ASSIGNMENT 5 — FUNCTIONAL PROGRAMMING

COMP 3010 — ORGANIZATION OF PROGRAMMING LANGUAGES

1. Higher-Order Functions

Exercise 1. Consider the following function definition in Standard ML:

```
fun mystery x y = fn z => x (y (x z));
```

(1) Translate the `mystery` function to a λ-calculus expression. *HINT:* You will need λs introducing the parameters x and y around the body of `mystery`.

(2) What happens when you evaluate

```
mystery (fn x => 1+x) (fn y => 2*y);
```

(3) What happens when you evaluate

```
mystery (fn x => 1+x) (fn y => 2*y) 5;
```

(4) What happens when you evaluate

```
mystery (fn x => 1+x) (fn y => 2*y) 5 6;
```

Exercise 2. The `map` function, which changes every element of a list using a given operation, is written in Scheme as

```
fun map change_elem []      = []
  | map change_elem (x::xs) = change_elem x :: map change_elem xs;
```

so that a list `[x, y, ..., z]` is transformed like so

```
map f [x, y, ..., z] = [f x, f y, ..., f z]
```

`reduce`, which compresses a list by replacing every list constructor `::` with a chosen binary operation and the final empty list `[]` with a chosen constant, is written in Standard ML as

```
fun reduce change_cons change_nil []      = change_nil
  | reduce change_cons change_nil (x::xs) =
                change_cons(x, reduce change_cons change_nil xs);
```

so that a list `x :: y :: ... z :: []` is transformed as

```
reduce f e (x :: y :: ... z :: [])
= (f x (f y (... (f z e) ...)))
```

(1) What is the result of evaluating

```
map (fn x => x*x) [1,2,3,4,5];
```

(2) What is the result of evaluating

```
reduce (fn(x,y) => x+y) 0 [1,2,3,4,5];
```

(3) What is the result of evaluating

```
reduce (fn(x,y) => x+y) 0 (map (fn x => x*x) [1,2,3,4,5]);
```

(4) (Multiple Choice) Consider this definition of the function `f`:

```
fun f xs = reduce (fn(x,y) => x+y) 0 (map (fn z => z*z) xs);
```

Which of the following alternate definitions of `f` is equivalent to the one above that used `map` and `reduce`?

Date: Fall 2025 — 2 pages.

(a)
```
fun f []      = 0
  | f (x::xs) = (x+x) * f xs;
```

(b)
```
fun f []      = 0
  | f (x::xs) = (x*x) + f xs;
```

(c)
```
fun f []      = 0
  | f (x::xs) = x + f xs;
```

(d)
```
fun f []      = 0
  | f (x::xs) = x * f xs;
```

2. Algebraic Data Types

Exercise 3. Do *Concepts In Programming Languages* Exercise 5.3 on Nonlinear Pattern Matching (page 123).

Note, for parts (a) and (b), you can write the described functions in SML syntax as asked by the exercise, *OR* in your choice of Ruby, Python, or C syntax.

Exercise 4. Do *Concepts In Programming Languages* Exercise 5.7 on Disjoint Unions (page 125).

Exercise 5. In SML, *all* references must point to real values in the heap. In other words, SML does not support implicit null pointers in place of a reference. Instead, the SML data type declaration

```
datatype 'a option = NONE | SOME of 'a;
```

defines the generic type `'a option` of references which could *either* point to nothing (represented by the `NONE` constructor containing no data) *or* point to some actual `'a` in the heap (represented by the `SOME` constructor containing a value of type `'a`).

For example, the integer division operation `x div y` will raise an exception when the divisor `y` is 0. A safe version of division, which never raises an exception, can be written in SML as

```
fun safe_div(x, 0) = NONE
  | safe_div(x, y) = SOME(x div y);
```

which takes a pair of `int`s and returns an `int option`.

(1) What is the difference between the result of evaluating `10 div 0` versus `safe_div(10, 0)`?
(2) What is the difference between the result of evaluating `10 div 5` versus `safe_div(10, 5)`?
(3) What happens when you try to evaluate `2 * (10 div 5)`? What happens when you try to evaluate `2 * (safe_div(10, 5))`?