IN3043 Functional Programming
Solutions to Exercises 3

1. There is only one function that has that type (excluding those that fail):

   ```
   swap :: (a,b) -> (b,a)
   swap (x,y) = (y,x)
   ```

2. Again the polymorphic type determines the implementation:

   ```
   dup :: a -> (a,a)
   dup x = (x,x)
   ```

3. (a) This is very similar to `plusPoint`:

   ```
   minusPoint :: Point -> Point -> Point
   minusPoint (Point x1 y1) (Point x2 y2) = Point (x1-x2)
      (y1-y2)
   ```

   (b) This is similar to `scale` in the lecture:

   ```
   timesPoint :: Int -> Point -> Point
   timesPoint n (Point x y) = Point (n*x) (n*y)
   ```

   (c) This is just a translation of the definition:

   ```
   normPoint :: Point -> Int
   normPoint (Point x y) = abs x + abs y
   ```

   (d) We can do this without looking inside the points:

   ```
   distance :: Point -> Point -> Int
   distance p1 p2 = normPoint (minusPoint p1 p2)
   ```

   (e) This is a simple function of the type we did last week:

   ```
   oneStep :: Direction -> Point
   oneStep North = Point 0 1
   oneStep East = Point 1 0
   oneStep South = Point 0 (-1)
   oneStep West = Point (-1) 0
   ```

   Without parentheses around the `-1` arguments, these would mean something different (and illegal).

4. (a) This is a simple record type

```
data Turtle = Turtle Point Direction PenState
    deriving Show

data PenState = PenUp | PenDown
    deriving Show
```

It could be argued that it would be simpler to use a **Bool** to indicate whether the pen is down. Defining a specialized enumerated type for this purpose is bit more work, but it saves us from having to remember what **True** means.

(b) The definition is a transcription of the above description:

```
startTurtle :: Turtle
startTurtle = Turtle origin North PenUp
```

(c) The function matches the record and extracts the relevant component:

```
location :: Turtle -> Point
location (Turtle pos dir pen) = pos
```

In the above definition, the variables **dir** and **pen** are unused, and so could be replaces with underscores.

(d) This is an enumerated type, except for the move command:

```
data Command
    = TurnLeft | TurnRight | Move Int | RaisePen |
       LowerPen
    deriving Show
```

(e) The brings together the various functions we've defined in the **Geometry** module. The trickiest part is the move command.

```
action :: Turtle -> Command -> Turtle
action (Turtle pos dir pen) TurnLeft =
    Turtle pos (turnLeft dir) pen
action (Turtle pos dir pen) TurnRight =
    Turtle pos (turnRight dir) pen
action (Turtle pos dir pen) (Move n) =
    Turtle (plusPoint pos (timesPoint n (oneStep
       dir))) dir pen
action (Turtle pos dir _) RaisePen =
    Turtle pos dir PenUp
action (Turtle pos dir _) LowerPen =
    Turtle pos dir PenDown
```

5. We use guards to pick out the error case:

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv x y
  | y == 0 = Nothing
  | otherwise = Just (div x y)
```

6. We could do this with four clauses, covering all the possible combinations:

```
pairMaybe :: Maybe a -> Maybe b -> Maybe (a,b)
pairMaybe Nothing Nothing = Nothing
pairMaybe (Just x) Nothing = Nothing
pairMaybe Nothing (Just y) = Nothing
pairMaybe (Just x) (Just y) = Just (x,y)
```

   or we could note that three of the cases produce the same result and simplify this to

```
pairMaybe :: Maybe a -> Maybe b -> Maybe (a,b)
pairMaybe (Just x) (Just y) = Just (x,y)
pairMaybe _ _ = Nothing
```

7. This function is defined in the `Data.Maybe` module:

```
fromMaybe :: a -> Maybe a -> a
fromMaybe def Nothing = def
fromMaybe def (Just x) = x
```

8. Once again, the polymorphic type gives us no choice:

```
whatever :: Either a a -> a
whatever (Left x) = x
whatever (Right y) = y
```

9. Each argument could be one of two things, so we can use four equations to cover all the possibilities:

```
both :: Err a -> Err b -> Err (a,b)
both (OK x) (OK y) = OK (x,y)
both (OK x) (Err msg) = Err msg
both (Err msg) (OK y) = Err msg
both (Err msg1) (Err msg2) = Err (msg1 ++ "\n" ++ msg2)
```