# Teoria das Categorias para Programadores

Fabrício Olivetti de França

17 de Agosto de 2019

UFABC

CMCC
Centro de Matemática, Computação e Cognição
Center for Mathematics, Computation and Cognition

# Topics

# Functors Adjuntos

# Isomorfismos de Categorias

Como podemos definir o isomorfismo entre duas categorias $C, D$?

# Isomorfismos de Categorias

Os morfismos entre duas categorias são Functors, duas categorias são isomorfas se temos dois Functors, $R : C \rightarrow D$ e $L : D \rightarrow C$, tal que a composição deles forma o Functor identidade.

# Isomorfismos de Categorias

$$R \circ L = I_D$$

e

$$L \circ R = I_C$$

# Isomorfismos de Categorias

Lembrando que o Functor identidade é dado por:

```haskell
data Identity a = Identity a

instance Functor Identity where
    fmap f (Identity x) = Identity (f x)
```

# Isomorfismos de Categorias

Para provar que duas categorias são isomórficas, precisamos definir transformações naturais entre a composição dos Functors e o Functor Identidade:

```
eta  :: Id    -> (R.L)
eta' :: (R.L) -> Id

eps  :: (L.R) -> Ic
eps' :: Ic    -> (L.R)
```

# Functors Adjuntos

Dizemos que dois Functors `R, L` são adjuntos se eles possuem as transformações naturais `eta, eps`, mas sem a necessidade de existir `eta', eps'`.

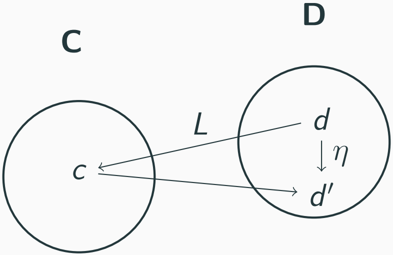
# Functors Adjuntos

- O Functor `L` é denominado adjunto esquerdo (*left adjunct*)
- O Functor `R` é o adjunto direito (*right adjunct*)
- `eta` é chamado de `unit`
- `eps` de `counit`.

Denotamos $L \dashv R$ como $L$ é o adjunto esquerdo de $R$.

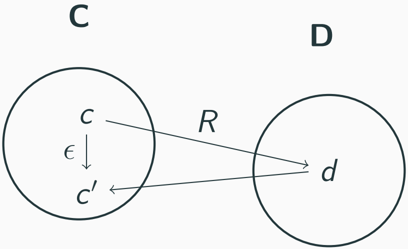
# Functors Adjuntos

A função `eta` pega um objeto de `D` e faz um *passeio* entre as categorias `C, D` utilizando os Functors `R . L`, retornando em outro objeto de `D`.



# Functors Adjuntos

A função eps indica como chegar em `c'` partindo de `c` seguindo o caminho `L . R`.



# Functors Adjuntos

Em outras palavras, o `unit` (também chamado de `return` e `pure` em outros contextos) permite introduzir um container ou Functor `R.L` em todo tipo `d`.

# Functors Adjuntos

Por outro lado, o `counit` (em algumas linguagens conhecido como `extract`) permite retirar um objeto de um container ou Functor:

```haskell
-- F = R . L
-- G = L . R

unit :: d -> F d

counit :: G c -> c
```

# Functors Adjuntos

A classe de Functors adjuntos é definido como:

```
class (Functor f, Representable y) =>
  Adjunction f u | f -> u, u -> f where
    unit   :: a -> u (f a)
    counit :: f (u a) -> a
```

# Functors Adjuntos

A condição `f -> u, u -> f` é uma **dependência funcional** e implica que só pode existir uma única instância para `f` na esquerda e para `u` a direita.

# Functors Adjuntos

Ou seja, se eu defino `Adjunction [] (Reader a)`, não posso definir `Adjunction Maybe (Reader a)` nem `Adjunction [] Maybe`.

# Functors Adjuntos

Junto dessas duas funções também podemos definir essa classe através de `leftAdjunction` e `rightAdjunction`:

```
class (Functor f, Representable y) =>
  Adjunction f u | f -> u, u -> f where
    leftAdjunct  :: (f a -> b) -> a -> u b
    rightAdjunct :: (a -> u b) -> f a -> b
```

# Functors Adjuntos

E elas se relacionam da seguinte forma:

```
unit          = leftAdjunct id
counit        = rightAdjunct id
leftAdjunct   = fmap g . unit
rightAdjunct = counit . fmap g
```

# Functors Adjuntos

Existem poucos Functors pertencentes a **Hask** que são adjuntos, porém a combinação $L.R$ e $R.L$ formam os conceitos de Monads e Comonads, conforme veremos mais adiante.

# Curry e Uncurry

Um exemplo de Functors adjuntos no Haskell são `(,a)` e `Reader a = (a ->)`. Podemos definir a instância como:

```haskell
instance Adjunction (,a) (Reader a) where
  -- unit :: c -> Reader a (c,a)
  unit x = \a -> (x, a)

  -- counit :: (Reader a c, a) -> c
  counit (f, x) = f x
```

# Curry e Uncurry

Podemos definir `leftAdjunct` e `rightAdjunct` automaticamente como:

```haskell
-- leftAdjunct :: ((x,a) -> y) -> x -> a -> y
leftAdjunct g = \x -> (fmap g . unit) x
= \x -> (fmap g) (unit x)
= \x -> (fmap g) (\a -> (x, a))
= \x -> g . (\a -> (x, a))
= \x -> \a -> g (x,a)
```

# Curry e Uncurry

```haskell
-- rightAdjunct :: (x -> (a -> y)) -> (x,a) -> y
rightAdjunct g (x,a) = counit . (fmap g) (x,a)
= counit (g x, a)
= g x a
```

Podemos perceber que `leftAdjunct = curry` e `rightAdjunct = uncurry`.

# Curry e Uncurry

Lembrando que `(,a) = Writer a` e, partindo da definição de Functors Adjuntos, podemos criar dois novos Functors com a composição de `Reader` com `Writer`:

## Curry e Uncurry

```haskell
-- State a b = Reader a (Writer a b)
type State a b = a -> (b,a)

-- Store a b = Writer (Reader a b) a
type Store a b = (a -> b, a)
```

Esses Functors permitem a criação de estados e armazenamento em uma linguagem puramente funcional.

# Monads

# Monads

> there are entire subcultures of young men these days who just hang out online waiting for someone to ask a question about monads
>
> — Monoid Mary (@argumatronic) 4 de março de 2019

# Monads

O uso de Monads gerou diversos mitos entre os programadores por conta de seu uso em programação (não necessariamente em Haskell).

# Monads

Isso motivou a criação de diversos tutoriais traçando uma analogia de Monads com outros conceitos fora da computação ou com enfoque em uma de suas aplicações práticas.

# Monads

De acordo com o Haskell wiki e sumarizado no texto What I wish I knew when learning Haskell, um Monad **não**:

- Define funções impuras
- Cria e gerencia estados de sistema
- Permite sequenciamento imperativo
- Define IO
- É dependente de avaliação preguiçosa
- É uma porta dos fundos para efeito colateral
- É uma linguagem imperativa dentro do Haskell
- Necessita de conhecimento de matemática abstrata para entender
- Exclusivos do Haskell

# Monads

A dificuldade em entender Monads se dá por conta do pensamento imperativo que costuma ser nosso primeiro c ontato com programação.

# Monads

Considere a função para calcular a magnitude de um vetor $3D$:

```
double vmag (double * v) {
   double d = 0.0;
   int i;
   for (i=0; i<3; i++)
       d += v[i]*v[i];
   return sqrt(d);
}
```

# Monads

Quando pensamos em reestruturar nosso código, verificamos trechos que podem ser utilizados por outras funções e modularizamos:

```
double square(double x) {
    return x*x;
}

double sum_with(double * v, std::function<double(double)> f) {
  double sum = 0.0;
  int i;
  for (i=0; i<3; i++) {
    sum += f(v[i]);
  }
  return sum;
}

double vmag (double * v) {
    return sqrt(sum_with(v, square));
}
```

# Monads

Que, de forma equivalente em Haskell temos:

```
vmag = sqrt . sum . fmap (^2)
```

# Monads

Vimos anteriormente o caso do nosso `Writer w a` que alterava a saída de nossas funções com um *embelezamento*, de forma a evitar transformar uma função pura em impura.

# Monads

Essa estrutura nos obrigou a criar um operador de composição específico para esses casos, gerando a Categoria Kleisli, que detalharemos em seguida.

# Categoria Kleisli

Relembrando nosso tipo `Writer` em Haskell:

```haskell
data Writer w a = Writer (a, w)
```

# Categoria Kleisli

Podemos criar uma instância de Functor fazendo:

```haskell
instance Functor (Writer w) where
  fmap f (Writer (a,w)) = Writer (f a, w)
```

# Categoria Kleisli

Utilizando esse tipo como saída de nossas funções temos que uma função `f :: a -> b` se torna uma função `f :: a -> Writer w b`.

# Categoria Kleisli

Fazendo `Writer w = m`, temos o padrão:

```
f :: a -> m b

(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

# Categoria Kleisli

Podemos então pensar na categoria Kleisli (**K**) como:

Partindo de uma categoria $C$ e um endofunctor $m$, a categoria **K** possui os mesmos objetos de $C$, mas com morfismos $a \to b$ sendo mapeados para $a \to mb$ na categoria $C$.

# Categoria Kleisli

Para ser uma categoria precisamos de um operador de composição (já temos o nosso peixe) e um morfismo identidade $a \to a$, que na categoria $C$ representa $a \to ma$.

# Categoria Kleisli

Com isso, dizemos que $m$ é um Monad se:

```haskell
class Monad m where
   (>=>)  :: (a -> m b) -> (b -> m c) -> (a -> m c)
   return :: a -> m a
```

# Categoria Kleisli

E apresenta as propriedades:

```
(f >=> g) >=> h = f >=> (g >=> h) = f >=> g >=> h
f >=> return = return >=> f = f
```

Em outras palavras, um Monad define como compor funções *embelezadas*.

# Categoria Kleisli

Como ficaria a instância completa do nosso Monad `Writer w`?

```
instance Monoid w => Monad (Writer w) where
  f >=> g = \a -> let Writer (b, s)  = f a
                      Writer (c, s') = g b
                  in  Writer (c, s `mappend` s')
  return a = Writer (a, mempty)
```

Indicando que `w` deve ser um Monoid, generalizamos a definição para outros tipos além de `String`.

# Dissecando o Peixe

Podemos perceber um padrão dentro do nosso operador >=> que nos ajudará a simplificá-lo:

```haskell
f >=> g = \a -> let mb = f a
                in  ...
```

# Dissecando o Peixe

O retorno da função deve ser uma função que recebe um `m b` e uma função `b -> m c` e retorna um `m c`.

# Dissecando o Peixe

Vamos criar o seguinte operador:

```haskell
(>>=) :: m a -> (a -> m b) -> m b
```

# Dissecando o Peixe

Que no caso do Monad `Writer w` fica:

```
(Writer (a,w)) >>= f = let Writer (b, w') = f a
                       in  Writer (b, w `mappend` w')
```

# Dissecando o Peixe

Tornando a definição do operador >=> como:

```
f >=> g = \a -> (f a) >>= g
```

Muito mais simples!

# Dissecando o Peixe

O operador >>= é conhecido como `bind` e define outra forma de instanciar um Monad:

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

# Dissecando o Peixe

Lembrando que um Monad também é um Functor, ele permite o uso de `fmap`.

# Dissecando o Peixe

Se tivermos duas funções:

```
f :: a -> m b

fmap :: (a -> b) -> m a -> m b
```

Ao aplicar `fmap f ma` sendo `ma` um monad `m a`, temos como saída um tipo `m (m b)`.

# Dissecando o Peixe

Precisamos de uma função que colapse a estrutura para apenas um Monad:

```
join :: m (m a) -> m a

ma >>= f = join (fmap f ma)
```

# Dissecando o Peixe

Com isso podemos definir um Monad também como:

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

# Dissecando o Peixe

A escolha de qual forma utilizar para instanciar um Monad depende da própria estrutura que queremos instanciar. Escolhemos o que for mais fácil definir e o resto é definido de graça!

# Dissecando o Peixe

A função `join` do nosso Monad `Writer w` fica:

```
join (Writer (Writer (a, w), w'))
               = Writer (a, w `mappend` w')
```

# Notação *do*

Relembrando um exemplo inicial da categoria Kleisli, tínhamos:

```haskell
notW :: Bool -> Writer Bool
notW b = (b, "not")

is_even :: Int -> Writer Bool
is_even x = (x `mod` 2 == 0, "even")

is_odd :: Int -> Writer Bool
is_odd = is_even >=> notW
```

# Notação *do*

O Haskell permite um *syntactic sugar* que transforma essa composição em uma notação similar ao paradigma imperativo:

```haskell
is_odd x = do
             ev <- is_even x
             is_odd ev
```

# Notação *do*

Que é *parseado* em:

```
is_even x >>= \ev -> is_odd ev
```

A notação `a <- f x` é lida como a recebe o resultado de `f x` sem a parte embelezada.

# Notação *do*

Para o caso do nosso Monad `Writer w` podemos também embelezar uma função automaticamente durante a notação `do`:

```
even x = x `mod` 2 == 0

tell :: w -> Writer w ()
tell s = Writer ((), s)

is_odd x = do tell "even"
              ev <- return (even x)
              tell "not"
              return (not ev)
```

# Notação *do*

Isso é traduzido para:

```
tell "even" >>= (\() -> return (even x)
            >>= \ev -> tell "not"
            >>= \() -> return (not ev))

Writer (f (), "even" ++ w')
Writer (g () (even x), "even" ++ "" + w'')
Writer (h () (even x) (), "even" ++ "" + "not" + w''')
Writer ((not.even) x, "even" ++ "" + "not" + "")
Writer ((not.even) x, "evennot")
```

# Notação *do*

Reparem que as funções `\() -> return x` não fazem uso do argumento de entrada, apenas altera o embelezamento da saída da função, podemos definir um operador específico para esses casos:

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

# Notação *do*

Fazendo com que o *desugaring* acima fique:

```
tell "even" >> return (even x)
            >>= \ev -> tell "not"
            >> return (not ev)
```

Esse operador descarta o argumento mas executa o *efeito colateral* da função.

# Notação *do*

Monads estão aos poucos aparecendo nas linguagens de programação orientadas a objetos.

A linguagem C++ está introduzindo o conceito de *resumable functions* que o Python já implementa com `yield` e Haskell com *continuation Monad*.

# Exemplos de Monads

# Monads e Efeitos

Uma das críticas frequentes ao Haskell é o fato de forçar a pureza das funções, o que teoricamente tornaria a linguagem por si só inútil na prática.

# Monads e Efeitos

Embora seja possível minimizar a quantidade de funções impuras de um programa, elas são necessárias pelo menos para a leitura dos dados de entrada e saída dos resultados.

Imagine um programa que apenas calcula o seno de 3 mas nunca apresenta o resultado!

# Monads e Efeitos

Algumas das utilidades de funções impuras conforme listado no artigo de Eugenio Moggi:

- **Parcialidade:** quando a computação de uma função pode não terminar.
- **Não determinismo:** quando a computação pode retornar múltiplos resultados dependendo do caminho da computação.
- **Efeitos colaterais:** quando a computação acessa ou altera um estado como
  - Read-only, leitura do ambiente
  - Write-only, escrita de um log
  - Read/Write

# Monads e Efeitos

Algumas das utilidades de funções impuras conforme listado no artigo de Eugenio Moggi:

- **Exceções:** funções parciais que podem falhar.
- **Continuações:** quando queremos salvar o estado de um programa e retomá-lo sob demanda.
- **Entrada e Saída Interativa**.

Todos esses efeitos podem ser tratados com embelezamento de funções e uso de Monads conforme veremos em seguida.

# Parcialidade

A parcialidade ocorre quando temos uma função que pode não terminar.

O Haskell inclui em todos os tipos o valor bottom $\perp$, com isso uma função `f :: a -> Bool` pode retornar `True`, `False` ou _|_.

Uma vez que Haskell dá preferência para avaliação preguiçosa, podemos compor funções que retornam $\perp$ contanto que nunca seja necessário avaliá-lo.

# Não-determinismo

Se uma função pode retornar diferentes resultados, dependendo de certos estados internos, ela é chamada de não-determinística.

# Não-determinismo

Por exemplo, a saída da função `getDate` depende do dia atual, assim como `random` depende do estado atual do gerador de números aleatórios.

Uma função que avalia a melhor jogada de um jogo de xadrez deve levar em conta todas as possibilidades de jogadas do seu adversário.

# Não-determinismo

Esse tipo de computação pode ser representada como uma lista contendo todas as possibilidades de saída.

Como no Haskell podemos trabalhar com listas infinitas (e a avaliação delas é preguiçosa), podemos usar o Monad `[]` para representar computações não-determinística.

# Não-determinismo

A instância Monad para listas é facilmente implementada pela função `join`:

```
instance Monad [] where
  join     = concat
  return x = [x]
```

# Não-determinismo

Essa definição é suficiente para o operador *bind*, que é definido como `as >>= k = concat (fmap k as)`.

# Não-determinismo

Em versões futuras do C++ teremos o `range comprehensions` que implementa uma lista preguiçosa similar ao Haskell:

```
template<typename T, typename Fun>
static generator<typename std::result_of<Fun(T)>::type>
bind(generator<T> as, Fun k)
{
    for (auto a : as) {
        for (auto x : k(a) {
            __yield_value x;
        }
    }
}
```

# Não-determinismo

E no Python, utilizamos os generators:

```python
def bind(xs, k):
    return (y for x in xs for y in k(x))
```

# Não-determinismo

Um exemplo interessante da expressividade do Monad lista no Haskell é o cálculo de todas as triplas pitagóricas, que pode ser implementada como:

```haskell
guard :: Bool -> [()]
guard True  = [()]
guard False = []

triples = do z <- [1..]
             x <- [1..z]
             y <- [x..z]
             guard (x^2 + y^2 == z^2)
             return (x, y, z)
```

# Não-determinismo

Reescrevendo utilizando *bind* percebemos melhor o que ele está fazendo:

```
[1..] >>= \z  -> [1..z]
      >>= \x  -> [x..z]
      >>= \y  -> guard (x^2 + y^2 == z^2)
      >>= \() -> return (x, y, z)

triples = concat (fmap fz [1..])
fz z    = concat (fmap fx [1..z])
fx x    = concat (fmap fy [x..z])
fy y    = concat (fmap f() (guard (x^2 + y^2 == z^2)))
f()     = [(x, y, z)]
```

# Não-determinismo

O Haskell também provê um *syntactic sugar* específico para listas, e essa mesma lista pode ser reescrita como:

```haskell
triples = [(x,y,z) | z <- [1..]
                   , x <- [1..z]
                   , y <- [x..z]
                   , x^2 + y^2 == z^2]
```

# Não-determinismo

No Python podemos fazer uma construção parecida como:

```
def triples(n):
  return ( (x,y,z) for z in range(1, n+1)
                   for x in range(1, z+1)
                   for y in range(x, z+1)
                   if (x**2 + y**2 == z**2))
```

# Read-only

A leitura de um estado externo de um ambiente genérico e é interpretado como uma função que recebe não só o argumento original como um argumento extra codificando o ambiente e:

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

O embelezamento está no argumento da função.

# Read-only

Ao aplicar o *currying* nessa função temos que ela é equivalente a `f :: a -> (e -> b)`, ou seja, `f :: a -> Reader e b`.

# Read-only

O Monad `Reader` faz o papel de manipulação de estados somente-leitura e vem equipado com as funções auxiliares `runReader`, que executa o `Reader` para um ambiente `e`, e `ask` que recupera o ambiente:

# Read-only

```haskell
data Reader e a = Reader (e -> a)

runReader :: Reader e a -> e -> a
runReader (Reader f) e = f e

ask :: Reader e e
ask = (Reader id)
```

# Read-only

Note que a definição do `Reader` e todas as funções que a utilizam são essencialmente puras, dada uma tabela grande o suficiente poderíamos memoizar todas as entradas e saídas possíveis para todo estado possível do ambiente `e`.

# Read-only

A definição de Monad para o `Reader e` é:

```haskell
instance Monad (Reader e) where
  -- (Reader e a) -> (a -> Reader e b) -> (Reader e b)
  -- (Reader f) >>= k = \e -> runReader (k (f e)) e
  ra >>= k = Reader (\e -> let a  = runReader ra e
                               rb = k a
                           in  runReader rb e)

  return a = Reader (\e -> a)
```

# Read-only

O *bind* do `Reader e` faz os seguintes passos:

1. executa `ra` no ambiente atual `e`, capturando o resultado puro `a`.
2. aplica a função `k` em `a` que retorna um `Reader e b`.
3. executa esse `Reader` no ambiente passado como argumento.

# Read-only

A função `return` simplesmente cria uma função constante que sempre retorna um valor a para qualquer ambiente (verifique a propriedade `ra >>= return = ra`).

# Read-only

Imagine que temos um algoritmo que possui uma estrutura de configuração utilizada por uma função principal e funções auxiliares.

# Read-only

```
data Config = Conf { alg :: String
                   , thr :: Double
                   , it  :: Int
                   }

go :: Int -> [Double] -> [Double]
go _ []     = []
go 0 xs     = xs
go i (x:xs) = (i' * x) : go (i-1) xs
  where i' = fromIntegral i

filterLess thr xs = filter (<thr) xs
```

# Read-only

```
f2 :: Config -> [Double] -> [Double]
f2 cfg xs = f3 cfg $ filterLess (thr cfg) xs

f3 :: Config -> [Double] -> [Double]
f3 cfg xs = go (it cfg) xs

algorithm :: Config -> [Double] -> [Double]
algorithm cfg xs | alg cfg == "f2" = f2 cfg xs
                 | otherwise       = f3 cfg xs
```

# Read-only

Para evitar ter que passar o parâmetro de configuração para todas as funções podemos definir `cfg` como uma variável global acessível por todas as funções.

# Read-only

# Read-only

Porém, se precisarmos carregar essas configurações de um arquivo externo, não podemos deixá-la como global no Haskell.

Nas linguagens que permitem o uso de variáveis globais, todas as funções que utilizam a estrutura de configuração em funções se tornariam impuras.

# Read-only

Utilizando o Reader Monad podemos resolver essa situação da seguinte forma:

```haskell
askFor f = fmap f ask

f3 :: [Double] -> Reader Config [Double]
f3 xs = askFor it >>= runGo
  where runGo t = return (go t xs)

f2 :: [Double] -> Reader Config [Double]
f2 xs = askFor thr >>= gof3
  where gof3 t = f3 (filterLess t xs)


algorithm :: [Double] -> Reader Config [Double]
algorithm xs = askFor alg >>= choice
  where choice a | a == "f2" = f2 xs
                 | otherwise = f3 xs
```

# Read-only

Nesse código o nosso ambiente é caracterizado por um tipo `Config`, que armazena a configuração do algoritmo.

# Read-only

O comando `fmap f ask` cria uma função que recebe um `Config` e retorna o parâmetro especificado por `f`.

# Read-only

O segundo parâmetro do operador `bind` é uma função que recebe um `f Config` e retorna o resultado esperado do tipo `[Double]`.

# Read-only

Com isso, o operador `bind` recebe uma função `Config -> f Config` e uma função `f Config -> [Double]` e transforma em uma função `Config -> [Double]`.

# Read-only

Utilizando do-notation o mesmo programa fica:

```
f3 :: [Double] -> Reader Config [Double]
f3 xs = do
  t <- askFor it
  return (go t xs)

f2 :: [Double] -> Reader Config [Double]
f2 xs = do
  t  <- askFor thr
  f3 (filterLess t xs)

algorithm :: [Double] -> Reader Config [Double]
algorithm xs = do
  alg <- askFor alg
  if alg == "f2"
  then f2 xs
  else f3 xs
```

Para executar o algoritmo precisamos fazer `runReader (algorithm xs) c`, sendo c a variável contendo a

# Read-only

As instruções `fmap f ask` recupera um elemento da nossa configuração.

Notem que a variável contendo a configuração não é passada diretamente para nenhum das funções do algoritmo, qualquer alteração que seja feita nessa estrutura ou no uso dela, não criará um efeito cascata de alterações no código.

# Read-only

Em Python podemos fazer algo muito similar:

```
from collections import namedtuple

Conf = namedtuple('Conf', ['alg', 'thr', 'it'])

def alg(c):
  return c.alg
def thr(c):
  return c.thr
def it(c):
  return c.it

def ask(e):
  return e

def askFor(f):
  return Reader(ask).fmap(f)
```

# Read-only

```
class Reader():
  # Reader e a
  def __init__(self, fun = None):
    self.r = fun

  def run(self, e):
    return self.r(e)

  # (a -> b) -> Reader e a -> Reader e b
  def fmap(self, f):
    return Reader(lambda e: f(self.r(e)))

  def unit(self, x):
    return Reader(lambda e: x)

  # Reader e a -> (a -> Reader e b) -> Reader e b
  def bind(self, fab):
    def f(e):
      a = self.r(e)
      rb = fab(a)
      return rb.run(e)
    return Reader(f)
```

# Read-only

```
def go(it, xs):
  ys = []
  for x in xs:
    ys.append(it*x)
    it = it - 1
  return ys

def f3(xs):
  runGo = lambda t: Reader().unit(go(t, xs))
  return (askFor(it)
          .bind(runGo))

def f2(xs):
  gof3 = lambda t: f3(filter(lambda x: x<t, xs))
  return (askFor(thr)
          .bind(gof3))

def algorithm(xs):
  f = {"f2" : f2, "f3" : f3}
  choice = lambda algo: f[algo](xs)
  return (askFor(alg)
          .bind(choice))
```

# Read-only

```
c = Conf("f2", 2.5, 5)
print(algorithm(range(1,11))
      .run(c))
```

# Write-only

Analogamente, podemos definir um estado *Write-only* como nosso Monad `Writer` equipado com uma função `runWriter` e a função `tell`:

```haskell
data Writer w a = Writer (a, w)

runWriter :: Writer w a -> (a, w)
runWriter (Writer aw) = aw

tell :: w -> Writer w ()
tell s = Writer ((), s)
```

# Write-only

```haskell
instance (Monoid w) => Monad (Writer w) where
  -- (Writer w a) -> (a -> Writer w b) -> (Writer w b)
  (Writer (a, w) >>= k = let (b, w') = runWriter (k a)
                         in  Writer (b, w `mappend` w')

  return a = Writer (a, mempty)
```

# Write-only

Motivamos o uso do `Writer` Monad com a implementação de um **traço de execução**:

```
notW :: Bool -> Writer Bool
notW b = (b, "not")

is_even :: Int -> Writer Bool
is_even x = (x `mod` 2 == 0, "even")

is_odd :: Int -> Writer Bool
is_odd = is_even >=> notW
```

# Write-only

Podemos generalizar ainda mais ao utilizar nossa função `tell`:

```
trace :: Int -> Writer String Bool
trace x = do
  tell "even"
  b <- return (even x)
  tell "not"
  return (not b)
```

# Write-only

Ou também:

```haskell
trace2 :: Int -> Writer String Bool
trace2 x = (evenW >=> notW) x
  where
    evenW :: Int -> Writer String Bool
    evenW x = tell "even" >> return (even x)

    notW :: Bool -> Writer String Bool
    notW  b = tell "not" >> return (not b)
```

# Write-only

Veja que dessa forma, a função `tell` pode transformar uma função `a -> b` para `a -> Writer s b` automaticamente, reduzindo as alterações necessárias em nosso programa.

# Write-only

Em C++ temos:

```
template<class A>
Writer<A> identity(A x) {
    return make_pair(x, "");
}

auto const compose = [](auto m1, auto m2) {
    return [m1, m2](auto a) {
        auto p1 = m1(a);
        auto p2 = m2(p1.first);
        return make_pair(p2.first, p1.second + p2.second);
    };
};
```

# Write-only

```cpp
Writer<bool> not(bool b) {
    return make_pair(!b, "not ");
}

Writer<bool> is_even(int x) {
    return make_pair(x%2==0, "even ");
}

Writer<bool> is_odd(int x) {
    return compose(is_even, not)(x);
}
```

# Write-only

Em Python:

```
def id_writer(x):
    return Writer(x, "")

def compose_writer(m1, m2):
    def f(a):
        b, s1 = m1(a)
        c, s2 = m2(b)
        return Writer(c, s1+s2)
    return f

def not(b):
    return (not b, "not")

def is_even(x):
    return (x%2==0, "even")

def is_odd(x):
    return compose_writer(is_even, not)(x)
```

# State

Um estado simplesmente é um ambiente *e* que permite leitura e escrita, ou seja, é a combinação dos Monads `Reader` e `Writer`.

Uma função `f :: a -> b` é embelezada para `f :: (a, s) -> (b, s)`, e utilizando *currying* temos `f :: a -> (s -> (b, s))`.

# State

```haskell
data State s a = State (s -> (a, s))
               = Reader s (Writer s a)

runState :: State s a -> s -> (a, s)
runState (State f) s = f s

get :: State s s
get = State (\s -> (s, s))

put :: s -> State s ()
put s' = State (\s -> ((), s'))
```

# State

Com isso temos a capacidade de ler ou alterar um estado.

A instância de Monad nesse caso fica muito parecida com o Monad `Reader`, exceto que tomamos o cuidado de passar o novo estado para o próximo `runState`.

# State

```haskell
instance Monad (State s) where
  -- (State s a) -> (a -> State s b) -> (State s b)
  sa >>= k = State (\s -> let (a, s') = runState sa s
                          in runState (k a) s')

  return a = State (\s -> (a, s))
```

# State

Uma aplicação desse Monad é na manipulação de números aleatórios em que queremos que o estado do gerador seja atualizado a cada chamada da função `random`.

# State

Vamos exemplificar com uma função que alterar uma lista de `Bool`, invertendo cada um de seus elementos, caso um certo valor aleatório seja `< 0.3`.

# State

```haskell
randomSt :: (RandomGen g) => State g Double
randomSt = state (randomR (0.0, 1.0))

mutation :: [Bool] -> State StdGen [Bool]
-- se a lista estiver vazia, nada a fazer
mutation [] = return []

mutation (b:bs) = do
    -- aplica o algoritmo no restante da lista,
    -- o estado atual do gerador é passado implicitamente
    -- para a função
    bs' <- mutation bs

    -- sorteia um valor aleatório e altera de acordo
    p   <- randomSt
    if p < 0.3
    then return (not b : bs')
    else return (b : bs')
```

# State

Alternativamente, sem o do-notation ficaria como:

```haskell
mutation (b:bs) = (mutation >=> choice) bs
  where
     choice bs' = randomSt >>= concat bs'
     concat bs' p = if p < 0.3
                      then return (not b : bs')
                      else return (b : bs')
```

# State

Em Python podemos escrever:

```python
import random

class State:
  # State s -> (a, s)
  def __init__(self, f = None):
    self.r = f

  def run(self, s):
    return self.r(s)

  def unit(self, x):
    return State(lambda s: (x, s))

  def bind(self, k):
    def f(s):
      (a, sn) = self.run(s)
      return k(a).run(sn)
    return State(f)
```

# State

```
getSt = State(lambda s: (s, s))
setSt = State(lambda s: (None, s))

def mutation(bs):
  if len(bs)==0:
    return State().unit([])

  b = bs.pop()

  myRandST = State(myRand)

  def ifthenelse(bsm, p):
    if p < 0.3:
      return State().unit([not b] + bsm)
    else:
      return State().unit([b] + bsm)

  return (mutation(bs)
          .bind(lambda bsm: myRandST
                            .bind(lambda p: ifthenelse(bsm, p))
               )
         )
```

# State

```
def myRand(s):
  random.setstate(s)
  x = random.random()
  return x, random.getstate()

print(mutation([True, True, False, True])
      .run(random.getstate())[0])
```

# Exceções

O tratamento de exceções é necessário quando uma função é parcial e pode falhar, por exemplo quando faz o processo de divisão em um valor que pode ser igual a zero.

A forma mais simples de tratar exceções no Haskell é através do Monad Maybe em que uma computação que falha é sinalizada com o valor `Nothing`.

# Exceções

```haskell
instance Monad Maybe where
  Nothing >>= k = Nothing
  Just a  >>= k = k a
  return a = Just a
```

# Exceções

Vimos um exemplo anteriormente em que a composição de duas funções que podem falhar não dá continuidade no processamento caso a primeira falhe:

```haskell
safeRoot :: Double -> Maybe Double
safeRoot x | x < 0     = Nothing
           | otherwise = Just (sqrt x)

safeReciprocal:: Double -> Maybe Double
safeReciprocal 0 = Nothing
safeReciprocal x = Just (1.0 / x)

sequencia x = (safeReciprocal x) >>= safeRoot
-- ou sequencia = safeReciprocal >=> safeRoot
```

# Exceções

Em Python:

```python
from math import sqrt

def safe_root(x):
    return sqrt(x) if x>=0 else None

def safe_reciprocal(x):
    return 1.0/x if x!=0 else None

def fish(f, g):
    def h(x):
        z = f(x)
        if z is None:
            return z
        else:
            return g(z)
    return h

sequencia = fish(safe_root, safe_reciprocal)
```

# IO

Quando dizemos que Haskell é uma linguagem de programação puramente funcional e **todas** suas funções são puras, a primeira questão que vem na mente é de como as funções de entrada e saída são implementadas.

Como as funções `getChar, putChar` podem ser puras se elas dependem do efeito colateral? Como é possível compor funções puras com a saída de `getChar` se a saída é, teoricamente, indeterminada?

# IO

O segredo das funções de manipulação de `IO` é que elas tem seus valores guardados dentro de um container (o `IO` Monad) que nunca pode ser aberto.

Ou seja, criamos funções que lidam com `Char` sem saber exatamente quem é esse caracter.

# IO

Podemos imaginar o Monad `IO` como uma caixa quântica contendo uma superposição de todos os valores possíveis de um tipo.

Toda chamada de função para esse tipo é **jogada lá dentro** e executada pelo sistema operacional quando apropriado.

# IO

As assinaturas de `getChar` e `putChar` são:

```
getChar :: IO Char -- () ->  IO Char

putChar :: Char :: IO ()
```

# IO

Note que a implementação da instância de Functor e Monad para `IO` é implementada internamente no Sistema Operacional e não temos um `runIO` que nos devolve um valor contido no container.

Ao fazer `fmap f getChar` a função será executada no retorno de `getChar` mas não poderemos ver seu resultado.

# IO

Uma outra forma de pensar no `IO` é como um tipo `State`:

```
data IO a = Mundo -> (a, Mundo) = State Mundo a
```

# IO

A sequência:

```haskell
do putStr "Hello"
   putStr "World"
```

Causa uma dependência funcional entre as duas funções de tal forma que elas serão executadas na sequência.

# Comonads

# Comonads

A categoria oposta da Kleisli, denominada **co-Kleisli** leva ao conceito de **Comonads**.

Agora temos endofunctors `w` e morfismos do tipo `w a -> b`.

# Comonads

Queremos definir um operador de composição para eles, da mesma forma que definimos o operador >=>:

```haskell
(=>=) :: (w a -> b) -> (w b -> c) -> (w a -> c)
```

# Comonads

Da mesma forma, nosso morfismo identidade é similar ao `return` mas com a seta invertida:

```haskell
extract :: w a -> a
```

Chamamos essa função de `extract` pois ela permite extrair um conteúdo do functor `w` (nosso container).

# Comonads

O oposto de nosso operador `bind` deve ter a assinatura `(w a -> b) -> w a -> w b`:

Dada uma função que retira um valor do tipo `a` de um container transformando em um tipo `b` no processo, retorne uma função que, dado um `w a` me retorne um `w b`.

Essa função é chamada de `extend` ou na forma de operador `=>>`.

# Comonads

Finalmente, o oposto de `join` é o `duplicate`, ou seja, insere um container dentro de outro container, sua assinatura é:

```
w a -> w (w a)
```

# Comonads

Nesse ponto, podemos perceber a dualidade entre Monad e Comonad.

# Comonads

Em um Monad criamos uma forma de colocar um valor dentro de um container, através do `return`, mas sem garantias de que poderemos retirá-lo de lá.

Envolvemos nosso valor em um contexto computacional, que pode ficar escondido até o final do programa, como vimos no comportamento do Monad IO.

# Comonads

Já um Comonad, nos traz uma forma de retirar um valor de um container, através de `extract`, sem prover uma forma de colocá-lo de volta.

Além disso, ele permite uma computação contextual de um elemento do container, ou seja, podemos focar em um elemento e manter todo o contexto em volta dele (e já vimos isso nos tipos buracos!).

# Comonads

Então nossa classe Comonad é definida por:

```
class Functor w => Comonad w where
    (=>=)     :: (w a -> b) -> (w b -> c) -> (w a -> c)
    extract   :: w a -> a
    (=>>)     :: (w a -> b) -> w a -> w b
    duplicate :: w a -> w (w a)
```

# Reader Comonad

Relembrando o Reader Monad que definimos no post anterior:

```haskell
data Reader e a = Reader (e -> a)

instance Monad (Reader e) where
  -- (Reader e a) -> (a -> Reader e b) -> (Reader e b)
  -- (Reader f) >>= k = \e -> runReader (k (f e)) e
  ra >>= k = Reader (\e -> let a  = runReader ra e
                               rb = k a
                           in  runReader rb e)

  return a = Reader (\e -> a)
```

# Reader Comonad

A versão embelezada dos morfismos na categoria Kleisli é `a -> Reader e b` que pode ser traduzido para `a -> (e -> b)` e colocado na forma *curry* de `(a, e) -> b`.

# Reader Comonad

Com isso, conseguimos definir o Comonad de `Writer` e como o complemento do Monad Reader:

```
instance Comonad (Writer e) where
  (=>=) :: (Writer e a -> b) -> (Writer e b -> c) -> (Writer e a
  f =>= g = \(Writer e a) -> let b = f (Writer e a)
                                 c = g (Writer e b)
                             in c

  extract (Writer e a) = a
```

# Reader Comonad

Basicamente, a função `extract` ignora o ambiente definido por `e` e retorna o valor `a` contido no container.

O operador de composição simplesmente pega duas funções que recebem tuplas como parâmetros, sendo a primeira do mesmo tipo, e aplica sequencialmente utilizando o mesmo valor de `e` nas duas chamadas (afinal `e` é *read-only*).

# Definições padrão

Examinando o operador =>= temos como argumentos `f :: w a -> b` e `g :: w b -> c` e precisamos gerar uma função `h :: w a -> c`. Para gerar um valor do tipo `c`, dado `f, g`, a única possibilidade é aplicar `g` em um tipo `w b`:

```
f =>= g = g ...
```

# Definições padrão

Tudo que temos a disposição é uma função `f` que produz um `b`. Precisamos então de uma função com a assinatura `(w a -> b) -> w a -> w b`, que é nossa função `extend` (=>>). Com isso temos a definição padrão:

```
f =>= g = \wa -> g . (f ==>) wa
-- ou
-- f =>= g = g . (f =>>)
```

# Definições padrão

Da mesma forma, pensando no operador `=>>` com assinatura `(w a -> b) -> w a -> w b`, percebemos que não tem como obter diretamente um `w b` ao aplicar a função argumento em `w a`.

# Definições padrão

Porém, uma vez que `w` necessariamente é um Functor, temos a disposição a função `fmap :: (c -> b) -> w c -> w b`, que ao fazer com que `c = w a`, temos `(w a -> b) -> w (w a) -> w b`.

Se conseguirmos produzir um `w (w a)`, podemos utilizar `fmap` para implementar =>>.

# Definições padrão

Temos a função duplicate, o que faz com que:

```haskell
class Functor w => Comonad w where
    extract   :: w a -> a

    (=>=)     :: (w a -> b) -> (w b -> c) -> (w a -> c)
    f =>= g = g . (=>>)

    (=>>)     :: (w a -> b) -> w a -> w b
    f =>> wa = fmap f . duplicate

    duplicate :: w a -> w (w a)
    duplicate = (id =>>)
```

# Definições padrão

A ideia de um Comonad é que você possui um container com um ou mais valores de `a` e que existe uma noção de *valor atual* ou *foco* em um dos elementos.

Esse valor atual é acessado pela função `extract`.

# Definições padrão

O operador *co-peixe* (=>=) faz alguma computação, definida pelas funções compostas, no valor atual porém tendo acesso a tudo que está em volta dele.

## Definições padrão

Já a função `duplicate` cria diversas versões de `w a` cada qual com um foco diferente, ou seja, temos todas as possibilidades de foco para aquela estrutura.

# Definições padrão

Finalmente, a função `extend` (=>>), primeiro gera todas as versões da estrutura via `duplicate` para então aplicar a função co-Kleisli através do `fmap`, ou seja, ela aplica a função em todas as possibilidades de foco.

# Stream de Dados

Podemos definir um Stream de dados como uma lista infinita não-vazia:

```haskell
data Stream a = Cons a (Stream a)

instance Functor Stream where
  fmap f (Cons a as) = Cons (f a) (fmap f as)
```

# Stream de Dados

Notem que essa estrutura possui naturalmente um foco em seu primeiro elemento. Com isso podemos definir a função `extract` simplesmente como:

```
extract (Cons a _) = a
```

# Stream de Dados

Lembrando que a função `duplicate` deve gerar uma Stream de Streams, cada uma focando em um elemento dela, podemos criar uma definição recursiva como:

```
duplicate (Cons a as) = Cons (Cons a as) (duplicate as)
```

# Stream de Dados

Cada chamada recursiva cria uma Stream com a cauda da lista original. Com isso temos as funções necessárias para criar uma instância de Comonad para Streams:

```
instance Comonad Stream where
  extract (Cons a _) = a
  duplicate (Cons a as) = Cons (Cons a as) (duplicate as)
```

# Stream de Dados

Como exemplo de aplicação vamos criar uma função que calcula a média móvel de um stream de dados. Começamos com a definição da média entre os *n* próximos elementos:

```haskell
sumN :: Num a => Int -> Stream a -> a
sumN n (Cons a as) | n <= 0    = 0
                   | otherwise = a + sumN (n-1) as

avgN :: Fractional a => Int -> Stream a -> a
avgN n as = (sumN n as) / (fromIntegral n)
```

# Stream de Dados

Notem que avgN tem assinatura `Int -> (w a -> a)`, para gerarmos uma Stream de médias móveis queremos algo como `Int -> (w a -> a) -> w a -> w a`, que remete a assinatura de =>>:

```haskell
movAvg :: Fractional a => Int -> Stream a -> Stream a
movAvg n as = (avgN n) =>> as
```

# Stream de Dados

Com isso, a função `avgN n` será aplicada em cada foco de `as` gerando uma nova Stream contendo apenas os valores das médias.

# Stream de Dados

Podemos implementar o Comonad Stream em Python criando uma lista ligada em que o próximo elemento é definido por uma função geradora passada como parâmetro para o construtor de objetos da classe.

# Stream de Dados

```python
class Stream:
  '''
  Data Stream in Python: infinite stream of data with generator
  -- equivalent to Haskell Stream a = Stream a (Stream a)
  x: initial value
  f: generator function (id if None)
  g: mapped function (Fucntor fmap)
  '''
  def __init__(self, x=1, f=None, g=None):
    idfun = lambda x: x

    self.f = idfun if f is None else f
    self.g = idfun if g is None else g
    self.x = x

  def next(self):
    ''' the tail of the Stream '''
    return Stream(self.f(self.x), self.f, self.g)
```

# Stream de Dados

```python
  def extract(self):
    ''' returns mapped current value '''
    return self.g(self.x)

  def duplicate(self):
    ''' a Stream of Streams '''
    def nextS(xs):
      return self.next()
    return Stream(self, nextS)

  def fmap(self, g):
    ''' Functor instance '''
    return Stream(self.x, self.f, g)

  def extend(self, g):
    ''' comonad extend =>> '''
    return self.duplicate().fmap(g)
```

# Stream de Dados

```python
def avgN(n, xs):
  s = 0
  for i in range(n):
    s += xs.extract()
    xs = xs.next()
  return s/n

def movAvg(n, xs):
  movAvgN = partial(avgN, n)
  return xs.extend(movAvgN)

def f(x):
  return x+1

xs = Stream(1, f)
print(xs)
print(movAvg(5, xs))
```

# Store Comonad

Relembrando o `State` Monad visto anteriormente, ele foi definido como a composição `(Reader s) . (Writer s)`:

```haskell
data State s a = State (s -> (a, s))
               = Reader s (Writer s a)
```

Isso foi possível pois os Functors `Reader` e `Writer` são adjuntos.

# Store Comonad

De forma análoga podemos fazer a composição complementar para criarmos um Comonad:

```haskell
data Store s a = Store (s -> a) s
               = Writer s (Reader s a)
```

# Store Comonad

A instância de Functor para esse Comonad é simplesmente a composição da função definida por `Reader s a`:

```haskell
instance Functor (Store s) where
  fmap g (Store f s) = Store (g . f) s
```

# Store Comonad

A assinatura de `extract` deve ser `Store s a -> a`, sendo que o tipo `Store` armazena uma função `s -> a` e um `s`, basta aplicar a função no estado `s` que ele armazena.

# Store Comonad

Por outro lado, a função `duplicate` pode se aproveitar da aplicação parcial na construção de um valor definindo:

```haskell
instance Comonad (Store s) where
  extract (Store f s)   = f s
  duplicate (Store f s) = Store (Store f) s
```

# Store Comonad

Podemos imaginar o Comonad `Store` como um par que contém um container (a função `f`) que armazena diversos elementos do tipo `a` indexados pelo tipo `s` (i.e., *array* associativa) e um `s` que indica o foco atual da estrutura (como em um Zipper, visto anteriormente).

# Store Comonad

Nessa interpretação temos que `extract` retorna o elemento `a` na posição atual `s` e `duplicate` simplesmente cria infinitas cópias desse container de tal forma que cada cópia está deslocada em $n$ posições para direita ou para a esquerda.

# Store Comonad

Como exemplo, vamos implementar o automato celular 1D conforme descrito por Wolfram.

# Store Comonad

Esse automato inicia com uma lista infinita indexada por valores inteiros (positivos e negativos) e centralizada em 0.

A lista contém inicialmente o valor 1 na posição central e 0 em todas as outras posições.

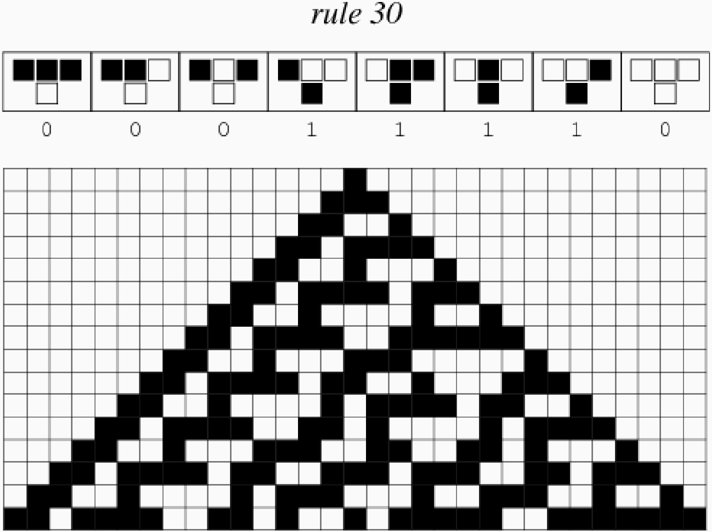
# Store Comonad

A cada passo da iteração, a lista é atualizada através das `regras n` em que $0 \leq n \leq 255$.

A numeração da regra codificam um mapa de substituição para o número binário formado pela subslita composta do valor atual e de seus dois vizinhos.

# Store Comonad

Por exemplo, a regra 30 codifica:



# Store Comonad

Podemos implementar esse autômato utilizando um `Store Int Int`, primeiro definindo a função que aplica a regra:

```haskell
rule :: Int -> Store Int Int -> Int
rule x (Store f s) = (succDiv2 !! bit) `rem` 2
  where
    -- qual o bit que devemos recuperar
    bit      = (f (s+1)) + 2*(f s) + 4*(f (s-1))
    -- divisao sucessiva de x por 2
    succDiv2 = iterate (`div` 2) x
```

# Store Comonad

Fazendo uma aplicação parcial do número da regra, a assinatura da função fica: `Store Int Int -> Int` que deve ser aplicada em um `Store Int Int` para gerar a próxima função de indexação. Isso sugere o uso de `extend` (=>>):

```
nextStep rl st = rl =>> st
```

# Store Comonad

Mas queremos uma aplica sucessiva dessa regra infinitamente, para isso podemos utilizar `iterate`:

```
wolfram :: (Store Int Int -> Int)
        -> Store Int Int
        -> [Store Int Int]
wolfram rl st = iterate (rl =>> st)
```

# Store Comonad

A representação inicial de nosso ambiente é feita por:

```haskell
f0 :: Int -> Int
f0 0 = 1
f0 _ = 0

fs :: Store Int Int
fs = Store f0 0
```

# Store Comonad

Finalmente, podemos capturar um certo instante do nosso autômato simplesmente acessando o elemento correspondente da lista:

```
wolf30 = wolfram (rule 30) fs

fifthIteration = wolf30 !! 5
```

# Store Comonad

E podemos imprimir nosso ambiente criando uma instância de Show:

```haskell
instance (Num s, Enum s, Show a) => Show (Store s a) where
  show (Store f s) = show [f (s+i) | i <- [-10 .. 10]]


main = print (take 5 wolf30)
```

# Store Comonad

Como referência, o código em Python ficaria:

```python
from functools import partial
import itertools

class Store:
  def __init__(self, f, s):
    self.f = f
    self.s = s

  def fmap(self, g):
    f = lambda s: g(self.f(s))
    return Store(f, self.s)

  def extract(self):
    return self.f(self.s)

  def duplicate(self):
    return Store(lambda s: Store(self.f, s), self.s)

  def extend(self, g):
    return self.duplicate().fmap(g)
```

# Store Comonad

```python
def rule(x, fs):
  succDiv2 = [x]
  while x!=0:
    x = x//2
    succDiv2.append(x)
  bit = fs.f(fs.s+1) + 2*fs.f(fs.s) + 4*fs.f(fs.s-1)
  if bit >= len(succDiv2):
    return 0
  return succDiv2[bit] % 2

def wolfram(rl, fs):
  while True:
    yield fs
    fs = fs.extend(rl)
```

# Store Comonad

```python
def f0(x):
  if x==0:
    return 1
  return 0

fs = Store(f0, 0)

wolf30 = wolfram(partial(rule, 30), fs)
top5 = itertools.islice(wolf30, 6)

for w in top5:
  print(w)
```

# Atividades para Casa

# Atividades para Casa

1. a