type as a record instead of a "one-of" datatype?

Q3: More datatype and Pattern-matching examples

a. Consider the following type and datatype:

```
type cart = real * real
datatype shape = Circle of cart * real (* coordinates and radius *)
      | Square of cart * real (* coordinates and side length *)
      | Rectangle of cart * real * real (* coordinates and side lengths *)
```

Write a function area which takes a shape as an argument and returns its area (as a real value).

b. Now recall this datatype to represent expression trees from lecture:

```
datatype exp = Constant of int
            | Negate of exp
            | Add of exp * exp
            | Multiply of exp * exp
```

Write a function `const_not_under_add` of type `exp -> bool` that returns true if and only if there exists a Constant in the expression that is not a child of an Add expression. For example, `const_not_under_add(Constant 341)` should return true, as should `const_not_under_add(Multiply(Constant 341, Add(Constant 0, Constant 1)))`.

Q4:

Consider the following code:

```
fun length l =
  case l of
      _::xs => 1 + length xs
    | [] => 0
```

Is it tail-recursive? Why/why not?

```
fun all_positive (accum, l) =
  case l of
```

```
  x::xs => all_positive (accum andalso x > 0, xs)
| [] => accum
```

Is it tail-recursive? Why/why not?

Datatypes and Pattern-Matching *(17au Midterm Question)*

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.