CS302: Paradigms of Programming

Lab 1: Functional Programming with Numbers

February 29^{th}, 2020

Q1. Compute the value of the following expression using Scheme:

$$\frac{5 + 4 + (2 - (3 - (6 + \frac{4}{5})))}{3(6-2)(2-7)}$$

Q2. Define a function `leastTwo` that returns the sum of the smallest two of its three inputs. For instance:

```
(leastTwo 3 1 2) = 3
(leastTwo 8 8 3) = 11
```

Q3. Given two integers x and n and an integer exponent y, write a function `(modexp x y n)` that will output $x^y \ mod \ n$.

Q4. Recall whom did we agree to have invented perhaps the largest number of things named after him. Apart from the square-root method (using `good-enough`, `improve`, and an initial `guess`), he had also invented a method to compute cube-roots. The method is based on the fact that if y is an approximation to the cube root of x, then a better approximation is given by:

$$\frac{x/y^2 + 2y}{3}$$

Use this formula to implement a cube-root function analogous to the square-root procedure.

Q5. Write a function `iterative-improve` that takes two functions as arguments: a method for telling whether a guess is good enough and a method for improving a guess. The function `iterative-improve` should return as its value another function that takes a guess as argument and keeps improving the guess until it is good enough. Rewrite the `cube-root` function from Q3 in terms of `iterative-improve`.

Go to Page 2 only after finishing with Page 1.

Q6. Recall the claim the instructor made in the previous class 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 () '()))
```

where, for now, say `'()` represents *empty*.

Say we additionally have the following two definitions for computing the successor and the predecessor functions:

```
(define (succ x) (lambda () x))
(define (pred x) (x))
```

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)))
```

where `null?` is a predefined procedure in Scheme.

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* (recall the representation of our 'zero'); that should give you an idea of the power of the **lambda calculus** – a topic we would discuss further in the next week!