Note 1: Recursive Functions in Lambda Calculus

EN.601.426/626 Principles of Programming Languages – SP26

In this note, we want to derive recursive functions in lambda calculus. We are looking at the factorial function `fac`. Mathematically, we define it as follows:

$$\texttt{fac}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \texttt{fac}(n-1) & \text{if } n > 0 \end{cases}$$

Lambda calculus is a theoretical model of computation. Let's use JavaScript as a practical language to implement the factorial function, since JavaScript has first-class functions and is quite similar to lambda calculus.

Let's start with a straightforward implementation of the factorial function.

```
function fac(n) {
  if (n == 0) { return 1 }
  else { return n * fac(n - 1) }
}
```

We can also write it using an anonymous function and assign it to a variable `fac`:

```
fac = n => {
  if (n == 0) { return 1 }
  else { return n * fac(n - 1) }
}
```

The body of the function seems ugly. We can use the ternary operator `?:` to make it more concise:

```
fac = n => n == 0 ? 1 : n * fac(n - 1)
```

This is a good program, but it is still not lambda calculus-y. We don't really like to have variable assignment and the ability to name functions. Let's try removing the variable assignment and function name:

```
// Note: incomplete program, but looks pretty
f => n => n == 0 ? 1 : n * f(n - 1)
```

The goal of the above code is to introduce a function `f` that can be used for recursion. The form of the program is very good, we would like this to be our final program to write for the factorial function. However, it is incomplete yet, since `f` is not provided to you. (we assume `n` will be provided when you call the function).

Therefore, the first attempt is to provide `f` with the same function itself, therefore We basically do self-application.

```
// Note: bad program
(f => n => n == 0 ? 1 : n * f(n - 1))
  (f => n => n == 0 ? 1 : n * f(n - 1))
```

The problem is that the function being passed in takes in two arguments (curried). But when we call it (`f(n - 1)`), we are only providing one argument, which is not correct. The solution is to do `f(f)(n - 1)` instead of `f(n - 1)`, which means we are applying the function to itself first, and then applying the result to `n - 1`.

```
(f => n => n == 0 ? 1 : n * f(f)(n - 1))
  (f => n => n == 0 ? 1 : n * f(f)(n - 1))
```

This is a perfectly good program! Applying the above code to an argument (e.g., `10`) will give you the factorial of that argument. At this point, **we have successfully derived a recursive function in lambda calculus**, already.

There are still some problems with the above program, though. The repetition overall seems ugly. Can we do better by introducing a simple abstraction for self-application? Yes! We can introduce a lambda function $\lambda g.g\ g$ to eliminate repetition.

```
(g => g(g))
  (f => n => n == 0 ? 1 : n * f(f)(n - 1))
```

Much better! There is one last problem, as we still have a self-application in the body of the function (`f(f)`). In order to eliminate that, we are going to abstract away the self-application again.

Looking at $\lambda g.g\ g$, we can see that it is a function that takes in a function and directly applying itself. What if we want it such that before application, the main function `f` is already applied to itself?

```
// Good Lambda Program, Bad JavaScript Program
(g => (h => g(h(h))) (h => g(h(h))))
  (f => n => n == 0 ? 1 : n * f(n - 1))
```

We have basically changed `g(g)` to `(h => g(h(h))) (h => g(h(h)))`. Notice that this is still a self-application. But the difference is that the thing we pass to `g` is not just a function, but `h(h)`, which is the `f(f)` we wanted to abstract away.

Actually, we have already derived the **Y-combinator**, which is a fixed-point combinator that allows us to define recursive functions in lambda calculus.

```
// Y-combinator:
//   - good lambda program, bad JavaScript program
(g => (h => g(h(h))) (h => g(h(h))))
```

However, the problem is that the above program is not a good JavaScript program. Before stepping into the actual factorial function, it will already cause infinite recursion because of the self-application.

In order to make it an executable JavaScript program, we have to defer the self-application until we actually get the argument value. Here, `g(h(h))` is eagerly applying `h(h)`, which is not what we want. We can change it to `x => g(h(h))(x)`, which means we are deferring the self-application until we get the argument `x`.

```
// Z-combinator:
(g => (h => x => g(h(h))(x)) (h => x => g(h(h))(x)))
  (f => n => n == 0 ? 1 : n * f(n - 1))
```

Voila! We have **successfully derived the Z-combinator in lambda calculus** that allows us to define recursive functions and can work in JavaScript.

Aesthetically though, the combinator looks very cryptic. But typical usage of lambda calculus allows us to hide the combinator in a library. The complexity of the recursion is hidden in the combinator, and the recursive function looks pretty nice.

```
Z = g => (h => x => g(h(h))(x)) (h => x => g(h(h))(x))
```

To use the combinator, we just need to pass in the body of the recursive function.

```
fac = Z(f => n => n == 0 ? 1 : n * f(n - 1))
fac(10) // 3628800
```

You might ask why we have to go through all this trouble to remove variable assignment, all-the-while ended up with a variable assignment. But the point is that there is no name-binding in the body of the function (i.e., during recursion, the name of the function cannot be used or appear inside the function).

This is still a significant achievement! We can try to use the same combinator to define other recursive functions, such as Fibonacci numbers!

```
fib = Z(f => n => n <= 1 ? 1 : f(n - 1) + f(n - 2))
fib(10) // 89
```