CS302: Paradigms of Programming

Lab 2: Church Numerals

March 9^{th}, 2021

Recall the claim the instructor had made that everything apart from lambdas is syntactic sugar. Let us get a flavour of the same.

Say we define the first number `zero` as:

```
(define zero (lambda () '()))
```

That is, zero is represented just as a lambda that returns an empty list.

Say we additionally have the following two definitions for computing the successor and the predecessor of a given number `x`:

```
(define (succ x) (lambda () x))     ; i.e., wrap x in a lambda
(define (pred x) (x))               ; i.e., unwrap x by calling that lambda
```

The purpose of the above two clever definitions is to ensure the property that the predecessor of the successor of a number n is n. Make sure you see this in the definitions (in terms of when is a procedure applied versus defined, etc.).

What is the number one? Successor of zero! What is the number two? Successor of one (or successor of successor of zero)! And so on. Let's define a few numbers this way:

```
(define one (succ zero))
(define two (succ one))
(define three (succ two))
(define four (succ three))
(define five (succ four))
```

Now say we have a way to compute if something is zero:

```
(define (is-zero? x) (null? (x)))
```

(works because `null?` checks for empty list.)

Next, let us define a procedure that checks if two of our newly defined ***Church numerals*** are equal:

```
(define (is-equal? x y)
  (cond ((is-zero? x) (is-zero? y))
    ((is-zero? y) (is-zero? x))
    (else (is-equal? (pred x) (pred y)))))
```

Basically, as the only actual equality check we have is for zeros (in the form of `is-zero?`), what we are doing in `(is-equal? x y)` is getting down to using `is-zero?` by recursively reducing `x` and `y` using the `pred` function.

Write code to verify the following kinds of properties:

1. `zero` is not equal to `one`, but is equal to `zero`
2. `four` is equal to `succ(succ(succ(succ(zero))))`
3. The predecessor of the successor of `two` is `two`

Apart from checking for equality, what else are numbers useful for? Addition, subtraction, multiplication, and so on. Understand and programmatically verify (like the items above) that the following `add-church` procedure works:

```
(define (add-church x y)
  (if (is-zero? y)
    x
    (add-church (succ x) (pred y))))
```

Now write your own versions of `subtract-church` and `multiply-church`.

Absorb the fact that we were able to implement *numbers* and (some of) the important associated operations just with *lambdas*; that should give you an idea of the power of the **lambda calculus** (and its sugarless world :p).

If you found this interesting, go ahead and look up how could you define booleans (a smaller set than numbers, isn't it?) and operations over booleans using lambdas, leading to the notion of *Church booleans*.