```haskell
-- Basic Syntax:
circ r = let l = 2*pi * r               (l,A) where l = 2*pi*r
             A = pi * r * r in (l,A)                A = pi*r*r -- let & where
not x = case x of                       sgn x | x > 0 = 1
        True → False                          | otherwise = -1 -- guarded equations
        False → True -- case-of
f (g x) = f $ g x,function composition: (.) f (g x) = (f.g) x,lambda: \x y → x + y …
factors n = [ x | x ← [1..n], mod n x == 0 ] -- list comprehension
type: alias. newtype: construct type with only 1 constructor & parameter. data: otherwise
-- Some functions in Prelude:
map,foldl,foldl',filter,zip,zipWith,all,any
takeWhile,dropWhile,take,drop,head([0]),tail([1:]),last([-1]),fst,snd
-- Functor & Applicative & Monoid & Foldable & Traversable:
class Functor f where
    fmap :: (a→b) → f a → f b ; (<$>) = fmap
class Functor f ⇒ Applicative f where
    pure :: a → f a
    (<*>) :: f (a→b) → f a → f b -- pure _ <*> = _ <$>
class Applicative f ⇒ Monad f where
    (>>=) :: f a → (a → f b) → f b ; return = pure ;
    (>>) :: m a → m b → m b; m >> k = m >>= \_ → k
do { x ← op1; op2 } := op1 >>= \x → op2; do { op1; op2 } := op1 >> op2
class Applicative f ⇒ Alternative f where
    (<|>) :: f a → f a → f a; empty :: f a
    many :: f a → f [a]; many v = some v <|> pure [] -- Zero or more.
    some :: f a → f [a]; some v = (:) <$> v <*> many v -- One or more.
class Semigroup a where
    (<>) :: a → a → a
class Semigroup a ⇒ Monoid a where
    mempty :: a; mappend = (<>); mconcat :: [a] → a -- fold list using monoid
fold :: Monoid a ⇒ [a] → a
fold [] = mempty; fold (x:xs) = x <> fold xs
class Foldable t where
    fold :: Monoid a ⇒ t a → a
    foldMap :: Monoid b ⇒ (a→b) → t a → b
    foldl :: (b→a→b) → b → t a → b; foldr :: (a→b→b) → b → t a → b
traverse :: (a → Maybe b) → [a] → Maybe [b]
traverse g [] = pure []; traverse g (x:xs) = pure (:) <*> g x <*> traverse g xs
class (Functor t, Foldable t) ⇒ Traversable t where
    traverse :: Applicative f ⇒ (a → f b) → t a → f (t b)
sequenceA :: Applicative f ⇒ t (f a) → f (t a)
sequenceA = traverse id; traverse g = sequenceA . fmap g
-- Fix: (acts like Y combinator)
fix :: (a → a) → a; fix f = x where x = f
fac = fix $ \rec n → if n ≤ 1 then 1 else n * rec $ n-1 -- write non-recursively
```