Homework set 21: Programming in the λ-calculus — due Friday 4 May

Total number of points available on this homework is 200. Full credit is equivalent to 0 points. In other words, this homework set is entirely optional, but it is highly recommended.

The following definitions are quite like the λ-expressions your interpreter from Project 1 should be able to process, except that some of them are recursive.

Can your interpreter actually handle recursion directly? You may wish to make that possible, so that you don't need to use the **Y** combinator all the time. Instead, you must make it possible for the interpreter to expand names ("ζ-conversion") repeatedly.

If you build the version of the interpreter that supports recursive definitions, then you should be able to evaluate the definitions below directly.

Regardless of whether you do that, you should also rewrite the definitions below using the **Y** combinator, without recursive definitions.

Notation: # stands for λ, and $ stands for λ on strict arguments. (If you haven't implemented strictness annotations, you should still be able to evaluate the terms in normal order, ignoring the annotations, though it may require a larger number of reduction steps.)

Note that some of these definitions are slightly different from the definitions for the corresponding arithmetic operations that we used in class. You must use exactly these definitions.

```
definitions
        Plus is $x.$y.#p.#q.(x p (y p q));
        Times is $x.$y.$f.(y (x f));
        Exp is $x.$y.(y x);

        Succ is $x.$y.#z.(y (x y z));
        + is Succ;
        True is #x.#y.x;
        False is #x.#y.y;
        Not is $b.#x.#y.(b y x);
        And is $b1.$b2.(b1 b2 False);

        IsZero is $k.(k (True False) True);
        Fact is $n.(IsZero n
                    1
                    (Times n (Fact (Pred n)))
                   );
        Pred is $k.(k
                    (#p.#u.
                     (u
                      (Succ (p True))
                      (p True)
                     )
                    )
                    (#u.(u 0 0))
                    False
                   );

        Over is $n.$k.( (= k 0) 1
                      ( (= k n) 1
```

```
                        (     (Over (Pred n) (Pred k))
                           +  (Over (Pred n) k)
                        )
                      ));

        Raise is $x.$n.( (= x 1) 1
                       ( (And (= n 0) (> x 0)) 1
                         (Sum 0 n
                           $k.(Times (Over n k)
                                     (Raise (Pred x) k)
                              )
                         )
                       ));

        Sum is $a.$b.#e.((> a b) 0
                            (Plus (e a)
                                  (Sum (Succ a) b e)
                             )
                         );

        >= is $x.$y.((IsZero x)(IsZero y)
                     (>= (Pred x)(Pred y)));

        = is $x.$y.((>= x y)(>= y x) False);

        > is $x.$y.((>= x y)  (Not (>= y x))
                           False);

        0 is #x.#y.y;
        1 is #x.#y.(x y);
        2 is #x.#y.(x (x y));
        3 is #x.#y.(x (x (x y)));
        4 is #x.#y.(x (x (x (x y))));

        SelfRaise is #x.(Raise x x);
        answer is (Sum 1 4 SelfRaise)
enddefinitions
```

Tasks:

1. (10 pts.) Write in conventional mathematical notation the computation defined above.
2. (90 pts.) Make sure that your intepreter is able to reduce the term `answer`. What is the normal form of `answer`? Which integer does this Church numeral represent?
3. (50 pts.) What is the running time (expressed as number of conversions and as CPU time in seconds) for the reduction of the expression (`SelfRaise` *n*), where for *n* we put Church numerals for 0, 1, 2, 3, 4, ... (go as high as you can depending on the speed of your computer and interpreter, but you must be able to go at least up to 4).

   If you have several different versions of the interpreter, compare their speed.
4. (50 pts.) If you have implemented strictness annotations, explore which arguments can safely be annotated as strict, and how that affects evaluation speed.