Natural Numbers as Church Numerals

Natural numbers are non-negative. Given a successor function, next, which adds one, we can define the natural numbers in terms of zero and next:

```
    1 = (next 0)
    2 = (next 1)
      = (next (next 0))
    3 = (next 2)
      = (next (next (next 0)))
```

and so on. Therefore a number n will be that number of successors of zero. Just as we adopted the convention TRUE = first, and FALSE = second, we adopt the following convention:

```
     zero  = Lf.Lx.x
     one   = Lf.Lx.(f x)
     two   = Lf.Lx.(f (f x))
     three = Lf.Lx.(f (f (f x)))
     four  = Lf.Lx.(F (f (f (f x))))
```

Therefore we have a "unary" representation of the natural numbers, such that n is represented as n applications of the function f to the argument x. This representation is refered to as CHURCH NUMERALS.

We can define the function next as follows:

```
  next = Ln.Lf.Lx.(f ((n f) x))
```

and therefore one as follows:

```
  one = (next zero)
      => (Ln.Lf.Lx.(f ((n f) x)) zero)
      => Lf.Lx.(f ((zero f) x))
      => Lf.Lx.(f ((Lg.Ly.y f) x)) (* alpha conversion avoids clash *)
      => Lf.Lx.(f (Ly.y x))
      => Lf.Lx.(f x)
```

and two as follows:

```
  two = (next one)
      => (Ln.Lf.Lx.(f ((n f) x)) one)
      => Lf.Lx.(f ((one f) x))
      => Lf.Lx.(f ((Lg.Ly.(g y) f) x)) (* again, alpha conversion *)
      => Lf.Lx.(f (Ly.(f y) x)
      => Lf.Lx.(f (f x))
```

```
val next = fn n => fn f => fn x => (f ((n f) x));
```

NOTE that ((two g) y) = (g (g y)). So if we had some function, say one that increments n:

```
   inc = Ln.(n+1)
```

then we can get a feel for a Church Numeral as follows:

```
   ((two inc) 0)
   => ((Lf.Lx.(f (f x)) inc) 0)
   => (Lx.(inc (inc x) 0)
   => (inc (inc 0))
   => (Ln.(n+1) (Ln.(n+1) 0))
   => (Ln.(n+1) (0 + 1))
   => ((0 + 1) + 1)
   => 2
```

We are now in a position to define addition in terms of next:

```
add = Lm.Ln.Lf.Lx.((((m next) n) f) x);
```

Therefore four may be computed as follows:

```
four = ((add two) two)
     => ((Lm.Ln.Lf.Lx.((((m next) n) f) x) two) two)
     => (Ln.Lf.Lx.((((two next) n) f) x)
     => Lf.Lx.((((two next) two) f x)
     => Lf.Lx.((((Lg.Ly.(g (g y)) next) two) f x)
     => Lf.Lx.(((Ly.(next (next y)) two) f) x)
     => Lf.Lx.(((next (next two)) f) x)
     => Lf.Lx.(((next (Ln.Lf.Lx.(f ((n f) x)) two)) f) x)
```

Multiplication is defined as follows:

```
mult = Lm.Ln.Lx.(m (n x))

six  = ((mult two) three)
     => ((Lm.Ln.Lx.(m (n x)) two) three)
     => (Ln.Lx.(two (n x) three)
     => Lx.(two (three x))
     => Lx.(two (Lg.Ly.(g (g (g y))) x))
     => Lx.(two Ly.(x (x (x y))))
     => Lx.(Lf.Lz.(f (f z)) Ly.(x (x (x y))))
     => Lx.Lz.(Ly.(x (x (x y))) (Ly.(x (x (x y))) z))
     => Lx.Lz.(Ly.(x (x (x y))) (x (x (x z))))
     => Lx.Lz.(x (x (x (x (x (x z))))))
```

And exponentiation as follows:

```
 power = Lm.Ln.(m n);

 nine  = ((power two) three)
       => ((Lm.Ln.(m n) two) three)
       => (Ln.(two n) three)
       => (two three)
       => (Lf.Lx.(f (f x)) three)
       => Lx. (three (three x))
       => Lx. (three (Lg.Ly.(g (g (g y))) x))
       => Lx. (three Ly.(x (x (x y))))
       => Lx. (Lg.Lz.(g (g (g z))) Ly.(x (x (x y))))
       => Lx.Lz.(Ly.(x (x (x y)))
                 (Ly.(x (x (x y)))
                   (Ly.(x (x (x y))) z)))
       => Lx.Lz.(Ly.(x (x (x y)))
                 (Ly.(x (x (x y)))
                   (x (x (x z)))))
       => Lx.Lz.(Ly.(x (x (x y)))
                   (x (x (x (x (x (x z)))))))
       => Lx.Lz.(x (x (x (x (x (x (x (x (x z))))))))))
```

The following lambda function tests for zero:

```
third = Lx.Ly.Lz.z
iszero = Ln.((n third) first)
```

Another approach:

```
T = Lx.Ly.x
F = Lx.Ly.y
af = Lx.F
zp = Ln.((n af) T)
```

Here we take a function "af" that always returns "false".

Since a church numeral is a function of two arguments, we
apply the number to be tested to "af" as the first argument
and to "T" as the second argument.

If the number being tested is zero (which is the same as "F"), then
it selects the second argument and so returns "T".

If the number is not zero then it will behave as a church
numeral and apply the function some number of times...
and the function it applies is a function that returns "F" in all
circumstances... no matter how many times it's applications are nested.

Let's try this for a test of two:

```
(zp two)  -->  (L n ((n af) T) two) --> ((two af) T)
          -->  ( (L g L y (g (g y)) af) T )
          -->  ( L y (af (af y)) T )
          -->  ( af ( af T ) )
          -->  ( L x F ( L x F T ) )
          -->  F
```

The lambda function pred delivers the predecessor of a
Church Numeral:

```
pair = Lx.Ly.Lf.((f x) y);
prefn = Lf.Lp.((pair (f (p first))) (p first))
pred = Ln.Lf.Lx.(((n (prefn f)) (pair x x)) second)
```

It should be of interest to note the following. A major
landmark in Lambda Calculus occurred in the 1930's when
Kleene discovered how to express the operation of
subtraction within Church's scheme (yes, that means that even
though Church invented/discovered the Lambda Calculus he was
unable to implement subtraction and subsequently division, within
that calculus)! Other landmarks then followed, such as the recursive
function Y. In 1937 Church and Turing, independently, showed that every
computable operation (algorithm) can be achieved in a Turing machine
and in the Lambda Calculus, and therefore the two are equivalent.
Similarly Godel introduced his description of computability, again
independently, in 1929, using a third approach which was again shown
to be equivalent to the other 2 schemes.

It appears that there is a "platonic reality" about
computability. That is, it was "discovered" (3 times idepenedently)
rather than "invented". It appears to be natural in some
sense.

(********** ml source **********)

```
(* Church Numerals *)

val zero = fn f => fn x => x;
val next = fn n => fn f => fn x => (f ((n f) x));
val add = fn m => fn n => fn f => fn x => ((((m next) n) f) x);
val mult = fn f => fn g => fn x => (f (g x));
val power = fn f => fn g => (f g);

(* Church Numerals *)
val one = (next zero);
val two = (next one);

val inc = fn n => n+1;

(* More Church Numerals *)
val four = (add one (add one (add one (add one zero))));
```

```
val three = ((add one) two);
val five = ((add three) two);
val six = ((mult three) two);
val nine = ((power two) three);
val ten = ((mult two) five);

(* A test to show what a Church Numeral does *)

((zero inc) 0);
((five inc) 0);
((ten inc) 0);

val first = fn x => fn y => x;
val second = fn x => fn y => y;
val third = fn x => fn y => fn z => z;

val iszero = fn n => ((n third) first);
(* Predicate, is a Church Numeral zero? *)

val T = fn x => fn y => x ;
val F = fn x => fn y => y ;
val af = fn x => F;
val zp = fn n => ((n af) T);  (* zp is a zero predicate *)

val pair = fn x => fn y => fn f => ((f x) y);

val prefn = fn f => fn p => ((pair (f (p first))) (p first));
val pred = fn n => fn f => fn x => (((n (prefn f))(pair x x)) second);
(* predecessor function *)

val eight = (pred nine);
```