Encoding natural numbers

```
datatype nat = Z | S of nat
val zero  = Z
val one   = S Z
val two   = S (S Z)
val three = S (S (S Z))
```

Define `fold` **s.t.** `fold` f x **replaces** `S` $\mapsto f$ **and** `Z` $\mapsto x$

```
fun fold f x Z     = x
  | fold f x (S n) = f (fold f x n)
```

Example: `fold (fn k => k+1) 0 three` $\Downarrow$ `3`

Church: represent n **as** $\lambda f.\lambda x.$`fold` f x n.

Church Numerals

Encoding natural numbers as lambda-terms

$$
\begin{aligned}
\text{zero} &= \lambda f.\lambda x.x \\
\text{one} &= \lambda f.\lambda x.f\ x \\
\text{two} &= \lambda f.\lambda x.f(f\ x) \\
\text{succ} &= \lambda n.\lambda f.\lambda x.f(n\ f\ x) \\
\text{plus} &= \lambda n.\lambda m.n\ \text{succ}\ m \\
\text{times} &= \lambda n.\lambda m.n\ (\text{plus}\ m)\ \text{zero}
\end{aligned}
$$

(Code follows)

Church Numerals to machine integers

```
; uscheme or possibly uhaskell
-> (define to-int (n)
             ((n ((curry +) 1)) 0))
-> (to-int three)
3
-> (to-int ((times three) four))
12
```

Church Numerals in λ

```
zero  = \f.\x.x;
succ  = \n.\f.\x.f (n f x);
plus  = \n.\m.n succ m;
times = \n.\m.n (plus m) zero;
 ...
-> four;
\f.\x.f (f (f (f x)))
-> three;
\f.\x.f (f (f x))
-> times four three;
\f.\x.f (f (f (f (f (f (f (f (f (f (f (f x))))))))))))
```