Haskell

functional

functional → **λx . x (λy . y)**

Haskell

functional

λx . x (λy . y)

```
\x -> x (\y -> y)
```

Haskell

functional

```
def f = λx .  x (λy .  y)
```

```
\x -> x (\y -> y)
```

Haskell

functional

```
def f = λx .  x (λy .  y)
```

```
f = \x -> x (\y -> y)
```

Haskell

functional

```
def f = λx .  x (λy .  y)
```

strongly-typed

```
f = \x -> x (\y -> y)
```

Haskell

functional

```
def f = λx .  x (λy .  y)
```

strongly-typed

```
f = \x -> x (\y -> y)
```

Haskell

functional

def f = λx . x (λy . y)

strongly-typed

```
f :: forall a b.
     ((b -> b) -> a) -> a
f = \x -> x (\y -> y)
```

Haskell

functional

```
def f = λx .  x (λy .  y)
```

strongly-typed

```haskell
f :: ((b -> b) -> a) -> a

f = \x -> x (\y -> y)
```

Haskell

functional

```
def f = λx .  x (λy .  y)
```

strongly-typed

```
f :: ((b -> b) -> a) -> a

f x = x (\y -> y)
```

Haskell

functional

```
def f = λx .  x (λy .  y)
```

strongly-typed

```
f :: ((b -> b) -> a) -> a

f x = x (\y -> y)
```

```
dup' :: a -> (a, a)
dup' x = (x, x)
```

Haskell

functional

def f = λx . x (λy . y)

strongly-typed f :: ((b -> b) -> a) -> a

f x = x (\y -> y)

```
dup' :: a -> (a, a)
dup' x = (x, x)

dup' "hi"
> ("hi", "hi")
```

Haskell

functional

```
def f = λx .  x (λy .  y)
```

strongly-typed

```
f :: ((b -> b) -> a) -> a

f x = x (\y -> y)
```

```
dup' :: a -> (a, a)
dup' x = (x, x)


dup' "hi"
> ("hi", "hi")

fst :: (a, b) -> a
fst (x, _) = x
```

Haskell

functional

def f = λx . x (λy . y)

strongly-typed

```
f :: ((b -> b) -> a) -> a

f x = x (\y -> y)
```

```
dup' :: a -> (a, a)
dup' x = (x, x)
```

cons

nil

```
dup' "hi"
> ("hi", "hi")
```

```
fst :: (a, b) -> a
fst (x, _) = x
```

Haskell

functional

def f = λx . x (λy . y)

strongly-typed

```
f :: ((b -> b) -> a) -> a

f x = x (\y -> y)
```

```
dup' :: a -> (a, a)
dup' x = (x, x)
```

cons ⟶ :

nil ⟶ []

```
dup' "hi"
> ("hi", "hi")
```

```
fst :: (a, b) -> a
fst (x, _) = x
```

Haskell

functional

$$\text{def } f = \lambda x .\ x\ (\lambda y .\ y)$$

strongly-typed

```
f :: ((b -> b) -> a) -> a

f x = x (\y -> y)
```

```
dup' :: a -> (a, a)
dup' x = (x, x)
```

cons → :

nil → []

cons 2 (cons 1 nil)

```
dup' "hi"
> ("hi", "hi")
```

```
fst :: (a, b) -> a
fst (x, _) = x
```

Haskell

functional

```
def f = λx .  x (λy .  y)
```

strongly-typed

```
f :: ((b -> b) -> a) -> a

f x = x (\y -> y)
```

```
dup' :: a -> (a, a)
dup' x = (x, x)

dup' "hi"
> ("hi", "hi")

fst :: (a, b) -> a
fst (x, _) = x
```

```
cons ⟶ :
nil ⟶ []
cons 2 (cons 1 nil)
⟶ 2:(1:[])
```

Haskell

functional

def f = λx . x (λy . y)

strongly-typed

```
f :: ((b -> b) -> a) -> a

f x = x (\y -> y)
```

```
dup' :: a -> (a, a)
dup' x = (x, x)
```

cons ⟶ :

nil ⟶ []

cons 2 (cons 1 nil)

⟶ 2:(1:[])

```
dup' "hi"
> ("hi", "hi")
```

```
fst :: (a, b) -> a
fst (x, _) = x
```

Haskell

functional

def f = λx . x (λy . y)

strongly-typed

```
f :: ((b -> b) -> a) -> a

f x = x (\y -> y)
```

```
dup' :: a -> (a, a)
dup' x = (x, x)

dup' "hi"
> ("hi", "hi")

fst :: (a, b) -> a
fst (x, _) = x
```

cons ⟶ :

nil ⟶ []

cons 2 (cons 1 nil)

⟶ **2:(1:[])**

[2, 1]

Haskell

functional

```
def f = λx .  x (λy .  y)
```

strongly-typed

```
f :: ((b -> b) -> a) -> a

f x = x (\y -> y)
```

```
dup' :: a -> (a, a)
dup' x = (x, x)

dup' "hi"
> ("hi", "hi")

fst :: (a, b) -> a
fst (x, _) = x
```

```
cons → :
nil → []
cons 2 (cons 1 nil)
→ 2:(1:[])
  [2, 1]
```

```
sum :: [Int] -> Int
```

Haskell

functional

def f = λx . x (λy . y)

strongly-typed

```
f :: ((b -> b) -> a) -> a

f x = x (\y -> y)
```

```
dup' :: a -> (a, a)
dup' x = (x, x)
```

cons ⟶ :

nil ⟶ []

cons 2 (cons 1 nil)

⟶ 2:(1:[])

[2, 1]

```
dup' "hi"
> ("hi", "hi")
```

```
sum :: [Int] -> Int
sum (x:y) = x + sum y
```

```
fst :: (a, b) -> a
fst (x, _) = x
```

Haskell

functional

def f = λx . x (λy . y)

strongly-typed

```
f :: ((b -> b) -> a) -> a

f x = x (\y -> y)
```

```
dup' :: a -> (a, a)
dup' x = (x, x)

dup' "hi"
> ("hi", "hi")

fst :: (a, b) -> a
fst (x, _) = x
```

cons ⟶ :

nil ⟶ []

cons 2 (cons 1 nil)

⟶ 2:(1:[])

[2, 1]

```
sum :: [Int] -> Int
sum (x:y) = x + sum y
sum [] = 0
```

Haskell

Haskell

```
type IntList = IntCons Int IntList | IntNil
```

Haskell

```
type IntList = IntCons Int IntList | IntNil
```

```
data IntList = IntCons Int IntList | IntNil
```

Haskell

```
type IntList = IntCons Int IntList | IntNil

data IntList = IntCons Int IntList | IntNil

l :: IntList
l = (IntCons 2 (IntCons 1 IntNil))
```

Haskell

```
type IntList = IntCons Int IntList | IntNil

data IntList = IntCons Int IntList | IntNil

 l :: IntList
 l = (IntCons 2 (IntCons 1 IntNil))
 l = 2 `IntCons` (1 `IntCons` IntNil)
```

Haskell

```haskell
type IntList = IntCons Int IntList | IntNil

data IntList = IntCons Int IntList | IntNil

 l :: IntList
 l = (IntCons 2 (IntCons 1 IntNil))
 l = 2 `IntCons` (1 `IntCons` IntNil)

    plus :: Int -> Int -> Int
    plus x y = x + y
```

Haskell

```
type IntList = IntCons Int IntList | IntNil

data IntList = IntCons Int IntList | IntNil

 l :: IntList
 l = (IntCons 2 (IntCons 1 IntNil))
 l = 2 `IntCons` (1 `IntCons` IntNil)

     plus :: Int -> Int -> Int
     plus x y = x + y

     plus 1 2
     > 3
```

Haskell

```
type IntList = IntCons Int IntList | IntNil

data IntList = IntCons Int IntList | IntNil

 l :: IntList
 l = (IntCons 2 (IntCons 1 IntNil))
 l = 2 `IntCons` (1 `IntCons` IntNil)

     plus :: Int -> Int -> Int
     plus x y = x + y

     plus 1 2
     > 3

     1 `plus` 2
     > 3
```

Haskell

```
type IntList = IntCons Int IntList | IntNil

data IntList = IntCons Int IntList | IntNil

l :: IntList
l = (IntCons 2 (IntCons 1 IntNil))
l = 2 `IntCons` (1 `IntCons` IntNil)
```

Haskell

```
type IntList = IntCons Int IntList | IntNil

data IntList = IntCons Int IntList | IntNil

 l :: IntList
 l = (IntCons 2 (IntCons 1 IntNil))

   intHead :: IntList -> Int
   intHead (IntCons h _) = h
   intHead IntNil = error "empty list"
```

Haskell

```
type IntList = IntCons Int IntList | IntNil

data IntList = IntCons Int IntList | IntNil

 l :: IntList
 l = (IntCons 2 (IntCons 1 IntNil))

   intHead :: IntList -> Int
   intHead (IntCons h _) = h
   intHead IntNil = error "empty list"


    intHead (IntCons 2 IntNil)
    > 2
```

Haskell

```
type IntList = IntCons Int IntList | IntNil

data IntList = IntCons Int IntList | IntNil

 l :: IntList
 l = (IntCons 2 (IntCons 1 IntNil))

   intHead :: IntList -> Int
   intHead (IntCons h _) = h
   intHead IntNil = error "empty list"


    intHead (IntCons 2 IntNil)
    > 2

    intHead IntNil
    *** Exception: empty list
```

Haskell

```
type IntList = IntCons Int IntList | IntNil

data IntList = IntCons Int IntList | IntNil

 l :: IntList
 l = (IntCons 2 (IntCons 1 IntNil))

   intHead :: IntList -> Int
   intHead (IntCons h _) = h
   intHead IntNil = error "empty list"

    intHead (IntCons 2 IntNil)
    > 2

    intHead IntNil
    *** Exception: empty list
```

Haskell

```
type IntList = IntCons Int IntList | IntNil

data IntList = IntCons Int IntList | IntNil

 l :: IntList
 l = (IntCons 2 (IntCons 1 IntNil))
```

```
intHead :: IntList -> Int
intHead (IntCons h _) = h
intHead IntNil = error "empty list"
```

"partial function"

```
intHead (IntCons 2 IntNil)
> 2

intHead IntNil
*** Exception: empty list
```

Haskell

```
type IntList = IntCons Int IntList | IntNil

data IntList = IntCons Int IntList | IntNil

 l :: IntList
 l = (IntCons 2 (IntCons 1 IntNil))
```

```
intHead :: IntList -> Int
intHead (IntCons h _) = h
intHead IntNil = error "empty list"
```

"partial function"

```
data MaybeInt = JustInt Int | NoInt
```

Haskell

```
type IntList = IntCons Int IntList | IntNil

data IntList = IntCons Int IntList | IntNil

 l :: IntList
 l = (IntCons 2 (IntCons 1 IntNil))
```

```
intHead :: IntList -> MaybeInt
intHead (IntCons h _) = JustInt h
intHead IntNil = NoInt
```

"total function"

```
data MaybeInt = JustInt Int | NoInt
```

Haskell

```
data List a = Cons a (List a)
            | Nil
```

```
type IntList = IntCons Int IntList | IntNil
```

```
data IntList = IntCons Int IntList | IntNil
```

```
l :: IntList
l = (IntCons 2 (IntCons 1 IntNil))
```

```
intHead :: IntList -> MaybeInt
intHead (IntCons h _) = JustInt h
intHead IntNil = NoInt
```

"total function"

```
data MaybeInt = JustInt Int | NoInt
```

Haskell

```
data List a = Cons a (List a)
            | Nil
```

```
l :: List String
l = "hello" `Cons` ("world" `Cons` Nil)
```

```
l :: IntList
l = (IntCons 2 (IntCons 1 IntNil))
```

```
intHead :: IntList -> MaybeInt
intHead (IntCons h _) = JustInt h
intHead IntNil = NoInt
```

"total function"

```
data MaybeInt = JustInt Int | NoInt
```

Haskell

```
data List a = Cons a (List a)
            | Nil

l :: List String
l = "hello" `Cons` ("world" `Cons` Nil)

data Maybe a = Just a | Nothing
```

```
intHead :: IntList -> MaybeInt
intHead (IntCons h _) = JustInt h
intHead IntNil = NoInt
```

"total function"

```
data MaybeInt = JustInt Int | NoInt
```

Haskell

```
data List a = Cons a (List a)
            | Nil

l :: List String
l = "hello" `Cons` ("world" `Cons` Nil)

data Maybe a = Just a | Nothing

head :: List a -> Maybe a
head (Cons h _) = h
head Nil        = Nothing
```

Haskell

```
data List a = Cons a (List a)
            | Nil

l :: List String
l = "hello" `Cons` ("world" `Cons` Nil)

data Maybe a = Just a | Nothing

head :: List a -> Maybe a
head (Cons h _) = h
head Nil        = Nothing


head l
> Just "hello"
```

Haskell

```
data List a = Cons a (List a)
            | Nil

l :: List String
l = "hello" `Cons` ("world" `Cons` Nil)

data Maybe a = Just a | Nothing

head :: List a -> Maybe a
head (Cons h _) = h
head Nil        = Nothing


head l
> Just "hello"

head (2 `Cons` (1 `Cons` Nil))
> Just 2
```

Haskell

```haskell
data List a = Cons a (List a)
            | Nil

l :: List String
l = "hello" `Cons` ("world" `Cons` Nil)

data Maybe a = Just a | Nothing

head :: List a -> Maybe a
head (Cons h _) = h
head Nil        = Nothing


head l
> Just "hello"

head (2 `Cons` (1 `Cons` Nil))
> Just 2

head Nil
> Nothing
```

Haskell

```
plus3 x y z = x + y + z
```

Haskell

```
plus3 x y z = x + y + z

:t plus3
> plus3 :: Int -> Int -> Int -> Int
```

Haskell

```
plus3 x y z = x + y + z
```

```
:t plus3
> plus3 :: Int -> Int -> Int -> Int
```

```
:t (plus3 4 2 1)
> (plus3 4 2 1) :: Int
```

Haskell

```
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
```

```
:t plus3
> plus3 :: Int -> Int -> Int -> Int
```

```
:t (plus3 4 2 1)
> (plus3 4 2 1) :: Int
```

Haskell

```
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
```

```
:t plus3
> plus3 :: Int -> Int -> Int -> Int
```

```
:t (plus3 4)
```

```
:t (plus3 4 2 1)
> (plus3 4 2 1) :: Int
```

Haskell

```
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
```

```
:t plus3
> plus3 :: Int -> Int -> Int -> Int
```

```
:t (plus3 4)
     (plus3 4) :: Int -> Int -> Int
     (plus3 4) y z = 4 + y + z
```

```
:t (plus3 4 2 1)
> (plus3 4 2 1) :: Int
```

Haskell

```
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
```

```
:t plus3
> plus3 :: Int -> Int -> Int -> Int
```

```
:t (plus3 4)

     (plus3 4) :: Int -> Int -> Int
     (plus3 4) y z = 4 + y + z

> (plus3 4) :: Int -> Int -> Int
```

```
:t (plus3 4 2 1)
> (plus3 4 2 1) :: Int
```

Haskell

```
plus3 :: Int  -> (Int  ->  Int  ->  Int)
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
```

```
:t plus3
> plus3 :: Int -> Int -> Int -> Int
```

```
:t (plus3 4)

     (plus3 4) :: Int -> Int -> Int
     (plus3 4) y z = 4 + y + z

> (plus3 4) :: Int -> Int -> Int
```

```
:t (plus3 4 2 1)
> (plus3 4 2 1) :: Int
```

Haskell

```
plus3 :: Int  -> (Int  ->  Int  ->  Int)
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
```

```
:t plus3
> plus3 :: Int -> Int -> Int -> Int
```

```
:t ((plus3 4) 2)
```

```
:t (plus3 4 2 1)
> (plus3 4 2 1) :: Int
```

Haskell

```
plus3 :: Int  -> (Int  ->  Int  ->  Int)
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z

:t plus3
> plus3 :: Int -> Int -> Int -> Int

:t ((plus3 4) 2)

     (plus3 4) 2) :: Int -> Int
     (plus3 4) 2) z = 4 + 2 + z

:t (plus3 4 2 1)
> (plus3 4 2 1) :: Int
```

Haskell

```
plus3 :: Int  -> (Int  ->  Int  ->  Int)
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
```

```
:t plus3
> plus3 :: Int -> Int -> Int -> Int
```

```
:t ((plus3 4) 2)

     (plus3 4) 2) :: Int -> Int
     (plus3 4) 2) z = 4 + 2 + z

> ((plus3 4) 2) :: Int -> Int
```

```
:t (plus3 4 2 1)
> (plus3 4 2 1) :: Int
```

Haskell

```
plus3 :: Int  -> (Int  -> (Int  ->  Int))
plus3 :: Int  -> (Int  ->  Int  ->  Int)
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
```

```
:t plus3
> plus3 :: Int -> Int -> Int -> Int
```

```
:t ((plus3 4) 2)

     (plus3 4) 2) :: Int -> Int
     (plus3 4) 2) z = 4 + 2 + z

> ((plus3 4) 2) :: Int -> Int
```

```
:t (plus3 4 2 1)
> (plus3 4 2 1) :: Int
```

Haskell

```
plus3 :: Int  -> (Int  -> (Int  ->  Int))
plus3 :: Int  -> (Int  ->  Int  ->  Int)
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
```

```
:t plus3
> plus3 :: Int -> Int -> Int -> Int
```

```
:t (plus3 4 2)
```

```
:t (plus3 4 2 1)
> (plus3 4 2 1) :: Int
```

Haskell

```
plus3 :: Int  -> (Int  -> (Int  ->  Int))
plus3 :: Int  -> (Int  ->  Int  ->  Int)
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
```

```
:t plus3
> plus3 :: Int -> Int -> Int -> Int
```

```
:t (plus3 4 2)

     (plus3 4 2) :: Int -> Int
     (plus3 4 2) z = 4 + 2 + z
```

```
:t (plus3 4 2 1)
> (plus3 4 2 1) :: Int
```

Haskell

```
plus3 :: Int  -> (Int  -> (Int  ->  Int))
plus3 :: Int  -> (Int  ->  Int  ->  Int)
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
```

```
:t plus3
> plus3 :: Int -> Int -> Int -> Int
```

```
:t (plus3 4 2)

     (plus3 4 2) :: Int -> Int
     (plus3 4 2) z = 4 + 2 + z

> (plus3 4 2) :: Int -> Int
```

```
:t (plus3 4 2 1)
> (plus3 4 2 1) :: Int
```

Haskell

```
plus3 :: Int  ->  Int  -> (Int  ->  Int)
plus3 :: Int  -> (Int  -> (Int  ->  Int))
plus3 :: Int  -> (Int  ->  Int  ->  Int)
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
```

```
:t plus3
> plus3 :: Int -> Int -> Int -> Int
```

```
:t (plus3 4 2)

     (plus3 4 2) :: Int -> Int
     (plus3 4 2) z = 4 + 2 + z

> (plus3 4 2) :: Int -> Int
```

```
:t (plus3 4 2 1)
> (plus3 4 2 1) :: Int
```

Haskell

```
plus3 :: Int  ->  Int  -> (Int  ->  Int)
plus3 :: Int  -> (Int  -> (Int  ->  Int))
plus3 :: Int  -> (Int  ->  Int  ->  Int)
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
```

Haskell

```
plus3 :: Int  ->  Int  -> (Int  ->  Int)
plus3 :: Int  -> (Int  -> (Int  ->  Int))
plus3 :: Int  -> (Int  ->  Int  ->  Int)
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
plus3 :: Int  -> (Int  ->  Int) ->  Int
```

Haskell

```
plus3 :: Int -> Int -> (Int -> Int)
plus3 :: Int -> (Int -> (Int -> Int))
plus3 :: Int -> (Int -> Int -> Int)
plus3 :: Int -> Int -> Int -> Int
plus3 x y z = x + y + z
plus3 :: Int -> (Int -> Int) -> Int
         x
```

Haskell

```
plus3 :: Int -> Int -> (Int -> Int)
plus3 :: Int -> (Int -> (Int -> Int))
plus3 :: Int -> (Int -> Int -> Int)
plus3 :: Int -> Int -> Int -> Int
plus3 x y z = x + y + z
plus3 :: Int -> (Int -> Int) -> Int
         x      y
```

Haskell

```
plus3 :: Int  ->  Int  -> (Int  ->  Int)
plus3 :: Int  -> (Int  -> (Int  ->  Int))
plus3 :: Int  -> (Int  ->  Int  ->  Int)
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
plus3 :: Int  -> (Int  ->  Int) ->  Int
          x            y          z???
```

Haskell

```
plus3 :: Int  ->  Int  -> (Int  ->  Int)
plus3 :: Int  -> (Int  -> (Int  ->  Int))
plus3 :: Int  -> (Int  ->  Int  ->  Int)
plus3 :: Int  ->  Int  ->  Int  ->  Int
plus3 x y z = x + y + z
```

Haskell

```
add :: Float -> Float -> Float
add x y = x + y
```

Haskell

```
add :: Float -> Float -> Float
add x y = x + y

mul :: Float -> Float -> Float
mul x y = x * y
```

Haskell

```
add :: Float -> Float -> Float
add x y = x + y

mul :: Float -> Float -> Float
mul x y = x * y
```

Haskell

```
add :: Float -> Float -> Float
add x y = x + y

mul :: Float -> Float -> Float
mul x y = x * y

sale :: Float -> Float
sale = mul 0.8
```

Haskell

```haskell
add :: Float -> Float -> Float
add x y = x + y

mul :: Float -> Float -> Float
mul x y = x * y

sale :: Float -> Float
sale = mul 0.8

coupon :: Float -> Float
coupon = add (-5.00)
```

Haskell

```
add :: Float -> Float -> Float
add x y = x + y

mul :: Float -> Float -> Float
mul x y = x * y

sale :: Float -> Float
sale = mul 0.8

coupon :: Float -> Float
coupon = add (-5.00)

tax :: Float -> Float
tax = mul 1.2
```

Haskell

```
add :: Float -> Float -> Float
add x y = x + y
mul :: Float -> Float -> Float
mul x y = x * y
sale :: Float -> Float
sale = mul 0.8
coupon :: Float -> Float
coupon = add (-5.00)
tax :: Float -> Float
tax = mul 1.2
cardFee :: Float -> Float
cardFee x = if x < 30 then x else (add 2.00 x)
```

Haskell

```
add :: Float -> Float -> Float
add x y = x + y
mul :: Float -> Float -> Float
mul x y = x * y
sale :: Float -> Float
sale = mul 0.8
coupon :: Float -> Float
coupon = add (-5.00)
tax :: Float -> Float
tax = mul 1.2
cardFee :: Float -> Float
cardFee x = if x < 30 then x else (add 2.00 x)

    finalPrice :: Float -> Float
    finalPrice i = cardFee (tax (coupon (sale i)))
```

Haskell

```haskell
add :: Float -> Float -> Float
add x y = x + y
mul :: Float -> Float -> Float
mul x y = x * y
sale :: Float -> Float
sale = mul 0.8
coupon :: Float -> Float
coupon = add (-5.00)
tax :: Float -> Float
tax = mul 1.2
cardFee :: Float -> Float
cardFee x = if x < 30 then x else (add 2.00 x)

    finalPrice :: Float -> Float
    finalPrice i = cardFee (tax (coupon (sale i)))
```

Haskell

```
add :: Float -> Float -> Float
add x y = x + y

mul :: Float -> Float -> Float
mul x y = x * y

sale :: Float -> Float
sale = mul 0.8

coupon :: Float -> Float
coupon = add (-5.00)

tax :: Float -> Float
tax = mul 1.2

cardFee :: Float -> Float
cardFee x = if x < 30 then x else (add 2.00 x)
```

```
compose :: (a -> b)
        -> (b -> c)
        -> (a -> c)
compose f g =
  (\x -> f (g x))
```

```
finalPrice :: Float -> Float
finalPrice i = cardFee (tax (coupon (sale i)))
```

Haskell

```haskell
add :: Float -> Float -> Float
add x y = x + y

mul :: Float -> Float -> Float
mul x y = x * y

sale :: Float -> Float
sale = mul 0.8

coupon :: Float -> Float
coupon = add (-5.00)

tax :: Float -> Float
tax = mul 1.2

cardFee :: Float -> Float
cardFee x = if x < 30 then x else (add 2.00 x)
```

```haskell
(.)     :: (a -> b)
        -> (b -> c)
        -> (a -> c)
f . g =
  (\x -> f (g x))
```

```haskell
finalPrice :: Float -> Float
finalPrice i = cardFee (tax (coupon (sale i)))
```

Haskell

```haskell
add :: Float -> Float -> Float
add x y = x + y
mul :: Float -> Float -> Float
mul x y = x * y
sale :: Float -> Float
sale = mul 0.8
coupon :: Float -> Float
coupon = add (-5.00)
tax :: Float -> Float
tax = mul 1.2
cardFee :: Float -> Float
cardFee x = if x < 30 then x else (add 2.00 x)
```

```haskell
(.)      :: (a -> b)
         -> (b -> c)
         -> (a -> c)
f . g =
  (\x -> f (g x))
```

```haskell
finalPrice :: Float -> Float
finalPrice = cardFee
    `compose` tax
    `compose` coupon
    `compose` sale
```

Haskell

```haskell
add :: Float -> Float -> Float
add x y = x + y

mul :: Float -> Float -> Float
mul x y = x * y

sale :: Float -> Float
sale = mul 0.8

coupon :: Float -> Float
coupon = add (-5.00)

tax :: Float -> Float
tax = mul 1.2

cardFee :: Float -> Float
cardFee x = if x < 30 then x else (add 2.00 x)
```

```haskell
(.)       :: (a -> b)
          -> (b -> c)
          -> (a -> c)
f . g =
  (\x -> f (g x))
```

```haskell
finalPrice :: Float -> Float
finalPrice = cardFee . tax . coupon . sale
```

Haskell

```haskell
add :: Float -> Float -> Float
add x y = x + y

mul :: Float -> Float -> Float
mul x y = x * y

sale :: Float -> Float
sale = mul 0.8

coupon :: Float -> Float
coupon = add (-5.00)

tax :: Float -> Float
tax = mul 1.2

cardFee :: Float -> Float
cardFee x = if x < 30 then x else (add 2.00 x)
```

```haskell
(.)      :: (a -> b)
         -> (b -> c)
         -> (a -> c)
f . g =
  (\x -> f (g x))
```

```haskell
finalPrice :: Float -> Float
finalPrice = cardFee . tax . coupon . sale
```

"pointfree style"

Haskell

```
add :: Float -> Float -> Float
add x y = x + y

mul :: Float -> Float -> Float
mul x y = x * y

sale :: Float -> Float
sale = mul 0.8

coupon :: Float -> Float
coupon = add (-5.00)

tax :: Float -> Float
tax = mul 1.2

cardFee :: Float -> Float
cardFee x = if x < 30 then x else (add 2.00 x)
```

```
(.)     :: (a -> b)
        -> (b -> c)
        -> (a -> c)
f . g =
  (\x -> f (g x))
```

```
finalPrice :: Float -> Float
finalPrice = cardFee . tax . coupon . sale
```

"pointfree style"

└── **combinators?**

Haskell

```
add :: Float -> Float -> Float
add x y = x + y

mul :: Float -> Float -> Float
mul x y = x * y

sale :: Float -> Float
sale = mul 0.8

coupon :: Float -> Float
coupon = add (-5.00)

tax :: Float -> Float
tax = mul 1.2

cardFee :: Float -> Float
cardFee x = if x < 30 then x else (add 2.00 x)
```

```
(.)     :: (a -> b)
        -> (b -> c)
        -> (a -> c)
f . g =
  (\x -> f (g x))
```

```
finalPrice :: Float -> Float
finalPrice = cardFee . tax . coupon . sale
```

"pointfree style"

└── combinators?
abstraction?

Haskell

```
c1 x y z = x (y z)
```

```
finalPrice :: Float -> Float
finalPrice = cardFee . tax . coupon . sale
```

"pointfree style"

└── **combinators?**
abstraction?

Haskell

```
c1 x y z = x (y z)
```

```
S (S (K K) (S (K S) (S (K K) I))) (K (S (S (K S) (S (K K) I)) (K I)))
```

```
finalPrice :: Float -> Float
finalPrice = cardFee . tax . coupon . sale
```

"pointfree style"

└── combinators?
abstraction?

Haskell

```
c1 x y z = x (y z)
```

```
S (S (K K) (S (K S) (S (K K) I))) (K (S (S (K S) (S (K K) I)) (K I)))
```

"pointless style"

```
finalPrice :: Float -> Float
finalPrice = cardFee . tax . coupon . sale
```

"pointfree style"

combinators?
abstraction?

Haskell

```
c1 x y z = x (y z)
```

```
S (S (K K) (S (K S) (S (K K) I))) (K (S (S (K S) (S (K K) I)) (K I)))
```

"pointless style"

difficult to read
difficult to refactor

```
finalPrice :: Float -> Float
finalPrice = cardFee . tax . coupon . sale
```

"pointfree style"

combinators?
abstraction?

Haskell

```
"lazy semantics"
    aka "normal order"
    aka "call-by-name"
```

Haskell

"lazy semantics"
aka "normal order"
aka "call-by-name"

Goal:

```
void inc(int *x) {
  *x++;
}

void f(int *y) {
  inc(y);
  inc(y);
}
```

```haskell
data Ptr a = ...
```

Goal:

```c
void inc(int *x) {
  *x++;
}

void f(int *y) {
  inc(y);
  inc(y);
}
```

Haskell

```haskell
data Ptr a = ...

inc :: Ptr Int -> ()
inc x = ...
```

Goal:

```c
void inc(int *x) {
  *x++;
}

void f(int *y) {
  inc(y);
  inc(y);
}
```

Haskell

```haskell
data Ptr a = ...

inc :: Ptr Int -> ()
inc x = ...

f :: Ptr Int -> ()
f y = let o1 = inc y
          o2 = inc y
      in ()
```

Goal:

```c
void inc(int *x) {
  *x++;
}

void f(int *y) {
  inc(y);
  inc(y);
}
```

Haskell

```haskell
data Ptr a = ...

inc :: Ptr Int -> ()
inc x = ...

f :: Ptr Int -> ()
f y = let o1 = inc y
          o2 = inc y
      in ()
```

Goal:

```c
void inc(int *x) {
  *x++;
}

void f(int *y) {
  inc(y);
  inc(y);
}
```

`let v = e1 in e2` ⟶ `(\v -> e2) e1`

Haskell

```haskell
data Ptr a = ...

inc :: Ptr Int -> ()
inc x = ...

f :: Ptr Int -> ()
f y = let o1 = inc y
          o2 = inc y
      in ()
```

Goal:

```c
void inc(int *x) {
  *x++;
}

void f(int *y) {
  inc(y);
  inc(y);
}
```

```
let v = e1 in e2 ⟶ (\v -> e2) e1
```

```
f z ⟶ (\o1 -> (\o2 -> ()) (inc z)) (inc z)
```

Haskell

```haskell
data Ptr a = ...

inc :: Ptr Int -> ()
inc x = ...

f :: Ptr Int -> ()
f y = let o1 = inc y
          o2 = inc y
      in ()
```

Goal:

```c
void inc(int *x) {
  *x++;
}

void f(int *y) {
  inc(y);
  inc(y);
}
```

`let v = e1 in e2 ⟶ (\v -> e2) e1`

```
f z ⟶ (\o1 -> (\o2 -> ()) (inc z)) (inc z)
    ⟶ (\o2 -> ()) (inc z)                    [β-reduce]
```

Haskell

```
data Ptr a = ...

inc :: Ptr Int -> ()
inc x = ...

f :: Ptr Int -> ()
f y = let o1 = inc y
          o2 = inc y
      in ()
```

Goal:

```
void inc(int *x) {
  *x++;
}

void f(int *y) {
  inc(y);
  inc(y);
}
```

`let v = e1 in e2 ⟶ (\v -> e2) e1`

```
f z ⟶ (\o1 -> (\o2 -> ()) (inc z)) (inc z)
    ⟶ (\o2 -> ()) (inc z)                    [β-reduce]
    ⟶ ()                                     [β-reduce]
```

Haskell

```haskell
data Ptr a = ...

inc :: Ptr Int -> ()
inc x = ...

f :: Ptr Int -> ()
f y = let o1 = inc y
          o2 = inc y
      in ()
```

Goal:

```c
void inc(int *x) {
  *x++;
}

void f(int *y) {
  inc(y);
  inc(y);
}
```

`let v = e1 in e2` ⟶ `(\v -> e2) e1`

```
f z ⟶ (\o1 -> (\o2 -> ()) (inc z)) (inc z)
    ⟶ (\o2 -> ()) (inc z)                    [β-reduce]
    ⟶ ()                                     [β-reduce]
```

We never evaluated `inc z`!

Haskell

Ban all mutability!

```
inc :: Ptr Int -> ()
inc x = ...

f :: Ptr Int -> ()
f y = let o1 = inc y
          o2 = inc y
      in ()
```

Goal:

```
void inc(int *x) {
  *x++;
}

void f(int *y) {
  inc(y);
  inc(y);
}
```

```
let v = e1 in e2 ⟶ (\v -> e2) e1
```

```
f z ⟶ (\o1 -> (\o2 -> ()) (inc z)) (inc z)
    ⟶ (\o2 -> ()) (inc z)                    [β-reduce]
    ⟶ ()                                     [β-reduce]
```

We never evaluated `inc z`!

Haskell

Ban all mutability!

```
inc :: Ptr Int -> ()
inc x = ...

f :: Ptr Int -> ()
f y = let o1 = inc y
          o2 = inc y
      in ()
```

Goal:

```
void inc(int *x) {
  *x++;
}

void f(int *y) {
  inc(y);
  inc(y);
}
```

```
let v = e1 in e2 ⟶ (\v -> e2) e1
```

```
f z ⟶ (\o1 -> (\o2 -> ()) (inc z)) (inc z)
    ⟶ (\o2 -> ()) (inc z)                    [β-reduce]
    ⟶ ()                                     [β-reduce]
```

We never evaluated `inc z`!

Haskell

Ban all mutability!

```
inc :: Int -> Int
inc x = ...

f :: Ptr Int -> ()
f y = let o1 = inc y
          o2 = inc y
      in ()
```

Goal:

```
void inc(int *x) {
  *x++;
}

void f(int *y) {
  inc(y);
  inc(y);
}
```

```
let v = e1 in e2 ⟶ (\v -> e2) e1
```

```
f z ⟶ (\o1 -> (\o2 -> ()) (inc z)) (inc z)
    ⟶ (\o2 -> ()) (inc z)                [β-reduce]
    ⟶ ()                                  [β-reduce]
```

We never evaluated inc z!

Haskell

Ban all mutability!

```
inc :: Int -> Int
inc x = ...

f :: Ptr Int -> ()
f y = let o1 = inc y
          o2 = inc y
      in ()
```

Goal:

```
void inc(int *x) {
  *x++;
}

void f(int *y) {
  inc(y);
  inc(y);
}
```

```
let v = e1 in e2 ⟶ (\v -> e2) e1
```

```
f z ⟶ (\o1 -> (\o2 -> ()) (inc z)) (inc z)
    ⟶ (\o2 -> ()) (inc z)                    [β-reduce]
    ⟶ ()                                      [β-reduce]
```

We never evaluated `inc z`!

Haskell

Ban all mutability!

```
inc :: Int -> Int
inc x = ...

f :: Int -> Int
f y = let y1 = inc y
          y2 = inc y
      in y2
```

Goal:

```
void inc(int *x) {
  *x++;
}

void f(int *y) {
  inc(y);
  inc(y);
}
```

`let v = e1 in e2` ⟶ `(\v -> e2) e1`

```
f z ⟶ (\o1 -> (\o2 -> ()) (inc z)) (inc z)
    ⟶ (\o2 -> ()) (inc z)                    [β-reduce]
    ⟶ ()                                     [β-reduce]
```

We never evaluated `inc z`!

Haskell

Haskell

Goal:

```
print("a");
print("b");
```

Haskell

```
print :: String -> ()
print msg = ...
```

Goal:

```
print("a");
print("b");
```

Haskell

```
print :: String -> ()
print msg = ...

seq :: (a -> ())
    -> (b -> ())
    -> a -> b -> ()
seq f g x y = let o1 = f x
                  o2 = g y
              in ()
```

Goal:

```
print("a");
print("b");
```

Haskell

```
print :: String -> ()
print msg = ...

seq :: (a -> ())
    -> (b -> ())
    -> a -> b -> ()
seq f g x y = let o1 = f x
                  o2 = g y
              in ()

seq print print "a" "b"
```

Goal:

```
print("a");
print("b");
```

Haskell

```
print :: String -> ()
print msg = ...

seq :: (a -> ())
    -> (b -> ())
    -> a -> b -> ()
seq f g x y = let o1 = f x
                  o2 = g y
              in ()
```

Goal:

```
print("a");
print("b");
```

```
seq print print "a" "b"
  → (\o1 -> (\o2 -> ()) (print "b")) (print "a")
```

Haskell

```
print :: String -> ()
print msg = ...

seq :: (a -> ())
    -> (b -> ())
    -> a -> b -> ()
seq f g x y = let o1 = f x
                  o2 = g y
               in ()
```

Goal:

```
print("a");
print("b");
```

```
seq print print "a" "b"
```

→ `(\o1 -> (\o2 -> ()) (print "b")) (print "a")`

→ `(\o2 -> ()) (print "b")` [β-reduce]

Haskell

```
print :: String -> ()
print msg = ...

seq :: (a -> ())
    -> (b -> ())
    -> a -> b -> ()
seq f g x y = let o1 = f x
                  o2 = g y
               in ()
```

Goal:

```
print("a");
print("b");
```

```
seq print print "a" "b"
  → (\o1 -> (\o2 -> ()) (print "b")) (print "a")
  → (\o2 -> ()) (print "b")          [β-reduce]
  → ()                               [β-reduce]
```

Haskell

```
print :: String -> ()
print msg = ...

seq :: (a -> ())
    -> (b -> ())
    -> a -> b -> ()
seq f g x y = let o1 = f x
                  o2 = g y
              in ()
```

Goal:

```
print("a");
print("b");
```

```
seq print print "a" "b"

  → (\o1 -> (\o2 -> ()) (print "b")) (print "a")
  → (\o2 -> ()) (print "b")          [β-reduce]
  → ()                                [β-reduce]
```

We never evaluated `print`!

Banning mutability is not enough

"side effects"

- mutation
- printing
- user input
- file system

Haskell

"side effects"

- mutation
- printing
- user input
- file system

"returns"

Haskell

"side effects"

- mutation
- printing
- user input
- file system

"returns" → "evaluates to"
"reduces to"

Haskell

"side effects"

- mutation
- printing
- user input
- file system

"returns" → "evaluates to"
"reduces to"

```
f = (\x. (\y. x + y + 1))

f a b
```

Haskell

"side effects"

- mutation
- printing
- user input
- file system

"returns" → "evaluates to"
"reduces to"

```
f = (\x. (\y. x + y + 1))

f a b
 → (\x. (\y. x + y + 1)) a b
```

Haskell

"side effects"

- mutation
- printing
- user input
- file system

"returns" → "evaluates to"
"reduces to"

```
f = (\x. (\y. x + y + 1))

f a b
 → (\x. (\y. x + y + 1)) a b
 → (\y. a + y + 1) a b  [β-reduce]
```

Haskell

"side effects"

- mutation
- printing
- user input
- file system

"returns" → "evaluates to"
"reduces to"

```
f = (\x. (\y. x + y + 1))

f a b
  → (\x. (\y. x + y + 1)) a b
  → (\y. a + y + 1) a b         [β-reduce]
  → a + b + 1                   [β-reduce]
```

Haskell

"side effects"

- mutation
- printing
- user input
- file system

"returns" → "evaluates to"
"reduces to"

```
f = (\x. (\y. x + y + 1))

f a b
 → (\x. (\y. x + y + 1)) a b
 → (\y. a + y + 1) a b        [β-reduce]
 → a + b + 1                  [β-reduce]
```

```
print :: String  -> ()
inc   :: Ptr Int -> ()
```

Haskell

"side effects"

- mutation
- printing
- user input
- file system

"returns" → "evaluates to"
"reduces to"

```
f = (\x. (\y. x + y + 1))

f a b
 → (\x. (\y. x + y + 1)) a b
 → (\y. a + y + 1) a b        [β-reduce]
 → a + b + 1                  [β-reduce]
```

evaluates to → ()

```
print :: String  -> ()
inc   :: Ptr Int -> ()
```

Haskell

"side effects"

- mutation
- printing
- user input
- file system

"returns" → "evaluates to"
"reduces to"

```
f = (\x. (\y. x + y + 1))

f a b
 → (\x. (\y. x + y + 1)) a b
 → (\y. a + y + 1) a b         [β-reduce]
 → a + b + 1                   [β-reduce]
```

evaluates to ↓

```
print :: String  -> ()
inc   :: Ptr Int -> ()
```

side effects are not captured in the evaluation result

Haskell

side effects are not captured in the evaluation result

Haskell

allow side effects

side effects are not captured in the evaluation result

Haskell

forbid <u>all</u> side effects

side effects are not captured in the evaluation result

allow side effects

Haskell

forbid all side effects

allow side effects

side effects are not captured in the evaluation result

Haskell

forbid all side effects

side effects are not captured in the evaluation result

allow side effects ("impure" language)

Haskell

OCaml

```
let hello () = print "hello" ;;
hello () ;;
```

forbid all side effects

allow side effects
("impure" language)

side effects are not captured in the evaluation result

Haskell

forbid all side effects

OCaml

```
let hello () = print "hello" ;;
hello () ;;
```

Scala

```
def printHello(): Unit = {
  println("Hello")
}
hello()
```

allow side effects
("impure" language)

side effects are not captured in the evaluation result

Haskell

forbid all side effects

OCaml

```
let hello () = print "hello" ;;
hello () ;;
```

Scala

```
def printHello(): Unit = {
  println("Hello")
}
hello()
```

F#

```
let hello() = printfn "hello"
hello()
```

allow side effects
("impure" language)

side effects are not captured in the evaluation result

Haskell

- prevents normal order evaluation

OCaml

```
let hello () = print "hello" ;;
hello () ;;
```

Scala

```
def printHello(): Unit = {
  println("Hello")
}
hello()
```

F#

```
let hello() = printfn "hello"
hello()
```

forbid all side effects

allow side effects ("impure" language)

side effects are not captured in the evaluation result

Haskell

- prevents normal order evaluation
  - easier reasoning about performance

OCaml

```
let hello () = print "hello" ;;
hello () ;;
```

Scala

```
def printHello(): Unit = {
  println("Hello")
}
hello()
```

F#

```
let hello() = printfn "hello"
hello()
```

allow side effects ("impure" language)

forbid all side effects

side effects are not captured in the evaluation result

Haskell

- prevents normal order evaluation
  - easier reasoning about performance
- **effectful computations are easy to introduce**

OCaml

```
let hello () = print "hello" ;;
hello () ;;
```

Scala

```
def printHello(): Unit = {
  println("Hello")
}
hello()
```

F#

```
let hello() = printfn "hello"
hello()
```

allow side effects ("impure" language)

forbid all side effects

side effects are not captured in the evaluation result

Haskell

- prevents normal order evaluation
  - easier reasoning about performance
- effectful computations are easy to introduce

OCaml

```
let hello () = print "hello" ;;
hello () ;;
```

Scala

```
def printHello(): Unit = {
  println("Hello")
}
hello()
```

F#

```
let hello() = printfn "hello"
hello()
```

allow side effects ("impure" language)

forbid all side effects

side effects are not captured in the evaluation result

Haskell

- prevents normal order evaluation
  - easier reasoning about performance
- effectful computations are easy to introduce

OCaml

```
let hello () = print "hello" ;;
hello () ;;
```

Scala

```
def printHello(): Unit = {
  println("Hello")
}
hello()
```

F#

```
let hello() = printfn "hello"
hello()
```

allow side effects ("impure" language)

forbid <u>all</u> side effects

- mutation
- printing
- user input
- file system

side effects are not captured in the evaluation result

Haskell

- prevents normal order evaluation
  - easier reasoning about performance
- effectful computations are easy to introduce

OCaml

```
let hello () = print "hello" ;;
hello () ;;
```

Scala

```
def printHello(): Unit = {
  println("Hello")
}
hello()
```

F#

```
let hello() = printfn "hello"
hello()
```

allow side effects ("impure" language)

forbid <u>all</u> side effects

- mutation
- printing
- user input
- file system

("pure" language)

side effects are not captured in the evaluation result

Haskell

- prevents normal order evaluation
  - easier reasoning about performance
- effectful computations are easy to introduce

OCaml

```
let hello () = print "hello" ;;
hello () ;;
```

Scala

```
def printHello(): Unit = {
  println("Hello")
}
hello()
```

F#

```
let hello() = printfn "hello"
hello()
```

allow side effects ("impure" language)

??????

forbid <u>all</u> side effects

- mutation
- printing
- user input
- file system

("pure" language)

side effects are not captured in the evaluation result

Haskell

```
int *x;

int incBy(int y) {
  int r = *x;
  *x += y;
  return r;
}

int even(int a, int b) {
  return (*x % 2) ? a : b;
}

void f(int a) {
  int z = incBy(1);
  if (even(a, z) == 3) {
    incBy(z);
  }
  incBy(1);
}
```

Haskell

```c
int *x;

int incBy(int y) {
  int r = *x;
  *x += y;
  return r;
}

int even(int a, int b) {
  return (*x % 2) ? a : b;
}

void f(int a) {
  int z = incBy(1);
  if (even(a, z) == 3) {
    incBy(z);
  }
  incBy(1);
}
```

```haskell
incBy :: Int
      -> Int

even :: Int
     -> Int
     -> Int

f :: Int
  -> ()
```

Haskell

```c
int *x;

int incBy(int y) {
  int r = *x;
  *x += y;
  return r;
}

int even(int a, int b) {
  return (*x % 2) ? a : b;
}

void f(int a) {
  int z = incBy(1);
  if (even(a, z) == 3) {
    incBy(z);
  }
  incBy(1);
}
```

```haskell
incBy :: Int
      -> Int
      -> Int

even :: Int
     -> Int
     -> Int
     -> Int

f :: Int
  -> Int
  -> ()
```

Haskell

```c
int *x;

int incBy(int y) {
  int r = *x;
  *x += y;
  return r;
}

int even(int a, int b) {
  return (*x % 2) ? a : b;
}

void f(int a) {
  int z = incBy(1);
  if (even(a, z) == 3) {
    incBy(z);
  }
  incBy(1);
}
```

```haskell
incBy :: Int
      -> Int
      -> (Int, Int)

even :: Int
     -> Int
     -> Int
     -> (Int, Int)

f :: Int
  -> Int
  -> (Int, ())
```

Haskell

```
incBy :: Int
      -> Int
      -> (Int, Int)
```

```
even :: Int
     -> Int
     -> Int
     -> (Int, Int)
```

```
f :: Int
  -> Int
  -> (Int, ())
```

Haskell

```haskell
incBy :: Int
      -> Int
      -> (Int, Int)
```

```c
int incBy(int y) {
  int r = *x;
  *x += y;
  return r;
}
```

```haskell
incBy x y = (x + y, x)
```

```haskell
even :: Int
     -> Int
     -> Int
     -> (Int, Int)
```

```haskell
f :: Int
  -> Int
  -> (Int, ())
```

Haskell

```
incBy :: Int
      -> Int
      -> (Int, Int)
```

```
even :: Int
     -> Int
     -> Int
     -> (Int, Int)
```

```
int even(int a, int b) {
  return (*x % 2) ? a : b;
}
```

```
even a b x
  = (x, if x % 2
          then a
          else b )
```

```
f :: Int
  -> Int
  -> (Int, ())
```

Haskell

```
incBy :: Int
      -> Int
      -> (Int, Int)
```

```
even :: Int
     -> Int
     -> Int
     -> (Int, Int)
```

```
f :: Int
  -> Int
  -> (Int, ())
```

```
void f(int a) {
  int z = incBy(1);
  if (even(a, z) == 3) {
    incBy(z);
  }
  incBy(1);
}
```

```
f a x0 = let
    (x1, z) = incBy 1 x0
    (x2, b) = even a z x1
    (x3, _) =
      (if b == 3
         then (incBy x2 z)
         else (x2, x2))
    (x4, _) = incBy x3 1
  in (x4, ())
```

Haskell

```
incBy :: Int
      -> Int
      -> (Int, Int)
```

```
even :: Int
     -> Int
     -> Int
     -> (Int, Int)
```

```
f :: Int
  -> Int
  -> (Int, ())
```

```
incBy :: Int
      -> (Int -> (Int, Int))
```

```
even :: Int
     -> Int
     -> (Int -> (Int, Int))
```

```
f :: Int
  -> (Int -> (Int, ()))
```

Haskell

```haskell
incBy :: Int
      -> (Int -> (Int, Int))
```

```c
int incBy(int y) {
  ...
}
```

```haskell
even :: Int
     -> Int
     -> (Int -> (Int, Int))
```

```c
int even(int a, int b) {
  ...
}
```

```haskell
f :: Int
  -> (Int -> (Int, ()))
```

```c
void f(int a) {
  ...
}
```

Haskell

```haskell
incBy :: Int
      -> (Int -> (Int, Int))
```

```c
int incBy(int y) {
  ...
}
```

```haskell
even :: Int
     -> Int
     -> (Int -> (Int, Int))
```

```c
int even(int a, int b) {
  ...
}
```

```haskell
f :: Int
  -> (Int -> (Int, ()))
```

```c
void f(int a) {
  ...
}
```

Haskell

```haskell
incBy :: Int
      -> (Int -> (Int, Int))
```

```c
int incBy(int y) {
  ...
}
```

```haskell
even :: Int
     -> Int
     -> (Int -> (Int, Int))
```

```c
int even(int a, int b) {
  ...
}
```

```haskell
f :: Int
  -> (Int -> (Int, ()))
```

```c
void f(int a) {
  ...
}
```

Haskell

```
incBy :: Int
      -> (Int -> (Int, Int))
```

```
int incBy(int y) {
  ...
}
```

```
even :: Int
     -> Int
     -> (Int -> (Int, Int))
```

```
int even(int a, int b) {
  ...
}
```

```
f :: Int
  -> (Int -> (Int, ()))
```

```
void f(int a) {
  ...
}
```

Haskell

```
incBy :: Int
      -> (Int -> (Int, Int))
```

```
int incBy(int y) {
  ...
}
```

```
even :: Int
     -> Int
     -> (Int -> (Int, Int))
```

```
int even(int a, int b) {
  ...
}
```

```
f :: Int
  -> (Int -> (Int, ()))
```

```
void f(int a) {
  ...
}
```

Haskell

```haskell
incBy :: Int
      -> (Int -> (Int, Int))
```

```c
int incBy(int y) {
  ...
}
```

```haskell
even :: Int
     -> Int
     -> (Int -> (Int, Int))
```

```c
int even(int a, int b) {
  ...
}
```

```haskell
f :: Int
  -> (Int -> (Int, ()))
```

```c
void f(int a) {
  ...
}
```

Haskell

```haskell
incBy :: Int
      -> (Int -> (Int, Int))
```

```c
int incBy(int y) {
  ...
}
```

```haskell
even :: Int
     -> Int
     -> (Int -> (Int, Int))
```

```c
int even(int a, int b) {
  ...
}
```

```haskell
f :: Int
  -> (Int -> (Int, ()))
```

```c
void f(int a) {
  ...
}
```

Haskell

```
incBy :: Int
      -> (Int -> (Int, Int))
```

```
int incBy(int y) {
  ...
}
```

```
even :: Int
     -> Int
     -> (Int -> (Int, Int))
```

```
int even(int a, int b) {
  ...
}
```

```
f :: Int
  -> (Int -> (Int, ()))
```

```
void f(int a) {
  ...
}
```

```
data XPtr r = XPtr (Int -> (Int, r))
```

Haskell

```
incBy :: Int
      -> XPtr Int
```

```
int incBy(int y) {
  ...
}
```

```
even :: Int
     -> Int
     ->  XPtr Int
```

```
int even(int a, int b) {
  ...
}
```

```
f :: Int
  ->  XPtr ()
```

```
void f(int a) {
  ...
}
```

```
data XPtr r = XPtr (Int -> (Int, r))
```

Haskell

```haskell
incBy :: Int
      -> XPtr Int
```

```c
int incBy(int y) {
  ...
}
```

```haskell
even :: Int
     -> Int
     -> XPtr Int
```

```c
int even(int a, int b) {
  ...
}
```

```haskell
f :: Int
  -> XPtr ()
```

```c
void f(int a) {
  ...
}
```

```haskell
data XPtr r = XPtr (Int -> (Int, r))
```

Now the side effect is in the evaluation result!

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))
```

```
incBy :: Int
      -> XPtr Int
```

```
int z = incBy(1);
```

```
\(x, a) ->
  let (x', z) = incBy 1
   in ...
```

```
even :: Int
     -> Int
     -> XPtr Int
```

```
f :: Int
  -> XPtr ()
```

```
bind (XPtr p) f
  = XPtr (\x0 ->
       let (y, x1) = p x0
        in f y)
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))
```

```
incBy :: Int
      ->  XPtr Int
```

```
int z = incBy(1);
```

```
\(x, a) ->
  let (x', z) = incBy 1
   in ...
```

```
even :: Int
     -> Int
     ->  XPtr Int
```

```
bind :: XPtr a
     -> (a -> XPtr b)
     -> XPtr b
bind (XPtr p) f
  = XPtr (\x0 ->
      let (y, x1) = p x0
       in f y)
```

```
f :: Int
  ->  XPtr ()
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))
```

```
incBy :: Int
      ->  XPtr Int
```

```
int z = incBy(1);
```

```
\(x, a) ->
  let (x', z) = incBy 1
   in ...
```

```
even :: Int
     -> Int
     ->  XPtr Int
```

```
bind :: XPtr a
     -> (a -> XPtr b)
     -> XPtr b
```

```
f :: Int
  ->  XPtr ()
```

```
(incBy 1) `bind` (\z -> ...)
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))
```

```
incBy :: Int
      -> XPtr Int

even :: Int
     -> Int
     -> XPtr Int

f :: Int
  -> XPtr ()
```

```
int z = 1;

\(x, a) ->
  let (x', z) = incBy 1
   in ...

bind :: XPtr a
     -> (a -> XPtr b)
     -> XPtr b

(incBy 1) `bind` (\z -> ...)
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))
```

```
incBy :: Int
      ->  XPtr Int
```

```
int z = 1;
```

```
\(x, a) ->
  let (x', z) = incBy 1
   in ...
```

```
even :: Int
     -> Int
     ->  XPtr Int
```

```
bind :: XPtr a
     -> (a -> XPtr b)
     -> XPtr b
```

```
f :: Int
  ->  XPtr ()
```

```
1  `bind` (\z -> ...)
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))
```

```
incBy :: Int
      ->  XPtr Int

even :: Int
     -> Int
     ->  XPtr Int

f :: Int
  ->  XPtr ()
```

```
int z = 1;

\(x, a) ->
  let (x', z) = incBy 1
   in ...

bind :: XPtr a
     -> (a -> XPtr b)
     -> XPtr b

     1 `bind` (\z -> ...)
```

1 does not have type XPtr!

Haskell

```haskell
data XPtr r = XPtr (Int -> (Int, r))
```

```haskell
incBy :: Int
      -> XPtr Int
```

```c
int z = 1;
```

```haskell
\(x, a) ->
  let (x', z) = incBy 1
   in ...
```

```haskell
even :: Int
     -> Int
     -> XPtr Int
```

```haskell
pure :: a -> XPtr a
bind :: XPtr a
     -> (a -> XPtr b)
     -> XPtr b
```

```haskell
f :: Int
  -> XPtr ()
```

```haskell
1 `bind` (\z -> ...)
```

1 does not have type XPtr!

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))
```

```
incBy :: Int
      ->  XPtr Int
```

```
even :: Int
     -> Int
     ->  XPtr Int
```

```
f :: Int
  ->  XPtr ()
```

```
int z = 1;
```

```
\(x, a) ->
  let (x', z) = incBy 1
   in ...
```

```
pure  :: a -> XPtr a
bind :: XPtr a
     -> (a -> XPtr b)
     -> XPtr b
```

```
(pure 1) `bind` (\z -> ...)
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))
```

```
incBy :: Int
      ->  XPtr Int
```

```
int z = 1;
```

```
\(x, a) ->
  let (x', z) = incBy 1
   in ...
```

```
even :: Int
     -> Int
     ->  XPtr Int
```

```
  pure  :: a -> XPtr a
bind :: XPtr a
     -> (a -> XPtr b)
     -> XPtr b
```

```
f :: Int
  ->  XPtr ()
```

```
(pure 1) `bind` (\z -> ...)
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))
```

```
incBy :: Int
      ->  XPtr Int
```

```
int z = 1;
```

```
\(x, a) ->
  let (x', z) = incBy 1
   in ...
```

```
even :: Int
     -> Int
     ->  XPtr Int
```

```
return :: a -> XPtr a
```

```
bind :: XPtr a
     -> (a -> XPtr b)
     -> XPtr b
```

```
f :: Int
  ->  XPtr ()
```

```
(pure 1) `bind` (\z -> ...)
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))
```

```
incBy :: Int
      ->  XPtr Int
```

```
int z = 1;
```

```
\(x, a) ->
  let (x', z) = incBy 1
   in ...
```

Monad (lecture 9)

return: a → M a

A function for creating an elemen

bind: M a → (a → M b) → M b

Sequencing: Take an element of a

```
return :: a -> XPtr a
bind :: XPtr a
     -> (a -> XPtr b)
     -> XPtr b
```

```
(pure 1) `bind` (\z -> ...)
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))
```

```
data State s a = State (s -> (s, a))
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))

data State s a = State (s -> (s, a))

data Maybe a = Just a | Nothing
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))

data State s a = State (s -> (s, a))

data Maybe a = Just a | Nothing

data Cont r a = Cont ((a -> r) -> r)
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))

data State s a = State (s -> (s, a))

data Maybe a = Just a | Nothing

data Cont r a = Cont ((a -> r) -> r)

data Reader s a = Reader (s -> a)
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))

data State s a = State (s -> (s, a))

data Maybe a = Just a | Nothing

data Cont r a = Cont ((a -> r) -> r)

data Reader s a = Reader (s -> a)

data Writer s a = Writer (s, a)
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))

data State s a = State (s -> (s, a))

data Maybe a = Just a | Nothing

data Cont r a = Cont ((a -> r) -> r)

data Reader s a = Reader (s -> a)

data Writer s a = Writer (s, a)

data Either a b = Left a | Right b
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))

data State s a = State (s -> (s, a))

data Maybe a = Just a | Nothing

data Cont r a = Cont ((a -> r) -> r)

data Reader s a = Reader (s -> a)

data Writer s a = Writer (s, a)

data Either a b = Left a | Right b

data List a = Cons a (List a) | Nil
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))

data State s a = State (s -> (s, a))

data Maybe a = Just a | Nothing

data Cont r a = Cont ((a -> r) -> r)

data Reader s a = Reader (s -> a)

data Writer s a = Writer (s, a)

data Either a b = Left a | Right b

data List a = Cons a (List a) | Nil

data Parser a = Parser (String -> Maybe (String, a))
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))
```

```
data State s a = State (s -> (s, a))

    bindForState :: State s a -> (a -> State s b) -> State s b
    returnForState :: a -> State s a

data Cont r a = Cont ((a -> r) -> r)

    bindForCont :: Cont r a -> (a -> Cont r b) -> Cont r b
    returnForCont :: a -> Cont s a
```

Haskell

```haskell
data XPtr r = XPtr (Int -> (Int, r))

data State s a = State (s -> (s, a))

    bindForState :: State s a -> (a -> State s b) -> State s b
    returnForState :: a -> State s a

data Cont r a = Cont ((a -> r) -> r)

    bindForCont :: Cont r a -> (a -> Cont r b) -> Cont r b
    returnForCont :: a -> Cont s a

compose   :: (a ->   b) -> (b ->   c) -> (a ->   c)
compose f g = (\x -> f (g x))
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))

data State s a = State (s -> (s, a))

    bindForState :: State s a -> (a -> State s b) -> State s b
    returnForState :: a -> State s a

data Cont r a = Cont ((a -> r) -> r)

    bindForCont :: Cont r a -> (a -> Cont r b) -> Cont r b
    returnForCont :: a -> Cont s a

      compose  :: (a ->   b) -> (b ->   c) -> (a ->   c)
      compose f g = (\x -> f (g x))

      mcompose :: (a -> m b) -> (b -> m c) -> (a -> m c)
      mcompose f g = (\x -> bind (g x) f)
```

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))

data State s a = State (s -> (s, a))

    bindForState :: State s a -> (a -> State s b) -> State s b
    returnForState :: a -> State s a

data Cont r a = Cont ((a -> r) -> r)

    bindForCont :: Cont r a -> (a -> Cont r b) -> Cont r b
    returnForCont :: a -> Cont s a

      compose  :: (a ->   b) -> (b ->   c) -> (a ->   c)
      compose f g = (\x -> f (g x))

      mcompose :: (a -> m b) -> (b -> m c) -> (a -> m c)
      mcompose f g = (\x -> bind (g x) f)
```

bind → bindForState? bindForCont? bindForMaybe? ...

Haskell

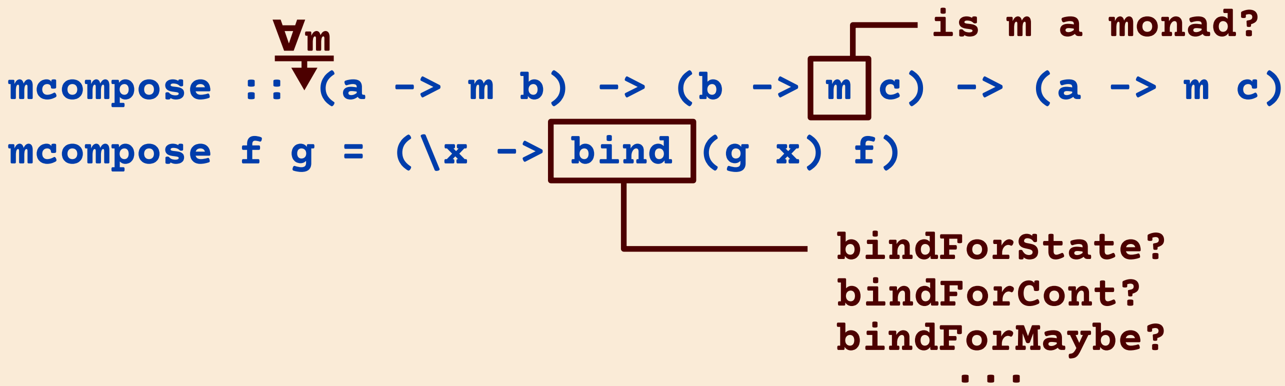
```
data XPtr r = XPtr (Int -> (Int, r))

data State s a = State (s -> (s, a))

    bindForState :: State s a -> (a -> State s b) -> State s b
    returnForState :: a -> State s a

data Cont r a = Cont ((a -> r) -> r)

    bindForCont :: Cont r a -> (a -> Cont r b) -> Cont r b
    returnForCont :: a -> Cont s a

        compose  :: (a ->   b) -> (b ->   c) -> (a ->   c)
        compose f g = (\x -> f (g x))

        mcompose :: (a -> m b) -> (b -> m c) -> (a -> m c)
        mcompose f g = (\x -> bind (g x) f)
```

is m a monad?

Haskell

```
data XPtr r = XPtr (Int -> (Int, r))


data State s a = State (s -> (s, a))

    bindForState :: State s a -> (a -> State s b) -> State s b
    returnForState :: a -> State s a

data Cont r a = Cont ((a -> r) -> r)

    bindForCont :: Cont r a -> (a -> Cont r b) -> Cont r b
    returnForCont :: a -> Cont s a


      compose  :: (a ->   b) -> (b ->   c) -> (a ->   c)
      compose f g = (\x -> f (g x))
                   ∀m                            is m a monad?
      mcompose :: (a -> m b) -> (b -> m c) -> (a -> m c)
      mcompose f g = (\x -> bind (g x) f)
```

Haskell


∀m
is m a monad?
mcompose :: (a -> m b) -> (b -> m c) -> (a -> m c)
mcompose f g = (\x -> bind (g x) f)
bindForState?
bindForCont?
bindForMaybe?
...

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)
```

∀m

```
mcompose :: (a -> m b) -> (b -> m c) -> (a -> m c)
mcompose f g = (\x -> bind (g x) f)
```

is m a monad?

bindForState?
bindForCont?
bindForMaybe?
...

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState
```

∀m

is m a monad?

```
mcompose :: (a -> m b) -> (b -> m c) -> (a -> m c)
mcompose f g = (\x -> bind (g x) f)
```

bindForState?
bindForCont?
bindForMaybe?
...

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState
```

```
mcompose :: MonadInstance m
         -> (a -> m b)
         -> (b -> m c)
         -> (a -> m c)
mcompose (MonadInstance return bind) f g =
  (\x -> bind (g x) f)
```

∀m

is m a monad?

```
mcompose :: (a -> m b) -> (b -> m c) -> (a -> m c)
mcompose f g = (\x -> bind (g x) f)
```

bindForState?
bindForCont?
bindForMaybe?
...

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState
```

```
mcompose :: MonadInstance m
         -> (a -> m b)
         -> (b -> m c)
         -> (a -> m c)
mcompose (MonadInstance return bind) f g =
  (\x -> bind (g x) f)
```

```
f :: a -> State s b
g :: b -> State s c
```

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState
```

```
mcompose :: MonadInstance m
         -> (a -> m b)
         -> (b -> m c)
         -> (a -> m c)
mcompose (MonadInstance return bind) f g =
  (\x -> bind (g x) f)
```

```
f :: a -> State s b
g :: b -> State s c
```

```
:t (mcompose stateMonad f g)
```

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState
```

```
mcompose :: MonadInstance m
         -> (a -> m b)
         -> (b -> m c)
         -> (a -> m c)
mcompose (MonadInstance return bind) f g =
  (\x -> bind (g x) f)
```

```
f :: a -> State s b
g :: b -> State s c
```

```
:t (mcompose stateMonad f g)
> ... :: a -> State s c
```

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState

     mcompose :: MonadInstance m
              -> (a -> m b)
              -> (b -> m c)
              -> (a -> m c)
     mcompose (MonadInstance return bind) f g =
       (\x -> bind (g x) f)
```

```
f :: a -> State s b
g :: b -> State s c

:t (mcompose stateMonad f g)
> ... :: a -> State s c
```

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState

     mcompose :: MonadInstance m
              -> (a -> m b)
              -> (b -> m c)
              -> (a -> m c)
     mcompose (MonadInstance return bind) f g =
       (\x -> bind (g x) f)

       f :: a -> State s b
       g :: b -> State s c
                                             contMonad?   ✗
       :t (mcompose stateMonad f g)
       > ... :: a -> State s c
```

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState

     mcompose :: MonadInstance m
              -> (a -> m b)
              -> (b -> m c)
              -> (a -> m c)
     mcompose (MonadInstance return bind) f g =
       (\x -> bind (g x) f)
```

```
f :: a -> State s b
g :: b -> State s c

:t (mcompose stateMonad f g)
> ... :: a -> State s c
```

contMonad? ✗

maybeMonad? ✗

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState

     mcompose :: MonadInstance m
              -> (a -> m b)
              -> (b -> m c)
              -> (a -> m c)
     mcompose (MonadInstance return bind) f g =
       (\x -> bind (g x) f)
```

```
f :: a -> State s b
g :: b -> State s c

:t (mcompose stateMonad f g)
> ... :: a -> State s c
```

contMonad? ✗
maybeMonad? ✗
listMonad? ✗

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState
```

```
mcompose :: MonadInstance m
         -> (a -> m b)
         -> (b -> m c)
         -> (a -> m c)
mcompose (MonadInstance return bind) f g =
  (\x -> bind (g x) f)
```

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState

instance Monad (State s) where
  return = returnForState
  bind = bindForState

     mcompose :: MonadInstance m
              -> (a -> m b)
              -> (b -> m c)
              -> (a -> m c)
     mcompose (MonadInstance return bind) f g =
       (\x -> bind (g x) f)
```

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState

instance Monad (State s) where
  return = returnForState
  bind = bindForState
```

```
mcompose :: Monad m =>
         -> (a -> m b)
         -> (b -> m c)
         -> (a -> m c)
mcompose f g =
  (\x -> bind (g x) f)
```

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState

instance Monad (State s) where
  return = returnForState
  bind = bindForState


     mcompose :: Monad m =>
              -> (a -> m b)
              -> (b -> m c)
              -> (a -> m c)
     mcompose f g =
       (\x -> bind (g x) f)
```

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState

instance Monad (State s) where
  return = returnForState
  bind = bindForState


     mcompose :: Monad m =>
                 -> (a -> m b)
                 -> (b -> m c)
                 -> (a -> m c)
     mcompose f g =
       (\x -> bind (g x) f)
```

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState

instance Monad (State s) where
  return = returnForState
  bind = bindForState
```

```
mcompose :: Monad m =>
         -> (a -> m b)
         -> (b -> m c)
         -> (a -> m c)
mcompose f g =
  (\x -> bind (g x) f)
```

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState

instance Monad (State s) where
  return = returnForState
  bind = bindForState

    mcompose :: Monad m =>
             -> (a -> m b)
             -> (b -> m c)
             -> (a -> m c)
    mcompose f g =
      (\x -> bind (g x) f)
```

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState

instance Monad (State s) where
  return = returnForState
  bind = bindForState

     mcompose :: Monad m =>
              -> (a -> m b)
              -> (b -> m c)
              -> (a -> m c)
     mcompose f g =
       (\x -> bind (g x) f)
```

Haskell

```
data MonadInstance m = MonadInstance
  (forall a. a -> m a)
  (forall a b. m a -> (a -> m b) -> m b)

class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForState bindForState

instance Monad (State s) where
  return = returnForState
  bind = bindForState

    mcompose :: Monad m =>
             -> (a -> m b)
             -> (b -> m c)
             -> (a -> m c)
    mcompose f g =
      (\x -> bind (g x) f)
```

✓

- \+ cleaner, more concise code
- \+ retrieving instances is easy
- − requires language support
- − multiple instances for type

Haskell

```
class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b
```

Haskell

```
class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

class Applicative m where
  pure :: a -> m a
  app :: m (a -> b) -> m a -> m b

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

Haskell

```
class MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

class Applicative m where
  pure :: a -> m a
  app :: m (a -> b) -> m a -> m b

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

Haskell

```haskell
class MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

class Applicative m where
  pure :: a -> m a
  app :: m (a -> b) -> m a -> m b

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

```haskell
class Alternative m where
  empty :: m a
  (<|>) :: m a -> m a -> m a
```

```haskell
class Traversable t where
  traverse :: Applicative f
           => (a -> f b)
           -> t a
           -> f (t b)
```

```haskell
class Foldable f where
  foldmap :: Monoid m
          => (a -> m)
          -> f a
          -> m
```

```haskell
class Semigroup a where
  (<>) :: a -> a -> a
```

```haskell
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
```

Haskell

```
class MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

```
class Alternative m where
  empty :: m a
  (<|>) :: m a -> m a -> m a
```

```
class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b
```

```
class Applicative m where
  pure :: a -> m a
  app :: m (a -> b) -> m a -> m b
```

```
class Traversable t where
  traverse :: Applicative f
           => (a -> f b)
           -> t a
           -> f (t b)
```

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

```
class Foldable f where
  foldmap :: Monoid m
          => (a -> m)
          -> f a
          -> m
```

```
class Semigroup a where
  (<>) :: a -> a -> a
```

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
```

Haskell

```
class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

class Applicative m where
  pure :: a -> m a
  app :: m (a -> b) -> m a -> m b


  flip :: (a -> b -> c) -> (b -> a -> c)
  flip f x y = f y x
```

Haskell

```
class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

class Applicative m where
  pure :: a -> m a
  app :: m (a -> b) -> m a -> m b

  flip :: (a -> b -> c) -> (b -> a -> c)
  flip f x y = f y x

      bind :: m a -> (a -> m b) -> m b

 (flip app) :: m a -> m (a -> b) -> m b
```

Haskell

```
class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

class Applicative m where
  pure :: a -> m a
  app :: m (a -> b) -> m a -> m b


  flip :: (a -> b -> c) -> (b -> a -> c)
  flip f x y = f y x


       bind :: m a -> (a -> m b) -> m b

(flip app) :: m a -> m (a -> b) -> m b
```

Haskell

```
class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

class Applicative m where
  pure :: a -> m a
  app :: m (a -> b) -> m a -> m b

  flip :: (a -> b -> c) -> (b -> a -> c)
  flip f x y = f y x
```

pure computations and effects can interact

```
      bind :: m a -> (a -> m b) -> m b

 (flip app) :: m a -> m (a -> b) -> m b
```

Haskell

```
class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

class Applicative m where
  pure :: a -> m a
  app :: m (a -> b) -> m a -> m b

  flip :: (a -> b -> c) -> (b -> a -> c)
  flip f x y = f y x
```

pure computations and effects can interact

```
      bind :: m a -> (a -> m b) -> m b

(flip app) :: m a -> m (a -> b) -> m b
```

effects and pure computations are separate

Haskell

```
class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

class Applicative m where
  pure :: a -> m a
  app :: m (a -> b) -> m a -> m b
```

```
ifM :: Monad m => m Bool -> m a -> m a -> m a
```

```
ifA :: Applicative m => m Bool -> m a -> m a -> m a
```

Haskell

```
class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

class Applicative m where
  pure :: a -> m a
  app :: m (a -> b) -> m a -> m b

    ite :: Bool -> a -> a -> a
    ite c t f = if c then t else f

    ifM :: Monad m => m Bool -> m a -> m a -> m a

    ifA :: Applicative m => m Bool -> m a -> m a -> m a
```

Haskell

```
class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

class Applicative m where
  pure :: a -> m a
  app :: m (a -> b) -> m a -> m b

    ite :: Bool -> a -> a -> a
    ite c t f = if c then t else f

    ifM :: Monad m => m Bool -> m a -> m a -> m a
    ifM c t f = c `bind` (\c' -> ite c' t f)

    ifA :: Applicative m => m Bool -> m a -> m a -> m a
    ifA c t f = (pure ite) `app` c `app` t `app` f
```

Haskell

```haskell
class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

class Applicative m where
  pure :: a -> m a
  app :: m (a -> b) -> m a -> m b

    ite :: Bool -> a -> a -> a
    ite c t f = if c then t else f

    ifM :: Monad m => m Bool -> m a -> m a -> m a
    ifM c t f = c `bind` (\c' -> ite c' t f)

      ifM (Just True) (Just 1) Nothing
      > Just 1

    ifA :: Applicative m => m Bool -> m a -> m a -> m a
    ifA c t f = (pure ite) `app` c `app` t `app` f
```

Haskell

```
class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

class Applicative m where
  pure :: a -> m a
  app :: m (a -> b) -> m a -> m b

    ite :: Bool -> a -> a -> a
    ite c t f = if c then t else f

    ifM :: Monad m => m Bool -> m a -> m a -> m a
    ifM c t f = c `bind` (\c' -> ite c' t f)

      ifM (Just True) (Just 1) Nothing
      > Just 1

    ifA :: Applicative m => m Bool -> m a -> m a -> m a
    ifA c t f = (pure ite) `app` c `app` t `app` f

      ifA (Just True) (Just 1) Nothing
      > Nothing
```