```haskell
-- RUN WITH "hugs -98 +O" for overlapping instances!


module NumAlg where

import Prelude hiding (succ, pred)      -- we'll re-define succ and pred
import Char (ord, chr)                  -- ASCII character conversions
import ShortParse                       -- abbreviated parsing library



----------   bits, binary numerals and "semantics"

data Bit = B0 | B1  deriving (Eq, Ord, Enum, Show)

bit '0' = B0
bit '1' = B1
bit  c  = error "bad bit literal"

type Binary = [Bit]

bin [] = error "empty binary numeral"
bin cs = map bit cs

semB  = fromEnum :: Bit -> Int
semBN = foldl (\v b -> 2 * v + semB (bit b)) 0
```

```haskell
----------   abstract algebra for Peano numerals

class Peano p where
    zero :: p
    succ, pred :: p -> p
    eqzero :: p -> Bool

semP s z n | eqzero n  = z
           | otherwise = s (semP s z (pred n))



----------   Integer instance of Peano

instance Peano Integer where
    zero = 0
    succ = (+) 1
    pred = \i -> max (i-1) 0
    eqzero = (==) 0



----------   Syntactic natural numbers, and instance of Peano

data Nat = Zero | Succ Nat  deriving (Eq, Ord, Show)

instance Peano Nat where
    zero = Zero
    succ = Succ
    eqzero = (==) Zero
    pred (Succ n) = n
    pred  Zero    = Zero
```

```haskell
----------   Unary notation instance of Peano

type Unary = [()]

instance Peano Unary where
    zero = []
    succ = (():)
    pred = drop 1
    eqzero = (==) []

uni2int = length
int2uni = (`replicate` ())



----------   Church numeral instance of Peano

newtype Church a = Church (forall a. (a -> a) -> (a -> a))

instance Peano (Church a) where
  zero              = Church (\f x -> x)
  succ   (Church n) = Church (\f x -> f (n f x))
  pred   (Church n) = Church (\f x -> n (\g h -> h (g f)) (\u -> x) (\u -> u))
  eqzero (Church n) = n (const False) True


-- could also directly implement +, * in Church numerals and prove equivalent

instance Show (Church a) where
  show (Church n) = "\\f x -> " ++ n ("(f "++) "x" ++ n (')':) ""
```

```haskell
----------   conversion conveniences

peano   n = semP succ zero (n :: Integer)
integer n = (peano n) :: Integer
nat     n = (peano n) :: Nat
unary   n = (peano n) :: Unary
church  n = (peano n) :: Church a

chapp n = g where Church g = church n



----------   abstract algebra for semirings, and instances

class SemiRing r where
    none, one :: r
    add, mul  :: r -> r -> r

exp n m = semP (mul n) one m

instance Peano p => SemiRing p where
    none = zero
    one  = succ zero
    add n m = semP succ n m
    mul n m = semP (add n) zero m


instance SemiRing Unary where
    none = []
    one  = [()]
    add  = (++)
    mul  = concatMap . const
```

```haskell
----------   binary operator algebras, semiring operators

data BopAlg n b = Lit n
                | Bop b (BopAlg n b) (BopAlg n b)  deriving (Eq, Ord, Show)

data SROpr = Add | Mul  deriving Show

type SRAlg n = BopAlg n SROpr

-- can't get instance Peano p => Peano (SRAlg p): no predecessor!



----------   semiring and operator semantics

semBA f g = s
            where s (Lit n)     = f n
                  s (Bop b l r) = g b (s l) (s r)

semSRO a m Add = a
semSRO a m Mul = m

eval l a m = semBA l (semSRO a m)

sreval :: (SemiRing a, Peano a) => BopAlg Integer SROpr -> a
sreval = eval peano add mul
```

```
----------   parsing for BopAlg Integer SROpr

expr = term `chainl1` opr '+' Add
term = fact `chainl1` opr '*' Mul
fact = intlit +++ paren expr

intlit  = do { i <- token int; return (Lit i) }
opr c f = do { lit c; return (Bop f) }

paren p = bracket (lit '(') p (lit ')')


----------   unparsing for BopAlg Integer SROpr

unparse fix = semBA show (fix . semSRO "+" "*")

infx o l r = par (unwords [l,o,r])
prfx o l r = unwords [o,l,r]
pofx o l r = unwords [l,r,o]

par s = "(" ++ s ++ ")"


----------   testing

test = parse expr " ( 2 + 1 ) * 4 "

chu12 = sreval test :: Church a
nat12 = sreval test :: Nat
int12 = sreval test :: Integer
uni12 = sreval test :: Unary

intest = unparse infx test
prtest = unparse prfx test
potest = unparse pofx test
```