Intro

Author: Matt Broadway
Date: 26/10/15
Disclaimer: This is not guaranteed to be correct (but I think it is). You decide whether to believe me :)

Question 4

“Implement an OCaml function called `tne` of type `((('a -> 'b) -> 'b) -> 'c) -> 'a -> 'c`.”

“Important: You may only use function definition (`let`) and function application.”

Simple Example

First consider this easier question, how could you implement a function with the type signature:

```
val f : ('a -> 'b) -> 'a -> 'b
```

This means in English:

- `f` is a function that takes two arguments and gives back a value of type `'b`
- the second argument has type `'a`, lets call it `x`
- the first argument is a function, lets call it `g`
  - `g` takes an argument of type `'a` and gives you a value of type `'b` in return.

From this information we can deduce that `f` must obtain a `'b` value by giving `g` what it wants, since it was not passed any `'b` values directly. So now we have:

(underscore indicates that we don’t know what should go there yet)

```
let f g x = g _;;
```

Now we examine what `g` wants. It wants a value of type `'a`. `f` was passed such a value (`x`) and so can give this value to `g`.

```
let f g x = g x;;
```

Solution To Exercise

Consider the type signature:

```
val tne : ((('a -> 'b) -> 'b) -> 'c) -> 'a -> 'c
```

This means in English:

- `tne` is a function that takes two arguments and gives back a value of type `'c`
- the first argument is a function (everything inside the brackets), lets call it `f`
- the second argument has type `'a`, lets call it `x`

We have so far:

(underscore indicates that we don't know what should go there yet)

```
val tne : "type of f" -> 'a -> 'c
val x   : 'a
val f   : _

let tne f x = _;;
```

focus on f

Just looking at the function now called `f`:

```
val f : (('a -> 'b) -> 'b) -> 'c
```

This means in English:

- `f` is a function that takes a single argument and gives back a value of type `'c`
- the argument is a function (everything inside the brackets), lets call it `g`

We can deduce that it is likely that we need to give `f` what it wants (`g`, which we will create), and the result of `f` is what `tne` its-self should return, since they both have a return type of `'c`

We have so far:

```
val tne : "type of f" -> 'a -> 'c
val x   : 'a
val f   : "type of g" -> 'c
val g   : _

let tne f x =
 let g _ = _
 in
    f g;;
```

focus on g

Just looking at the function now called `g`:

```
val g : ('a -> 'b) -> 'b
```

This means in English:

- `g` is a function that takes a single argument and gives back a value of type `'b`
- the argument is a function, lets call it `h`
    - `h` takes a single argument of type `'a` and gives back a value of type `'b`

We never call `g` ourselves. We need to write `g` in order to give it to `f` so that it will give back a `'c`. Because we never call `g` ourselves, we don't have to worry about writing the function `h`. Whoever calls the function `g` has to give it an appropriate `h` function.

We can however *use* `h` inside `g` because we are given it. We can give `x` to `h`, since `h` wants a type `'a` and `x` is of type `'a`. Doing this: `h` will give back a `'b`, which is the return type of `g`, so no more work has to be done in `g` (we have what we were after).

```
val tne : "type of f" -> 'a -> 'c
val x   : 'a
val f   : "type of g" -> 'c
val g   : "type of h" -> 'b
val h   : 'a -> 'b

let tne f x =
 let g h = h x
 in
    f g;;
```

This is the solution.

Alternative Solution

This is the model solution written by the demonstrators:

```
let dne a f = f a;;
let tne f a = f (dne a);;
```

This works by giving `a` to `dne` for it to use later, rather than relying on the fact that functions defined in a `let ... in` expression has access to the arguments of the function it was defined in. Otherwise the solutions are the same.