```haskell
-- Haskell Curry's combinators S, B and I
s f g x = f x (g x)
b f g x = f (g x)
i x = x
```

```haskell
-- Raymond Smullyan's combinator bird names
starling = s
blueBird = b
idiot = i
```

```haskell
-- Alternative combinator names
distributor = s
compositor = b
identity = i
```

Bluebird	$B\ f\ g\ x = f\ (g\ x)$	Compositor	B	fmap
Kestrel	$K\ x\ y = x$	Cancellator	K	pure
Starling	$S\ f\ g\ x = f\ x\ (g\ x)$	Distributor	S	(<*>)

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```haskell
instance Functor ((->) r) where
  fmap = (.)
```

```haskell
class Functor f => Applicative f where
   pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

```haskell
instance Applicative ((->) r) where
  pure x = (\_ -> x)
  f <*> g = \x -> f x (g x)
```

recursive definition

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) = if (p x) then x:(filter p xs) else filter p xs
```

folding

```haskell
filter p = foldr (\x -> if p x then (:) x else id) []
```

folding with combinators

```haskell
filter p = foldr (s (b (bool i) (:)) p) []
```

Curry's combinator names

```haskell
-- returns x when c is False
-- and y when c is True
bool x y c = if c
             then y
             else x
```

```haskell
filter p = foldr (starling (blueBird (bool idiot) (:)) p) []
```

Smullyan's combinator names

```haskell
filter p = foldr (distributor (compositor (bool identity) (:)) p) []
```

Alternative combinator names

folding with applicative functor and identity function

```haskell
filter p = foldr ((<*>) ((.) (bool id) (:)) p) []
```

prefix function invocation

```haskell
filter p = foldr (((bool id) . (:)) <*> p) []
```

infix function invocation

```haskell
filter p = foldr (bool id . (:) <*> p) []
```

no superfluous parentheses