Anonymous Functions and Functions as Values

Anonymous Functions

Much of the power and flexibility of functional programming comes from its grounding in the "lambda calculus", an alternative but equivalent model of computation to Turing Machines. The lambda calculus emphasizes "abstraction" of variables over functions. Namely the main construct takes an expression like `x + x` and "abstracts" the variable `x` to obtain a function `fun(x) -> x + x`.

In programming languages this idea translates to the ability to create an "anonymous" function via a similar construct. Namely the following is valid OCAML code:

```
fun x -> x + x
```

If you type this in OCAML you will get back:

```
int -> int = <fun>
```

OCAML understood this as defining a new function. But just as if you had typed `(2, 3)`, it just prints that information and that information is then lost. We could do something with it first though, like actually calling the function right away, or storing it in a variable:

```
(fun x -> x + x) 5;;        (* prints 10 *)

let f = fun x -> x + x
in f 5;;
```

In fact in many ways the following two lines are equivalent:

```
let f = fun x -> x + x
let f x = x + x
```

The second version is simply "syntactic sugar" for the first version. To put it another way: Once we have "anonymous functions" in our language, and we have let bindings, we also automatically have "named functions". The only exception to that is the fact that recursive functions are a bit harder to pull off, as since the anonymous function has no name it would be hard to call it recursively. We will revisit this topic later.

Before moving on, let us consider using a previously defined function binding in a later assignment, like so:

```
let g x = x + 3
let f1 = g
let f2 x = g x
```

Exercise:

Describe the differences between the `f1` assignment and the `f2` assignment. Identify how they behave differently, and what may happen in the future. The `f2` assignment is often called *unnecessary function wrapping*.

Functions as Values

We have briefly discussed the idea of functions as values. We said that they are actually "closures", i.e. the function bodies themselves together with the current environment at the point of definition. What we focus on in this section is the idea that these closures can be the inputs to other functions, or the returned values from other functions.

Let's look at an example: Here is a, perhaps somewhat silly, function "evaluate" that takes a pair of a function and a value, and returns the result of evaluating the function at that value:

```
let evaluate (f, x) = f x
```

Before we continue, use type inference to determine the type of this function.

Let us look at another example: The function double takes as input a function, and returns as its result a function that on a given input applies f twice in a row. So the definition of this could look like:

```
let double f = fun x -> f (f x)
```

Once again, start by determining the type of double using type inference.

Let's see this in action. We define a function that adds 3 to a number, then we feed it into double:

```
let add3 x = x + 3
let double f = fun x -> f (f x)
let mystery = double add3
```

Here is another example: A function that takes as input an integer and returns the function that to an integer adds this integer. We can define it thus:

```
let addTo x = fun y -> x + y
```

We could then write our mystery function earlier as:

```
let mystery = double (addTo 3)
```

Functions that return functions as their values will play a recurring theme, especially in our study of currying in the next section.

Question: Why does addTo work? Why does the returned function know what x is when it gets called later on?