Coder Decoder

Functional Programmer Lingo Explained, with Pictures

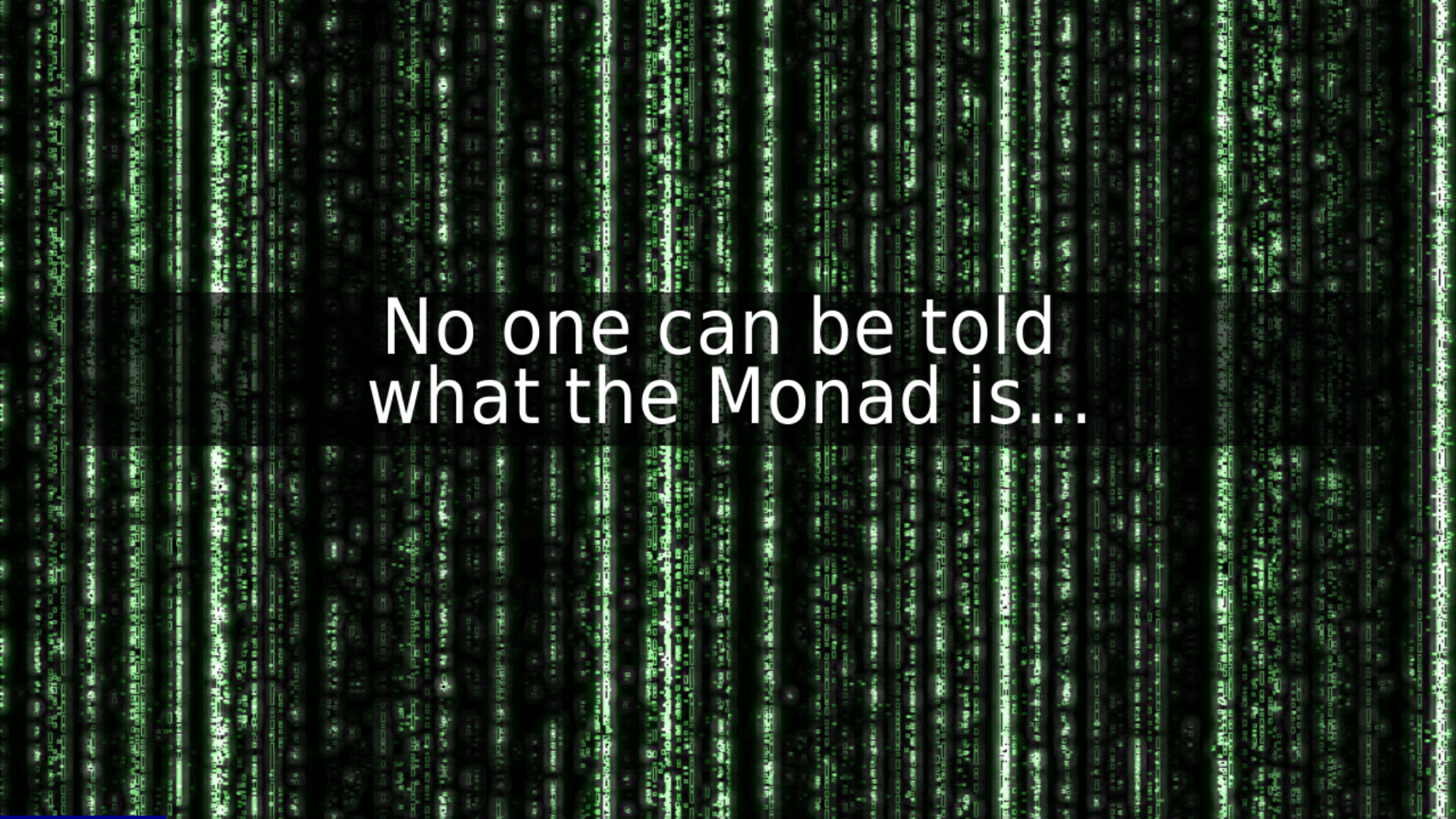
Katie Miller (@codemiller)
OpenShift Developer Advocate at Red Hat


OHMAGIF.COM


GIFAK.NET

Mission

Caveats


No one can be told
what the Monad is...

Setting the Scene

Haskell

Haskell

```
list1 = [1,2,3]
list2 = [4,5,6]
```

Haskell

```
list1 = [1,2,3]
list2 = [4,5,6]

list1 ++ list2
```

Haskell

```
list1 = [1,2,3]
list2 = [4,5,6]

list1 ++ list2
-- [1,2,3,4,5,6]
```

Haskell

```
list1 = [1,2,3]
list2 = [4,5,6]

list1 ++ list2
-- [1,2,3,4,5,6]

catcon list1 list2
```

Haskell

```
list1 = [1,2,3]
list2 = [4,5,6]

list1 ++ list2
-- [1,2,3,4,5,6]

catcon list1 list2
-- [4,5,6,1,2,3]
```

Haskell

```
list1 = [1,2,3]
list2 = [4,5,6]

list1 ++ list2
-- [1,2,3,4,5,6]

catcon list1 list2
-- [4,5,6,1,2,3]

catcon :: [Int] -> [Int] -> [Int]
```

Haskell

```haskell
list1 = [1,2,3]
list2 = [4,5,6]

list1 ++ list2
-- [1,2,3,4,5,6]

catcon list1 list2
-- [4,5,6,1,2,3]

catcon :: [Int] -> [Int] -> [Int]
```

Haskell

```haskell
list1 = [1,2,3]
list2 = [4,5,6]

list1 ++ list2
-- [1,2,3,4,5,6]

catcon list1 list2
-- [4,5,6,1,2,3]

catcon :: [Int] -> [Int] -> [Int]
```

Haskell

```
list1 = [1,2,3]
list2 = [4,5,6]

list1 ++ list2
-- [1,2,3,4,5,6]

catcon list1 list2
-- [4,5,6,1,2,3]

catcon :: Num a => [a] -> [a] -> [a]
```

Haskell

```
list1 = [1,2,3]
list2 = [4,5,6]

list1 ++ list2
-- [1,2,3,4,5,6]

catcon list1 list2
-- [4,5,6,1,2,3]

catcon :: Num a => [a] -> [a] -> [a]
```

Haskell

```
list1 = [1,2,3]
list2 = [4,5,6]

list1 ++ list2
-- [1,2,3,4,5,6]

catcon list1 list2
-- [4,5,6,1,2,3]

catcon :: Num a => [a] -> [a] -> [a]
```

```
ghci> :info Num
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  (-) :: a -> a -> a
  ...
  -- Defined in `GHC.Num'
instance Num Integer -- Defined in `GHC.Num'
instance Num Int -- Defined in `GHC.Num'
instance Num Float -- Defined in `GHC.Float'
instance Num Double -- Defined in `GHC.Float'
```

Haskell

```haskell
list1 = [1,2,3]
list2 = [4,5,6]

list1 ++ list2
-- [1,2,3,4,5,6]

catcon list1 list2
-- [4,5,6,1,2,3]

catcon :: Num a => [a] -> [a] -> [a]
```

```
interface Num<A> {
  A add(A x, A y);
  A multiply(A x, A y);
  A subtract(A x, A y);
  ...
}
```

Haskell

```haskell
list1 = [1,2,3]
list2 = [4,5,6]

list1 ++ list2
-- [1,2,3,4,5,6]

catcon list1 list2
-- [4,5,6,1,2,3]

catcon :: [a] -> [a] -> [a]
```

Haskell

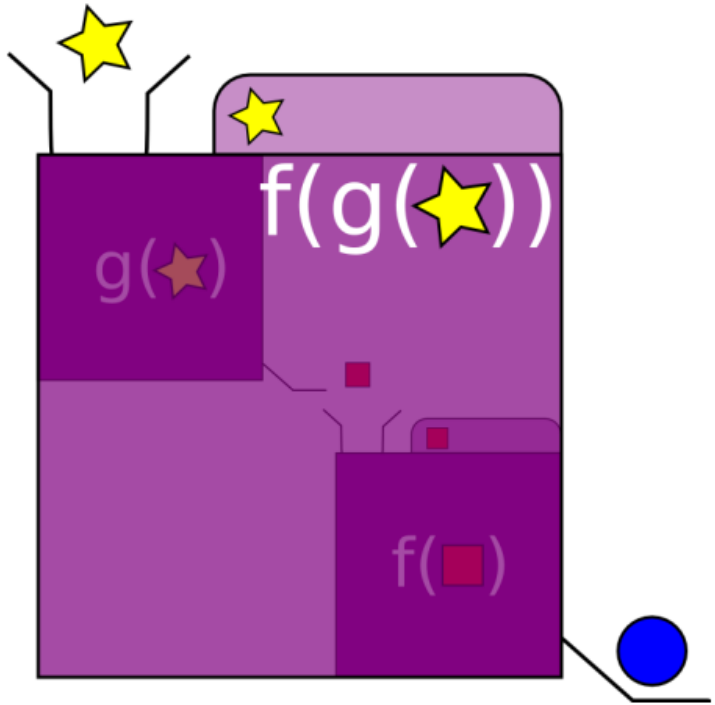
```
list1 = [1,2,3]
list2 = [4,5,6]

list1 ++ list2
-- [1,2,3,4,5,6]

catcon list1 list2
-- [4,5,6,1,2,3]

catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs
```

Haskell

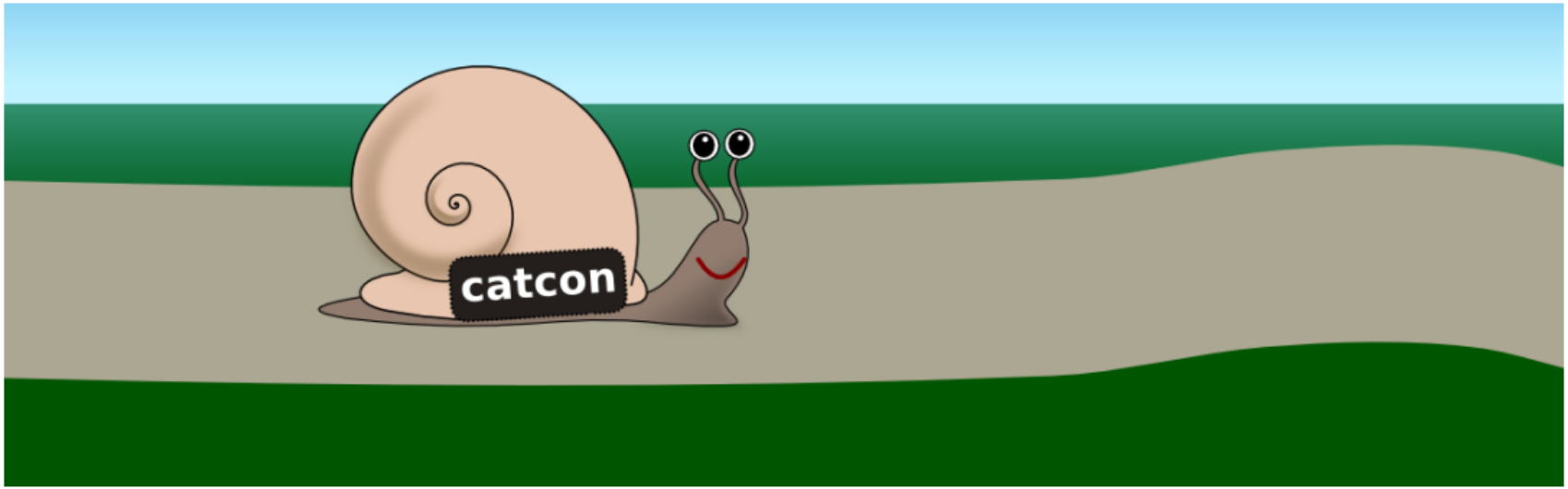
```
list1 = [1,2,3]
list2 = [4,5,6]

list1 ++ list2
-- [1,2,3,4,5,6]

catcon list1 list2
-- [4,5,6,1,2,3]

catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs
```

Haskell

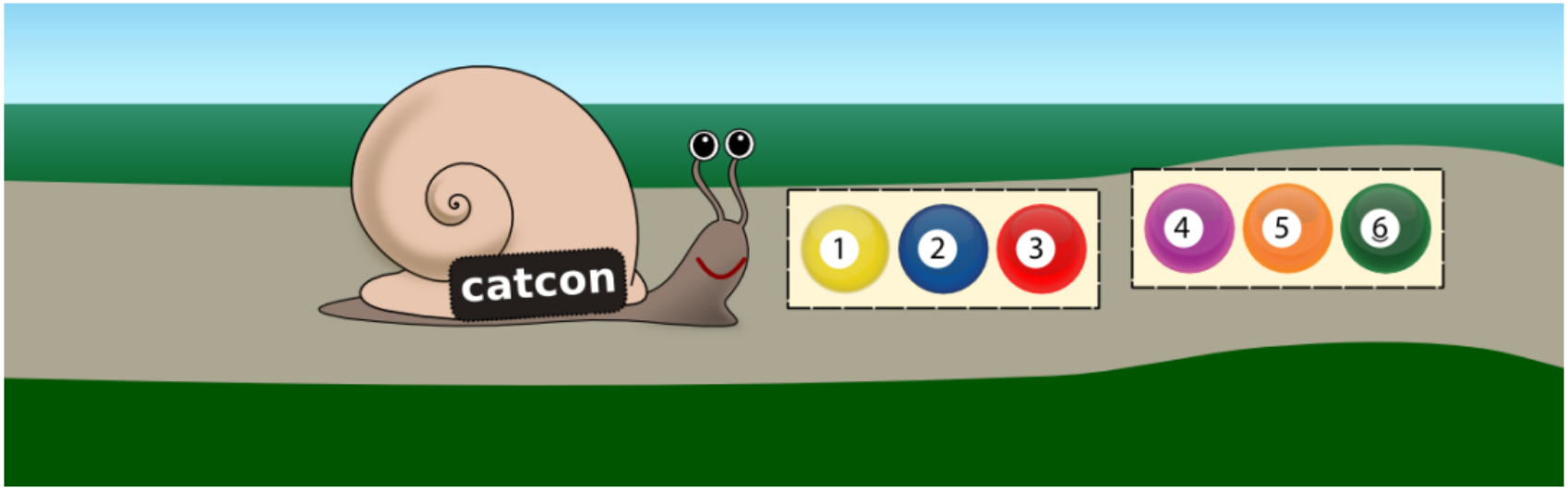
```haskell
list1 = [1,2,3]
list2 = [4,5,6]

list1 ++ list2
-- [1,2,3,4,5,6]

catcon list1 list2
-- [4,5,6,1,2,3]

catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs
```


f(g(★))
g(★)
f(■)

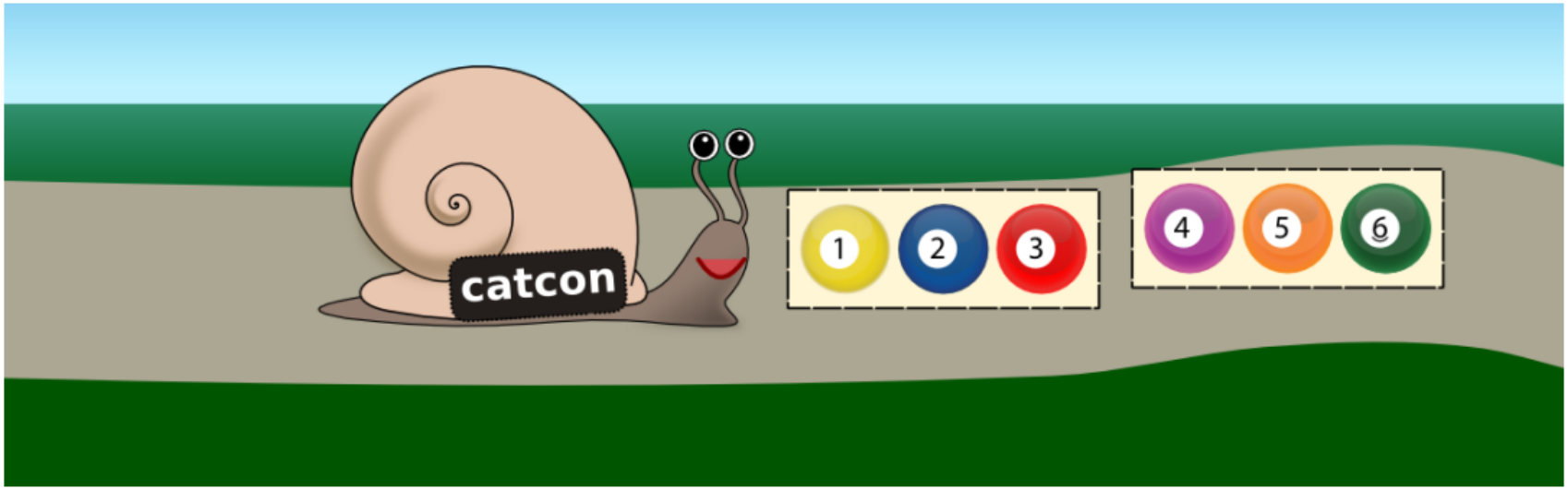
Friendly Funky Snail



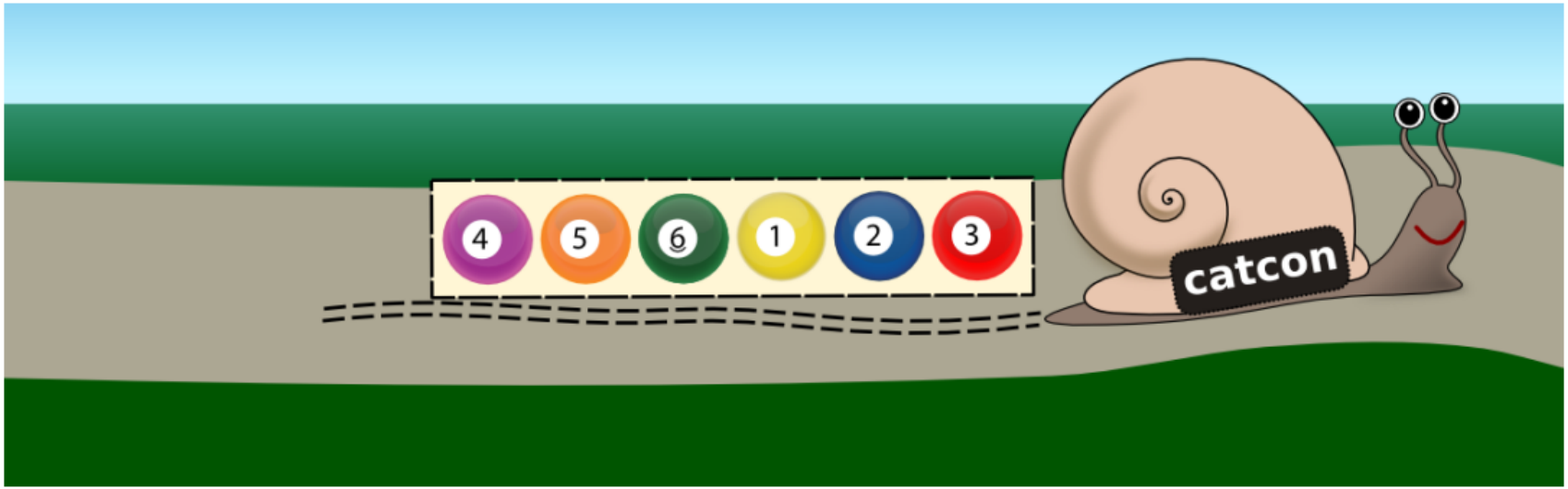
Friendly Funky Snail



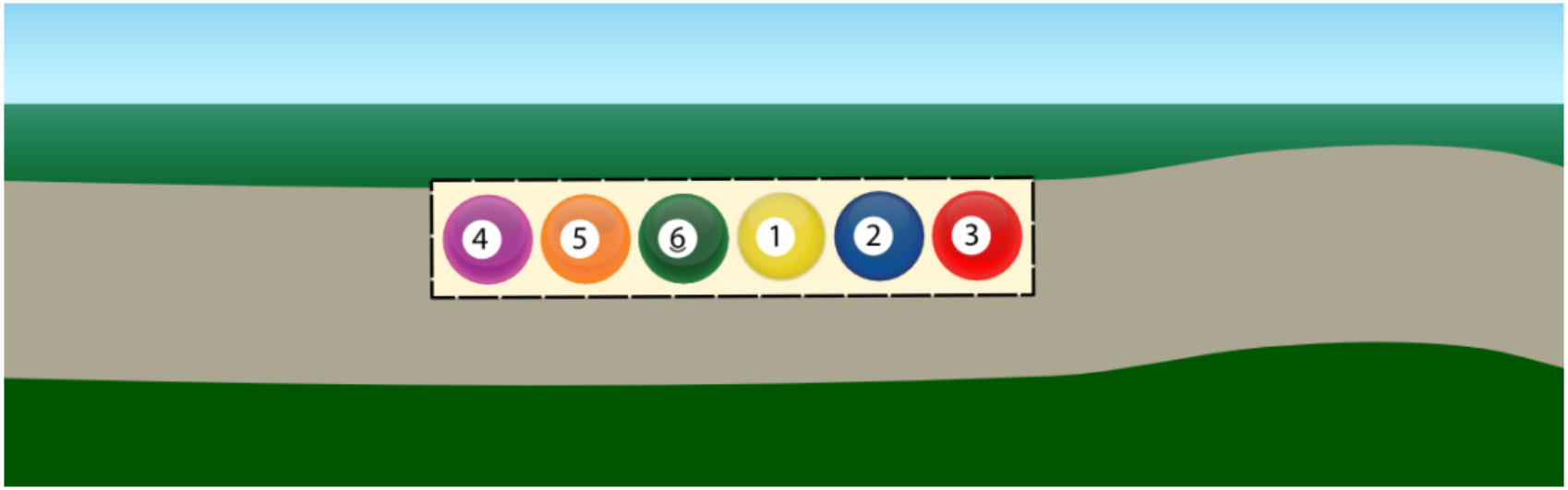
Friendly Funky Snail



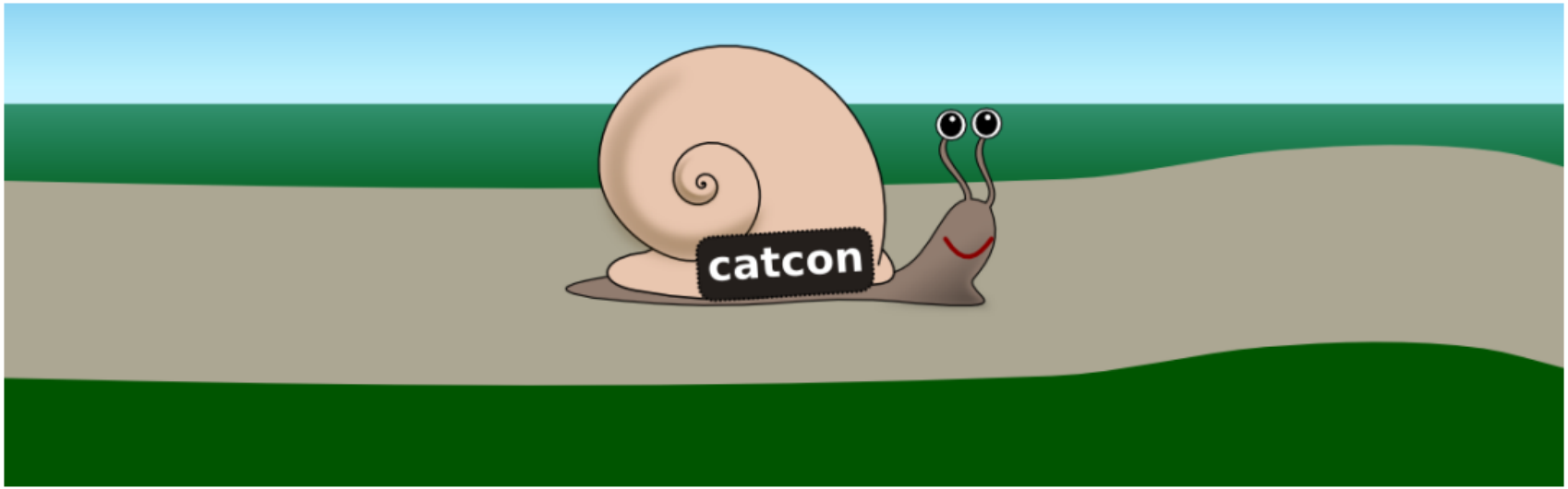
Friendly Funky Snail



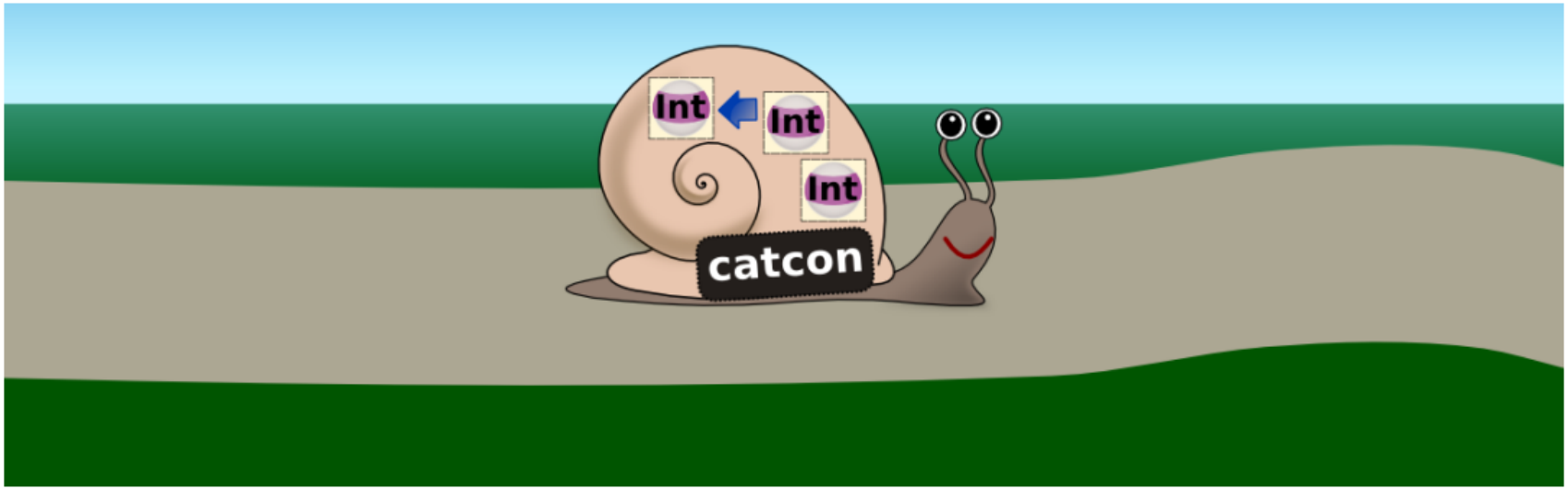
Friendly Funky Snail



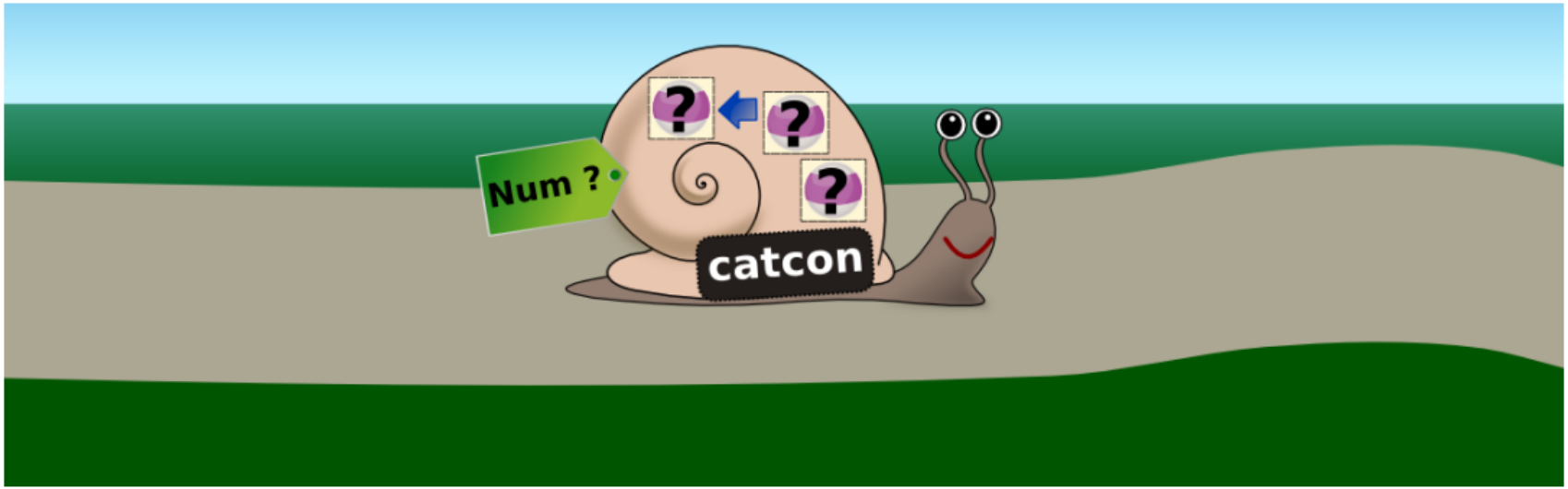
Friendly Funky Snail



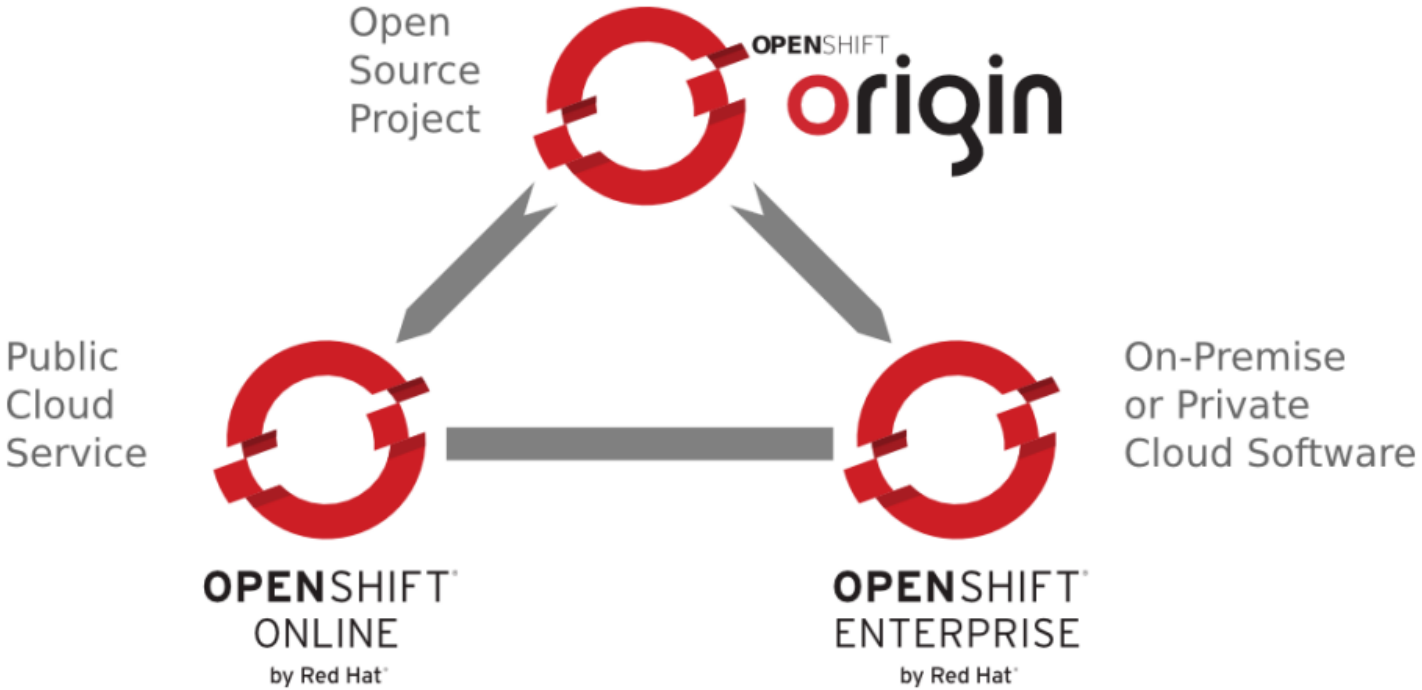
Friendly Funky Snail



Friendly Funky Snail



OpenShift


Open
Source
Project
OPENSHIFT
origin
Public
Cloud
Service
On-Premise
or Private
Cloud Software
OPENSHIFT
ONLINE
by Red Hat
OPENSHIFT
ENTERPRISE
by Red Hat

Wubble Demo App

Wubble

Wordset:

Functional Programming 101

Number of bubbles:

Submit

Referential Transparency

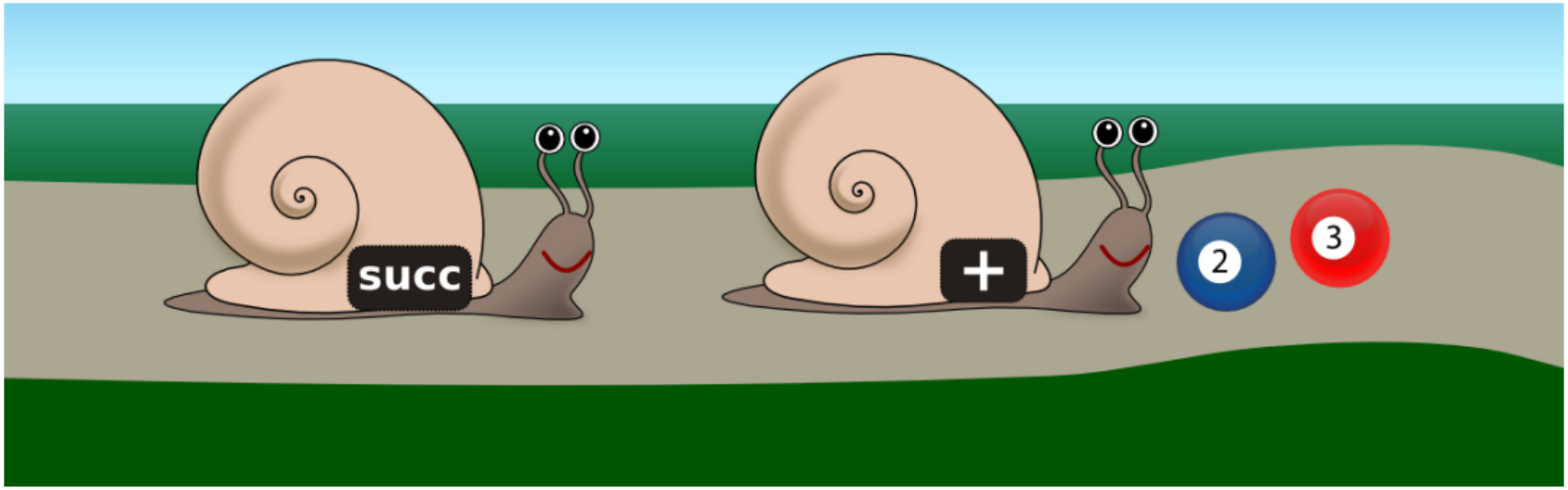
WHAT: A property of an expression in a program that can be replaced with its value without changing the behaviour of the program

WHY: We can reason about program behaviour

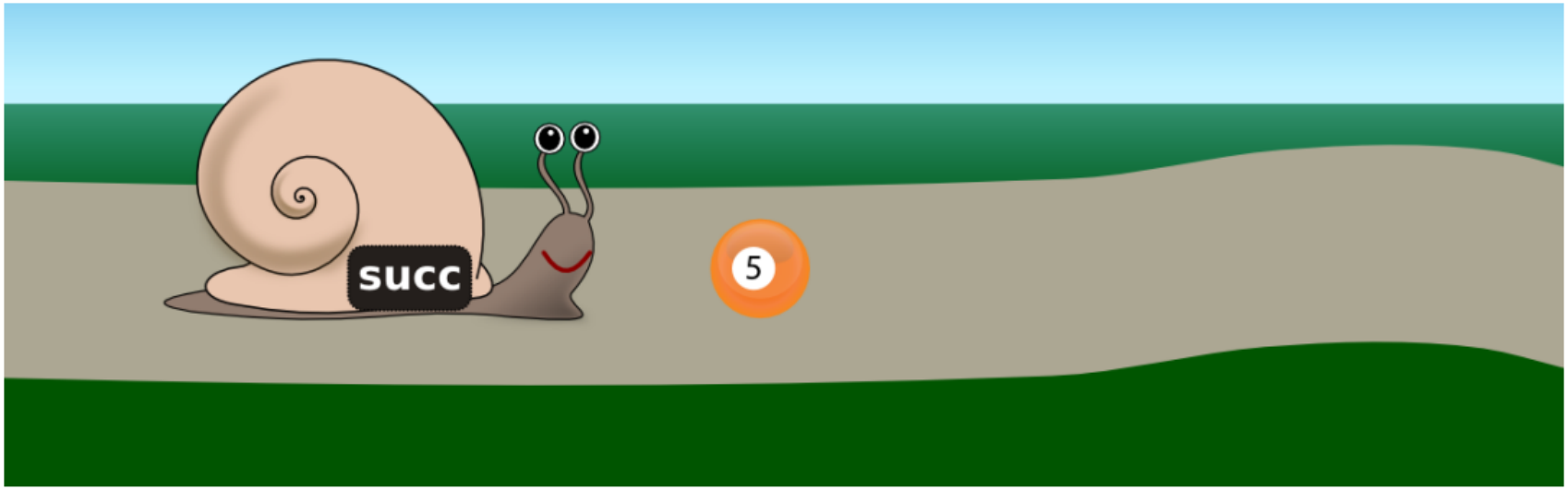
```
score = succ (2 + 3)
main = putStrLn (show score)
```

```
score = succ (2 + 3)
main = putStrLn (show score)

$ ./main
6
```

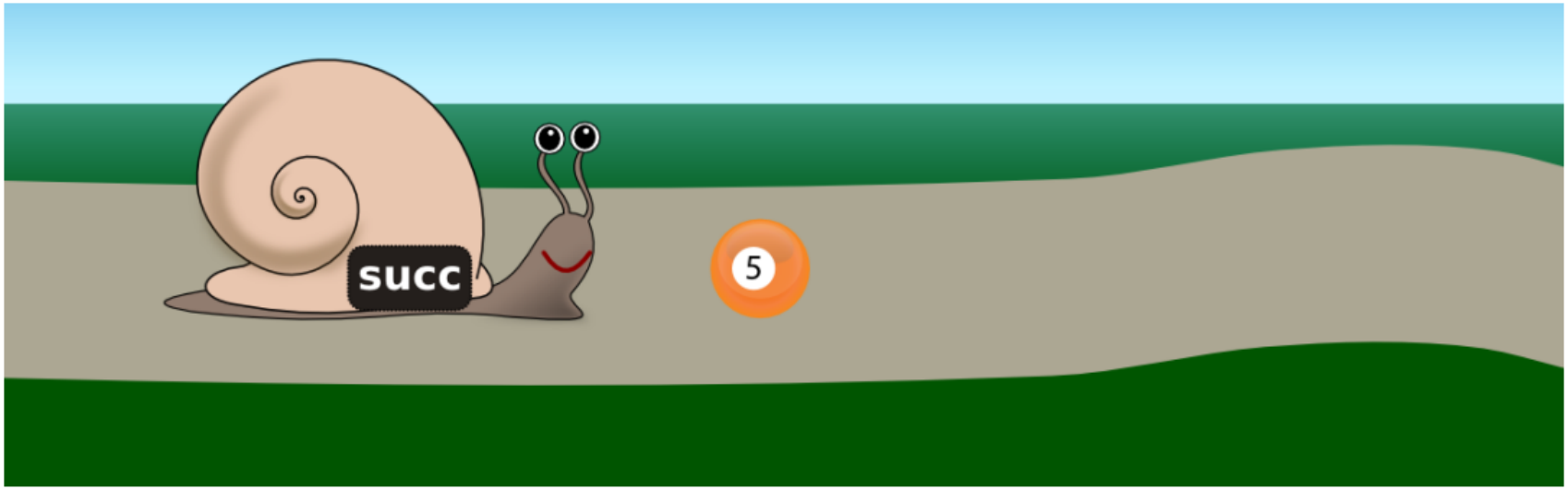
```
score = succ (2 + 3)
main = putStrLn (show score)

$ ./main
6
```



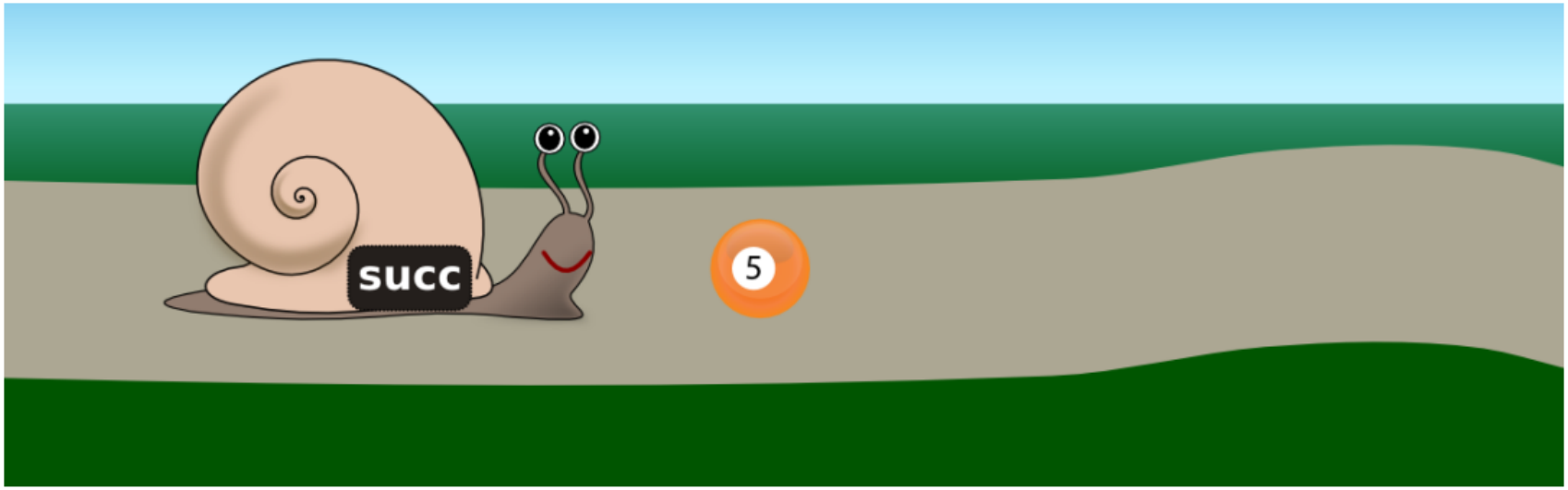
```
score = succ 5
main = putStrLn (show score)

$ ./main
6
```



```
score = succ 5
main = putStrLn (show score)

$ ./main
6
```



```
score = succ 5
main = putStrLn (show (succ 5))

$ ./main
6
```



```
staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"
```

```
staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"
```

```
staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"
```

Higher-Order Function

WHAT: A function that takes a function as an argument or returns one as a result

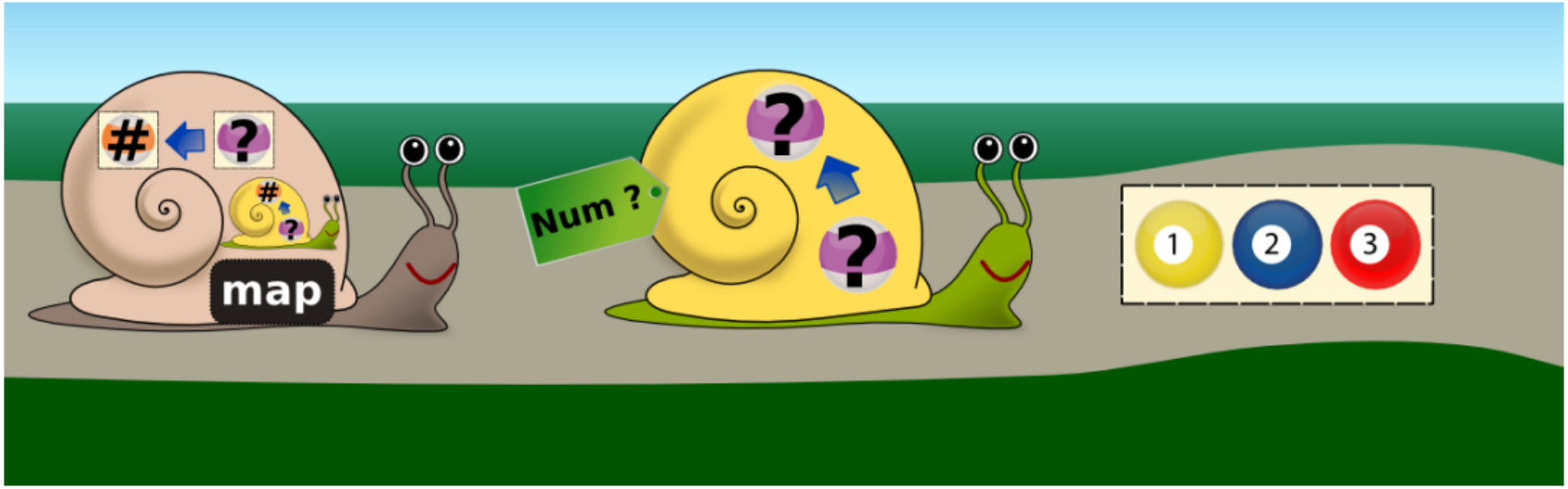
WHY: Glue pieces of code together; reduce code repetition; modular design

```
ghci> let list1 = [1,2,3]
```

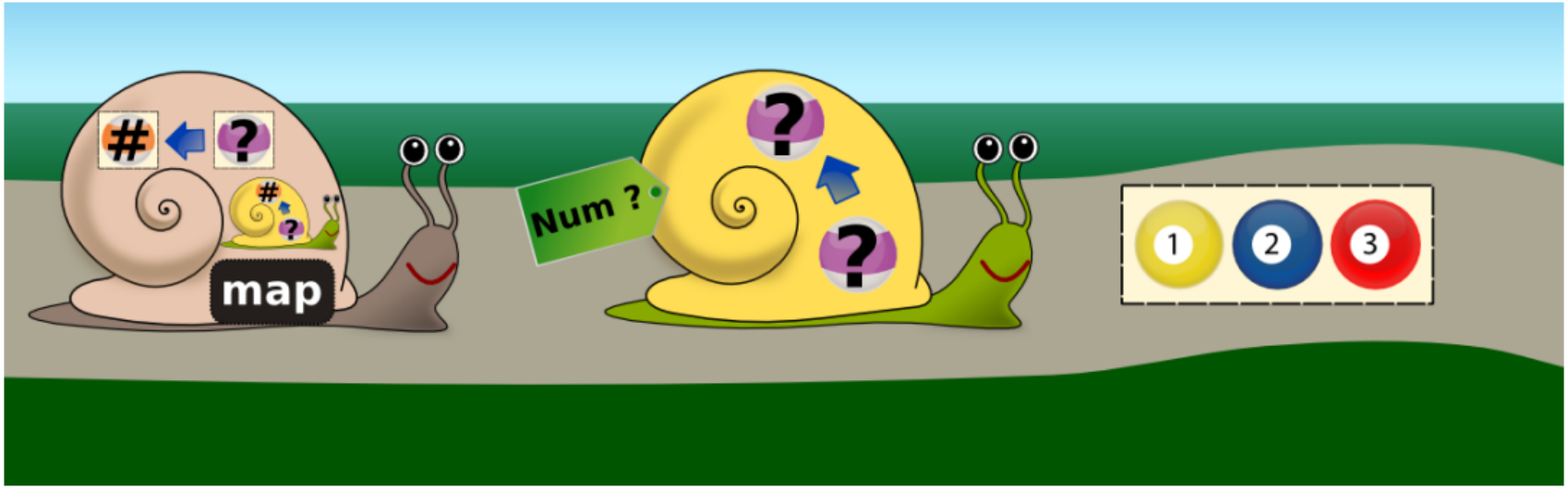
```
ghci> let list1 = [1,2,3]

ghci> :type map
```

```
ghci> let list1 = [1,2,3]

ghci> :type map
map :: (a -> b) -> [a] -> [b]
```

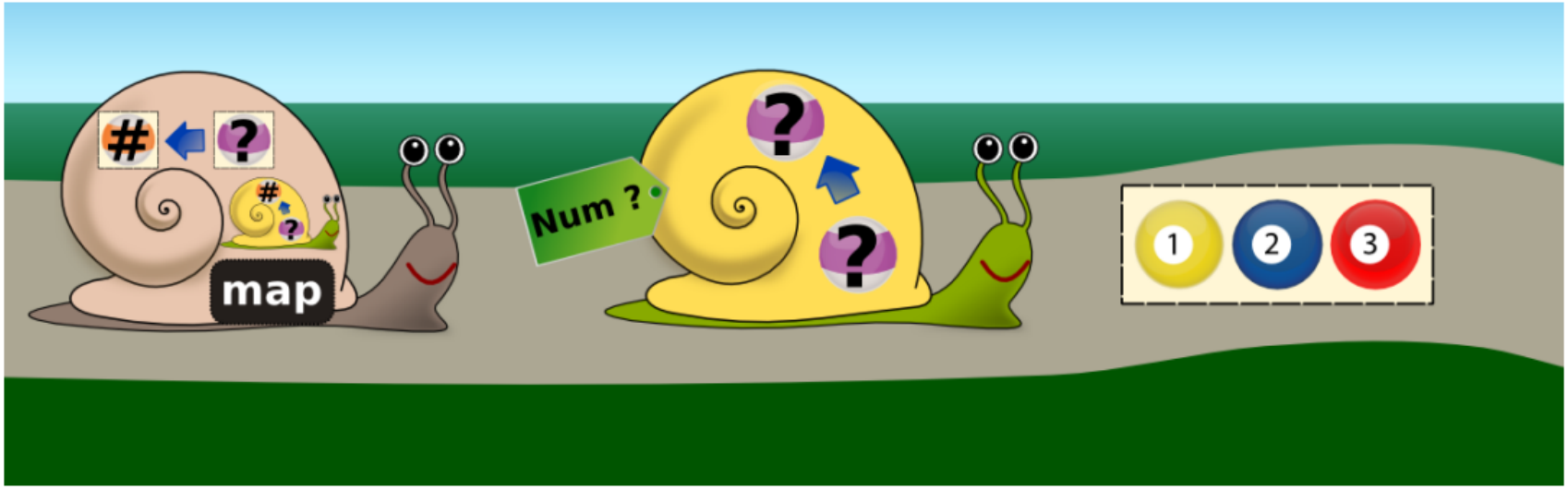
```
ghci> let list1 = [1,2,3]

ghci> :type map
map :: (a -> b) -> [a] -> [b]

ghci> map (\elem -> elem * 2) list1
```



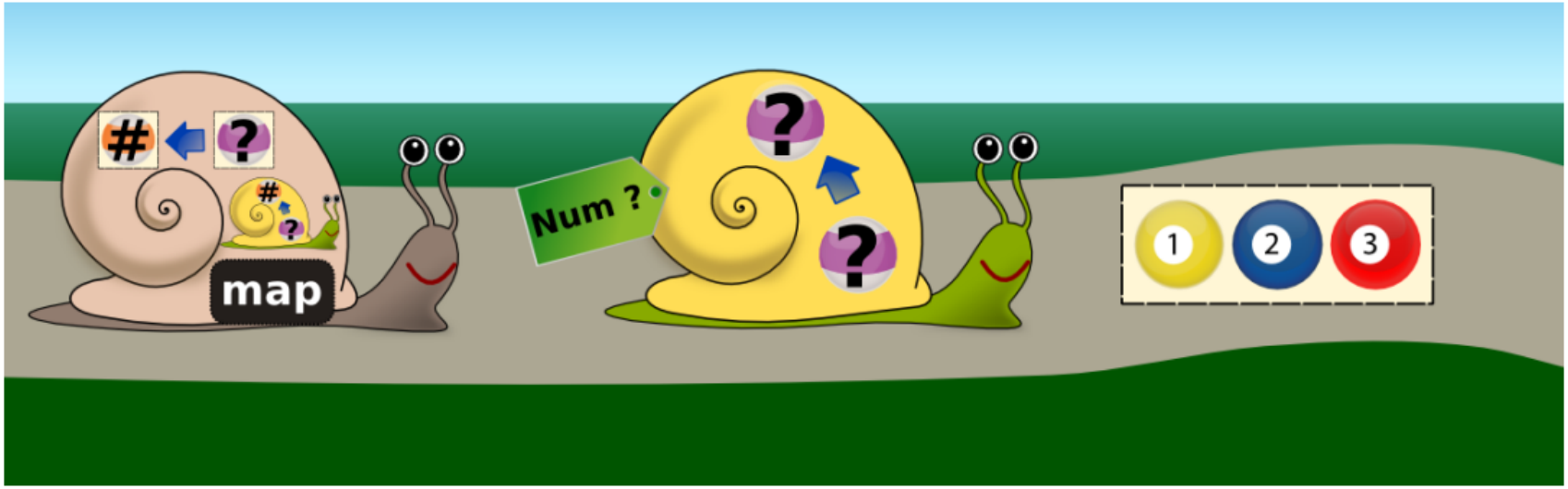
```
ghci> let list1 = [1,2,3]

ghci> :type map
map :: (a -> b) -> [a] -> [b]

ghci> map (\elem -> elem * 2) list1
```



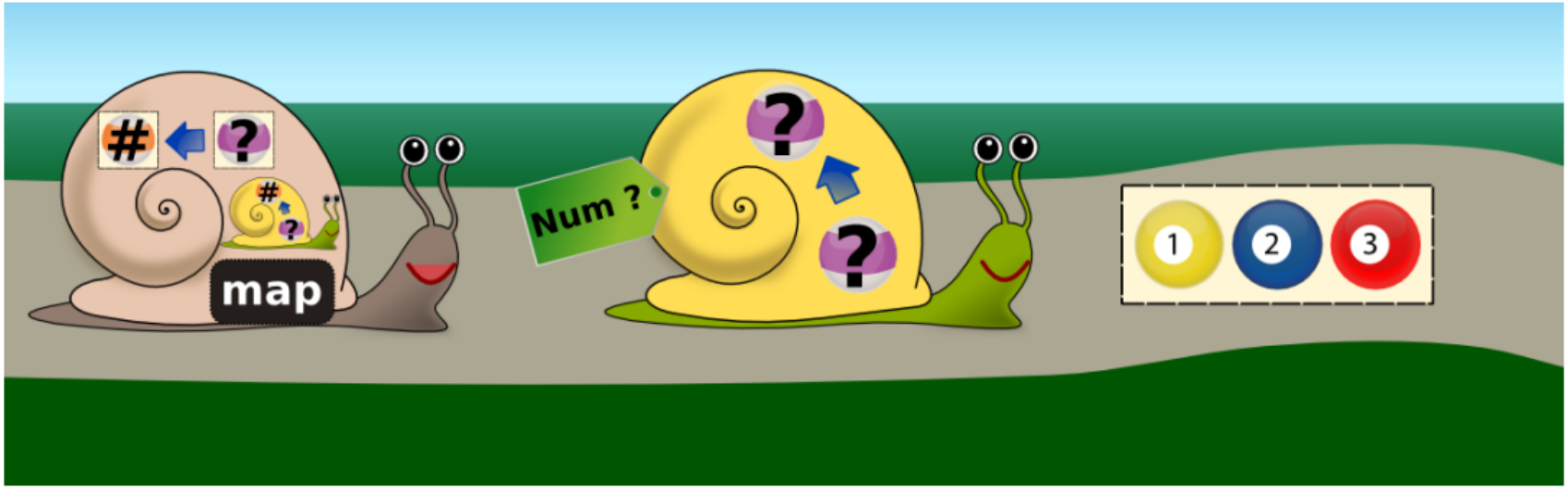
```
ghci> let list1 = [1,2,3]

ghci> :type map
map :: (a -> b) -> [a] -> [b]

ghci> map (\elem -> elem * 2) list1
```



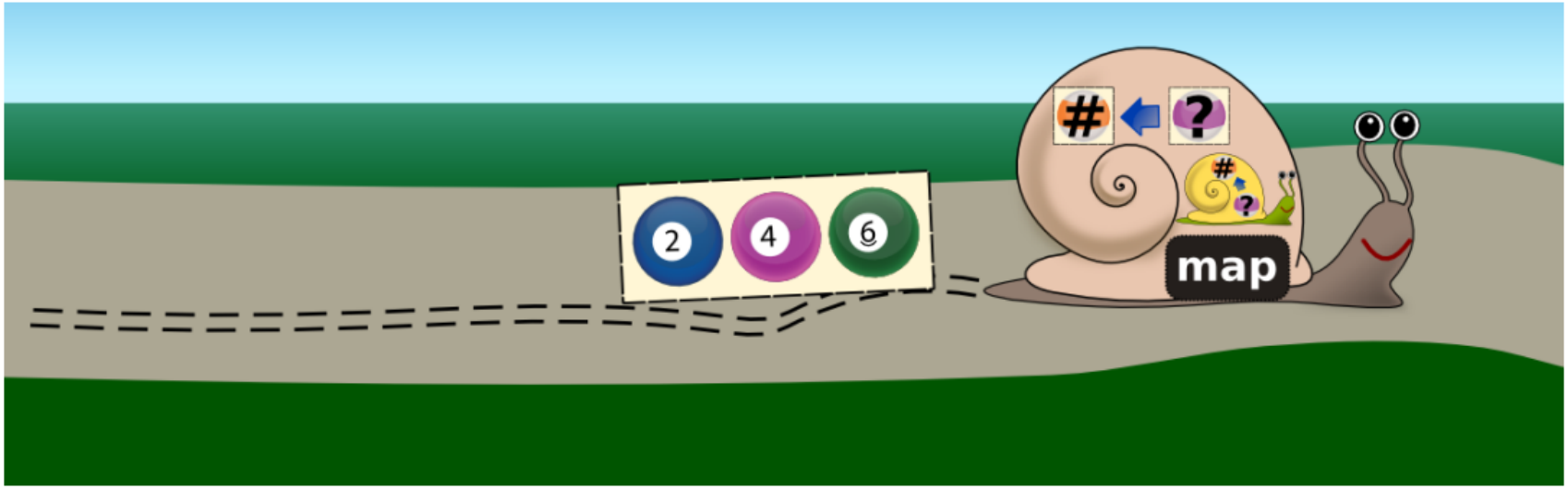
```
ghci> let list1 = [1,2,3]

ghci> :type map
map :: (a -> b) -> [a] -> [b]

ghci> map (\elem -> elem * 2) list1
```



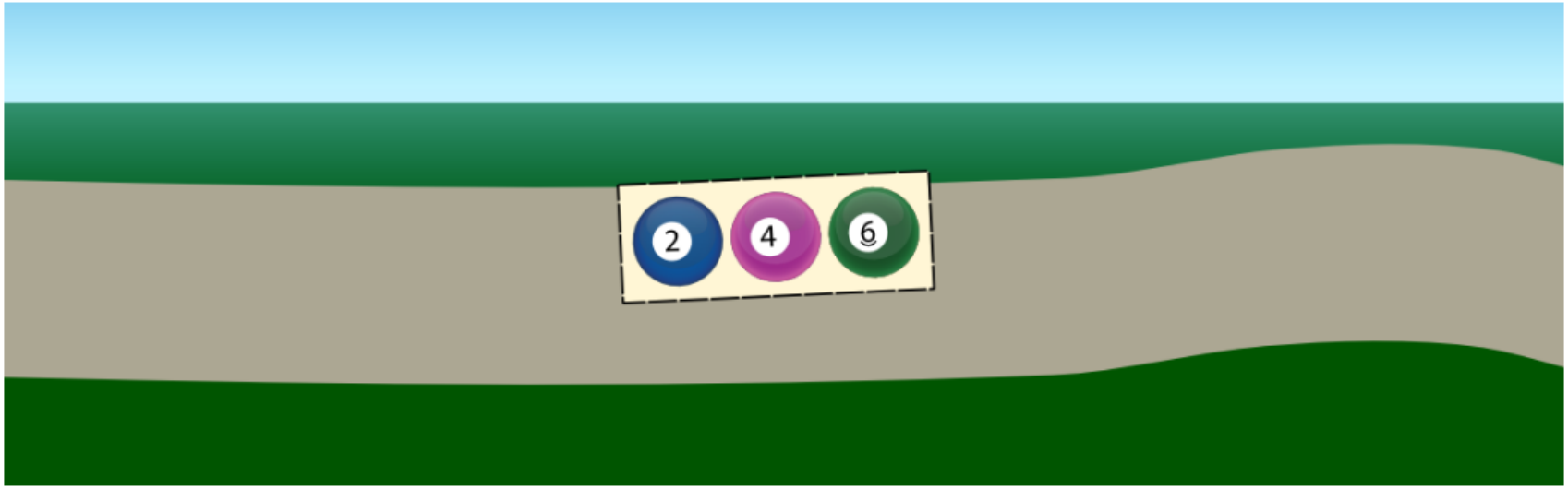
```
ghci> let list1 = [1,2,3]

ghci> :type map
map :: (a -> b) -> [a] -> [b]

ghci> map (\elem -> elem * 2) list1
```



```
ghci> let list1 = [1,2,3]

ghci> :type map
map :: (a -> b) -> [a] -> [b]

ghci> map (\elem -> elem * 2) list1
[2,4,6]
```

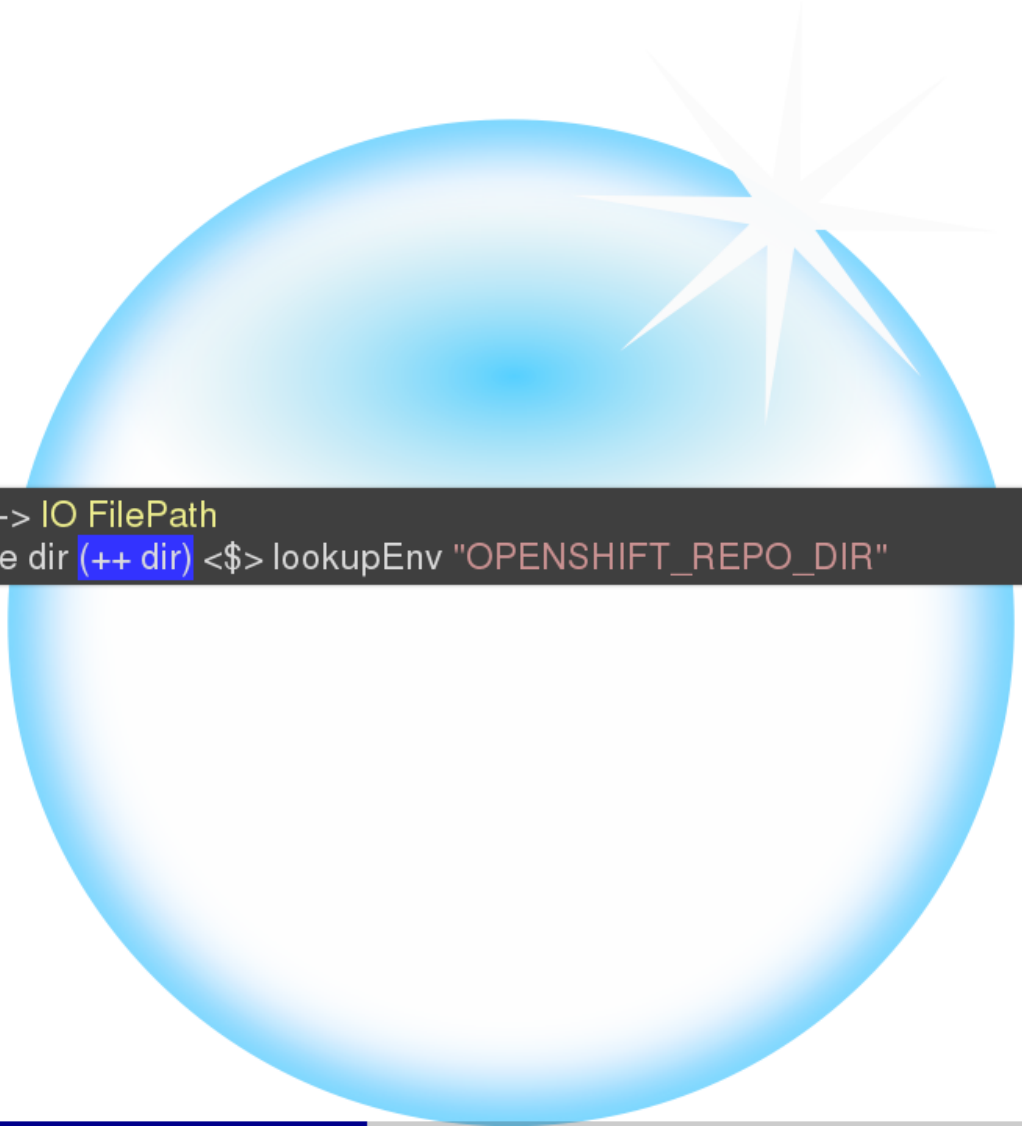


```
ghci> let list1 = [1,2,3]

ghci> :type map
map :: (a -> b) -> [a] -> [b]

ghci> map (\elem -> elem * 2) list1
[2,4,6]
```



```
staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"
```

```
staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"
```



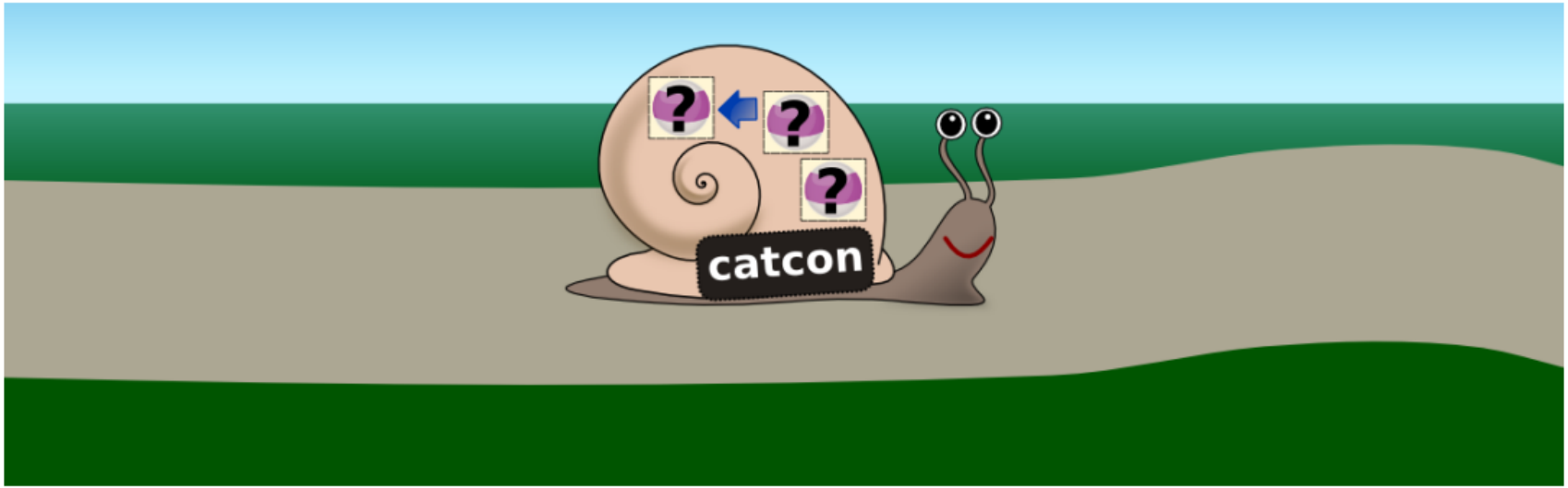
```
staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"
```

Currying

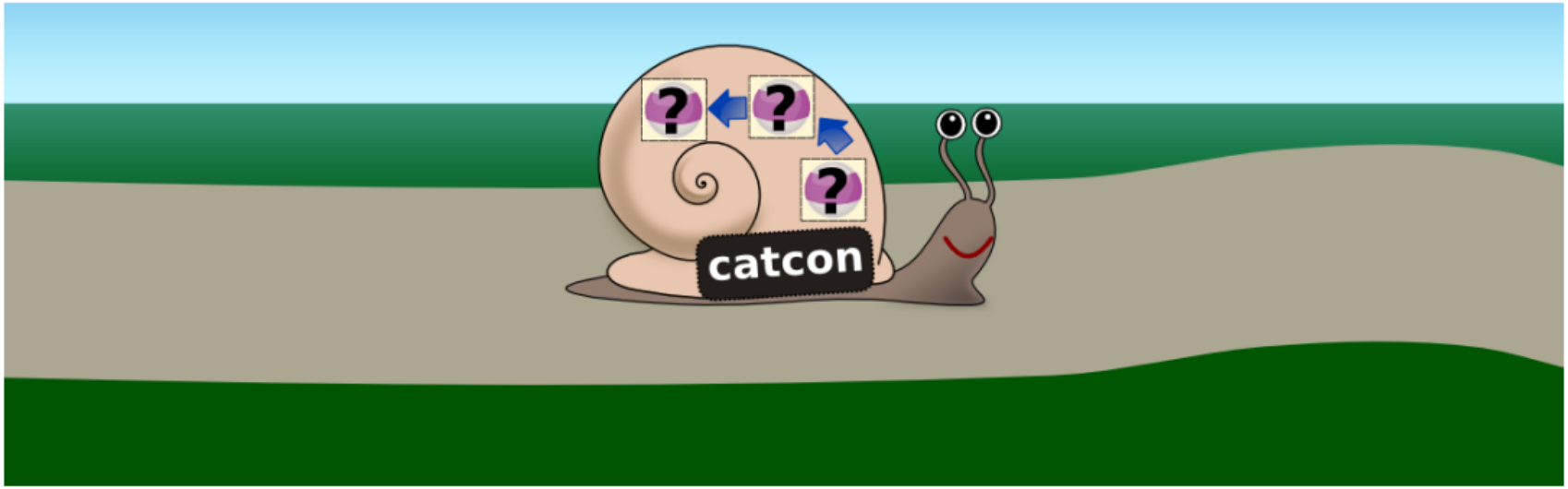
WHAT: Evaluating multi-argument functions as a chain of functions that each take exactly one argument

WHY: Aids code reuse; can make code more readable

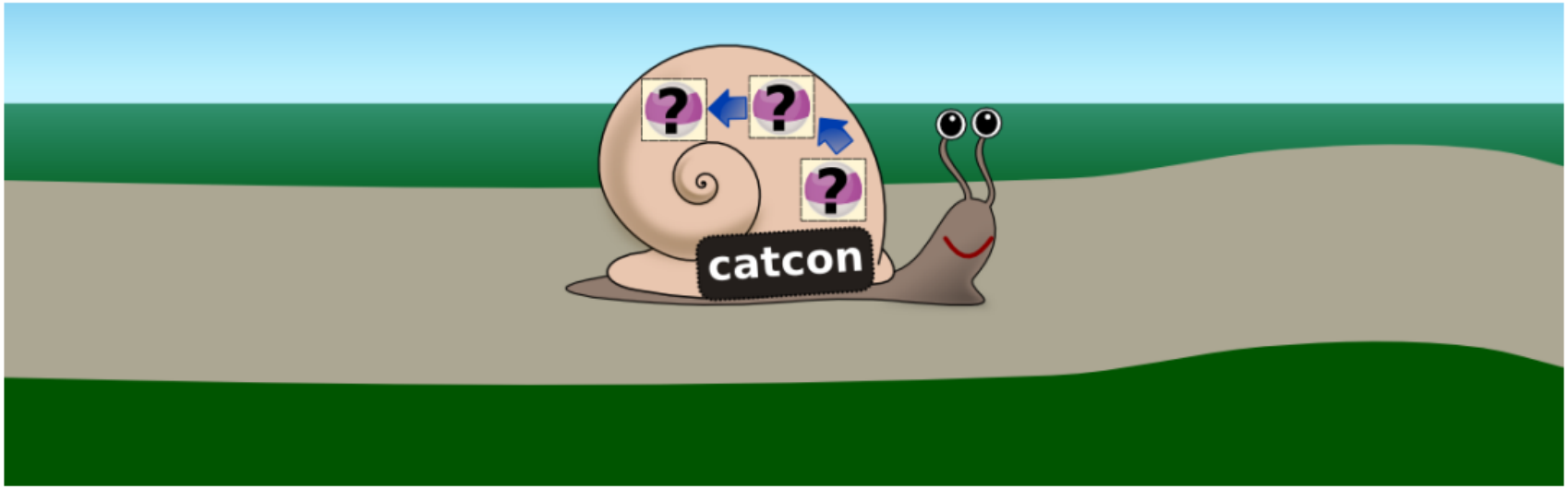
```haskell
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs
```



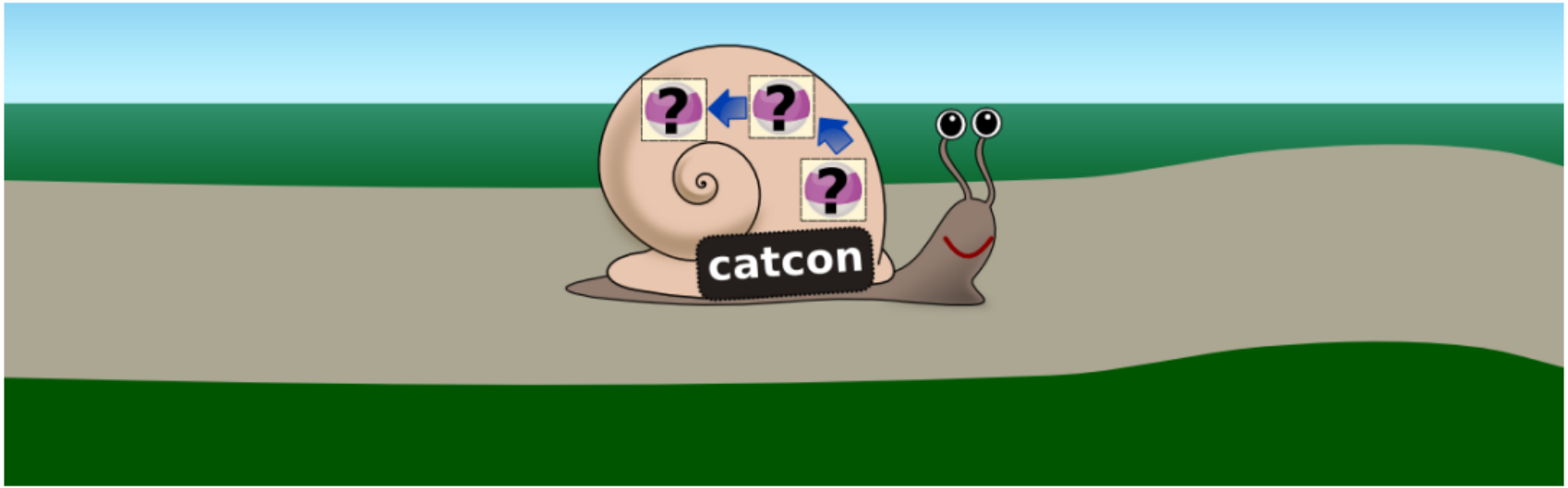
```haskell
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs
```



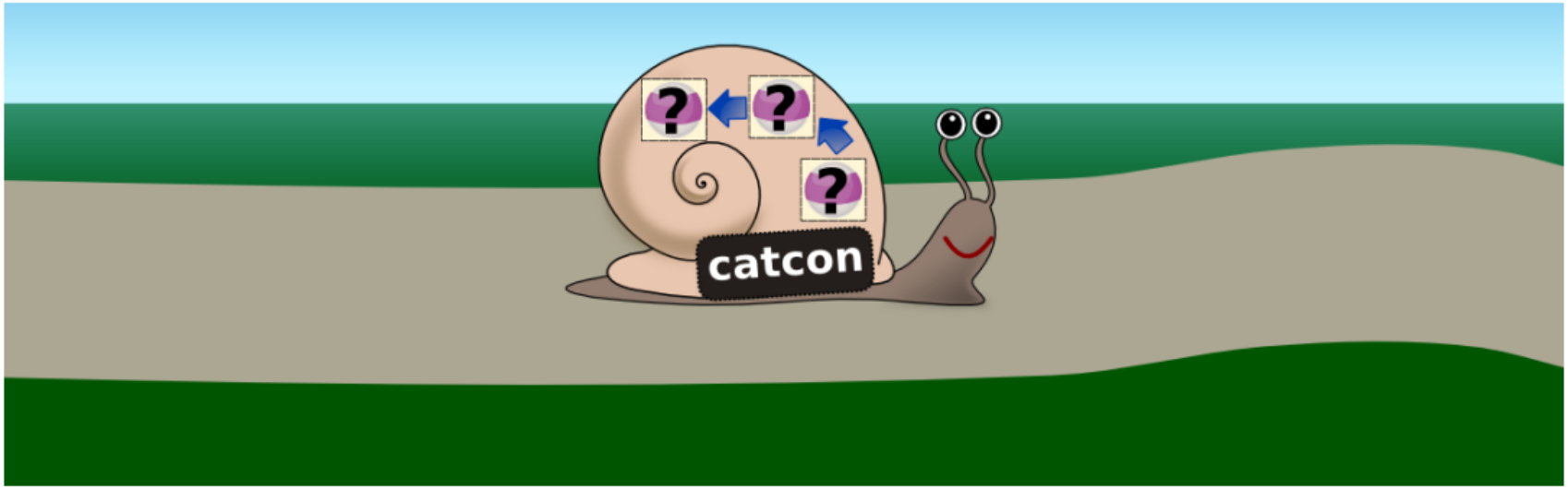
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs
```



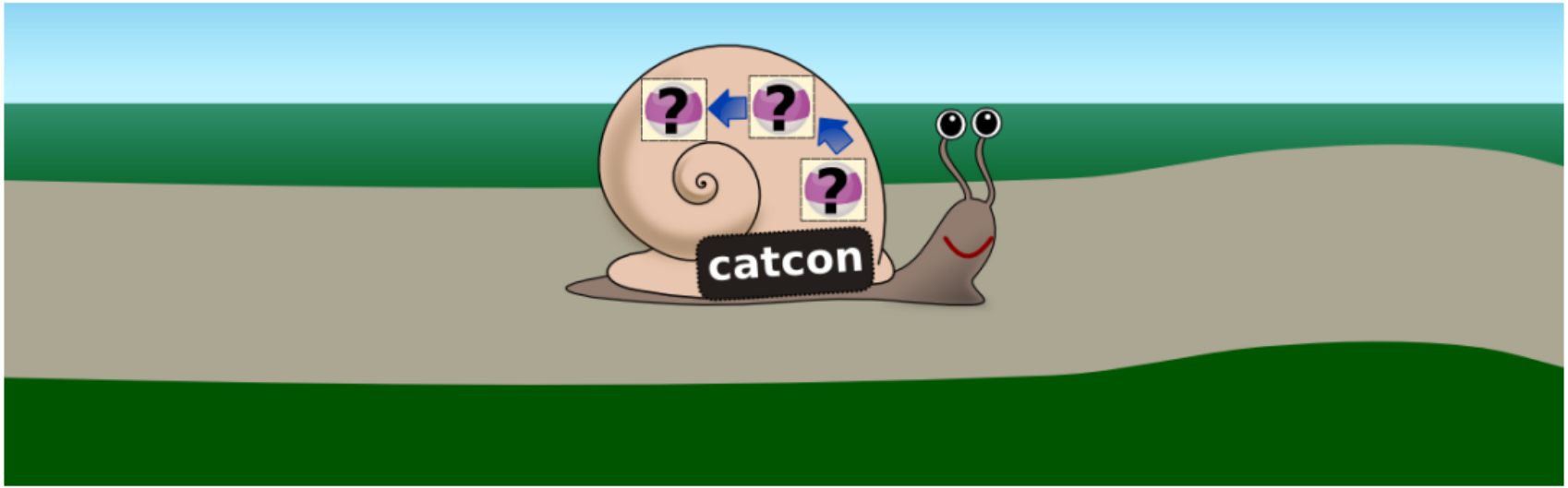
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs
```



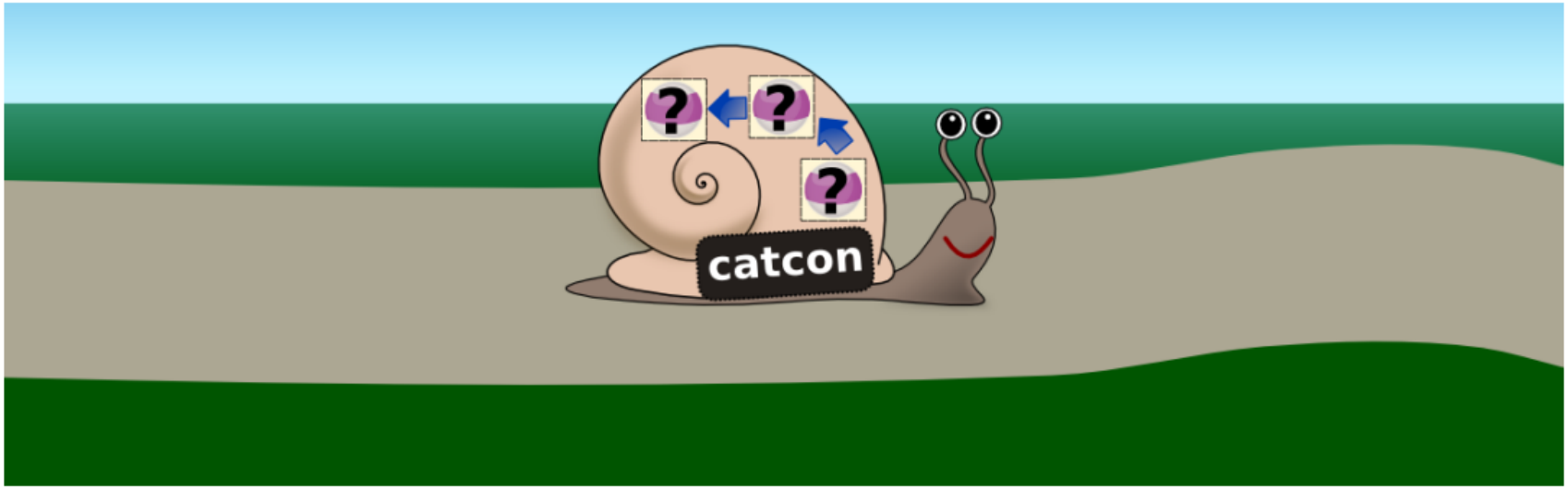
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs

ghci> :type catcon [1,2,3]
```



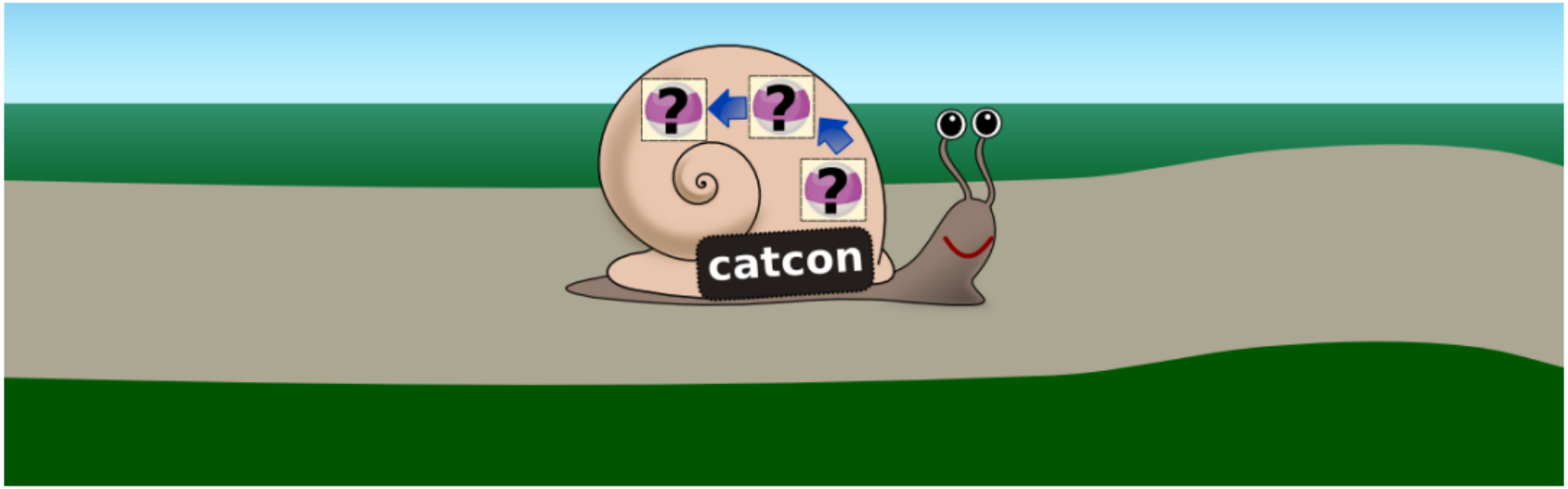
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs

ghci> :type catcon [1,2,3]
catcon [1,2,3] :: Num a => [a] -> [a]
```



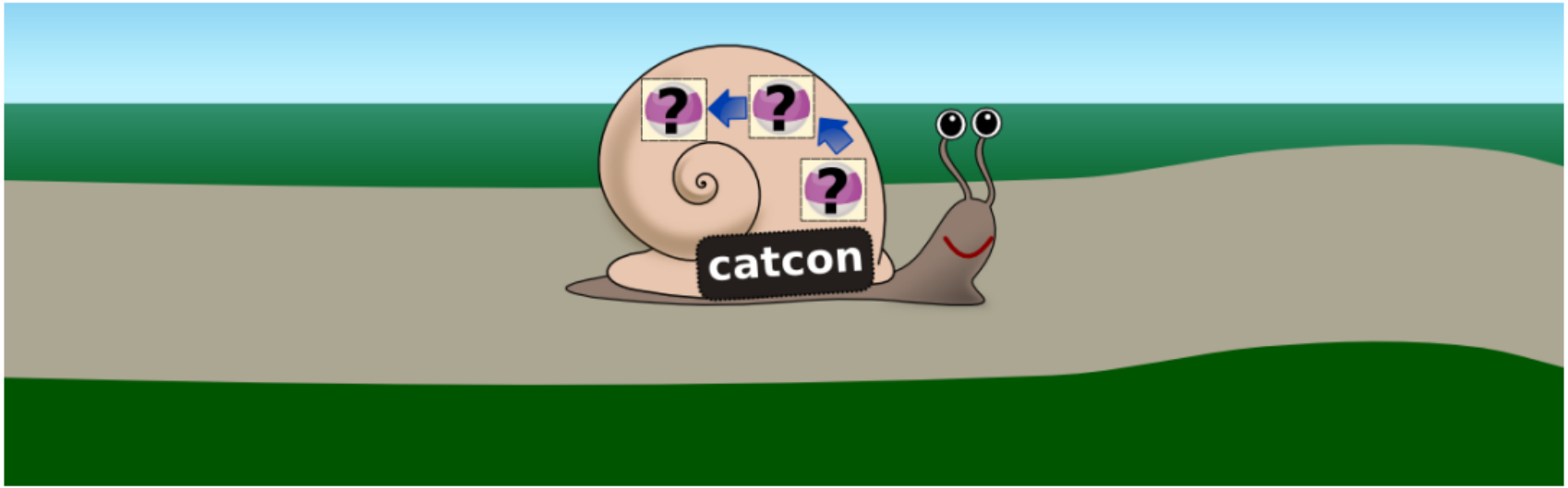
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs

ghci> :type catcon [1,2,3]
catcon [1,2,3] :: Num a => [a] -> [a]
```



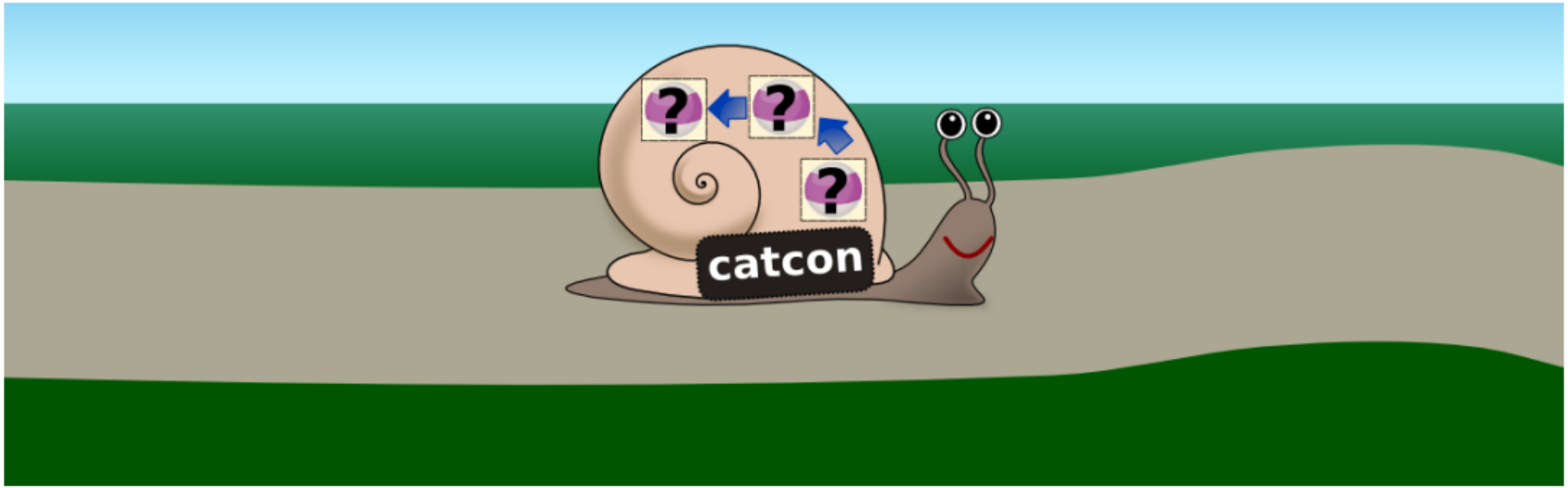
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs

ghci> :type catcon [1,2,3]
catcon [1,2,3] :: Num a => [a] -> [a]
ghci> :type map (catcon [1,2,3])
```



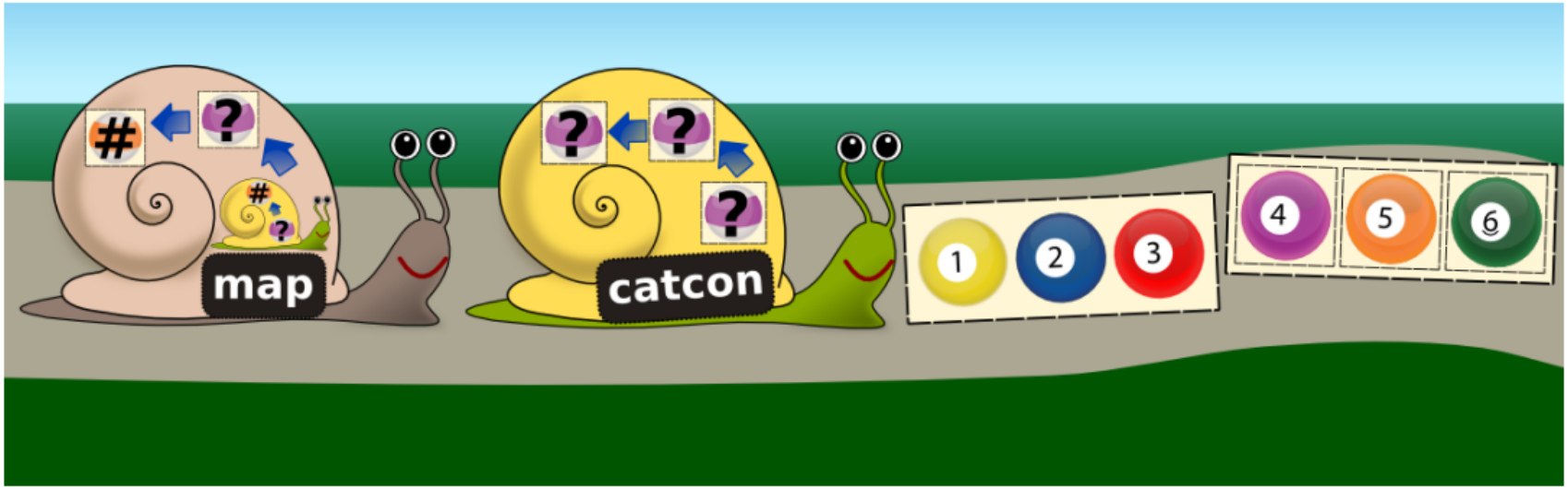
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs

ghci> :type catcon [1,2,3]
catcon [1,2,3] :: Num a => [a] -> [a]
ghci> :type map (catcon [1,2,3])
map (catcon [1,2,3]) :: Num a => [[a]] -> [[a]]
```



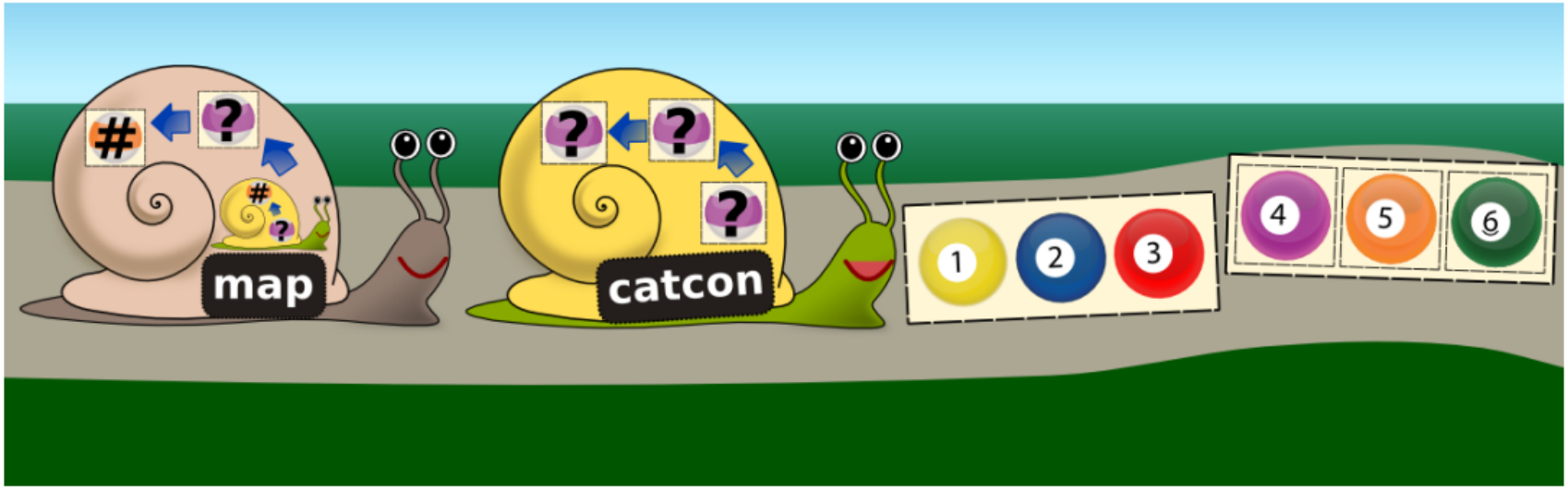
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs

ghci> :type catcon [1,2,3]
catcon [1,2,3] :: Num a => [a] -> [a]
ghci> :type map (catcon [1,2,3])
map (catcon [1,2,3]) :: Num a => [[a]] -> [[a]]
```



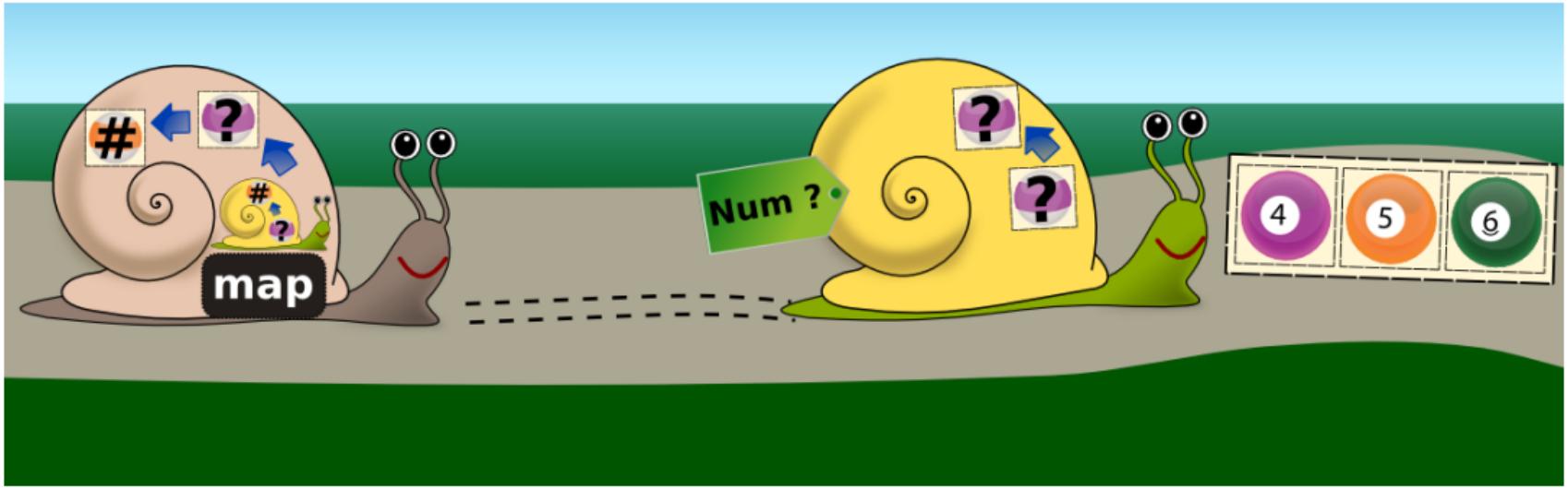
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs

ghci> :type catcon [1,2,3]
catcon [1,2,3] :: Num a => [a] -> [a]
ghci> :type map (catcon [1,2,3])
map (catcon [1,2,3]) :: Num a => [[a]] -> [[a]]
ghci> map (catcon [1,2,3]) [[4],[5],[6]]
```



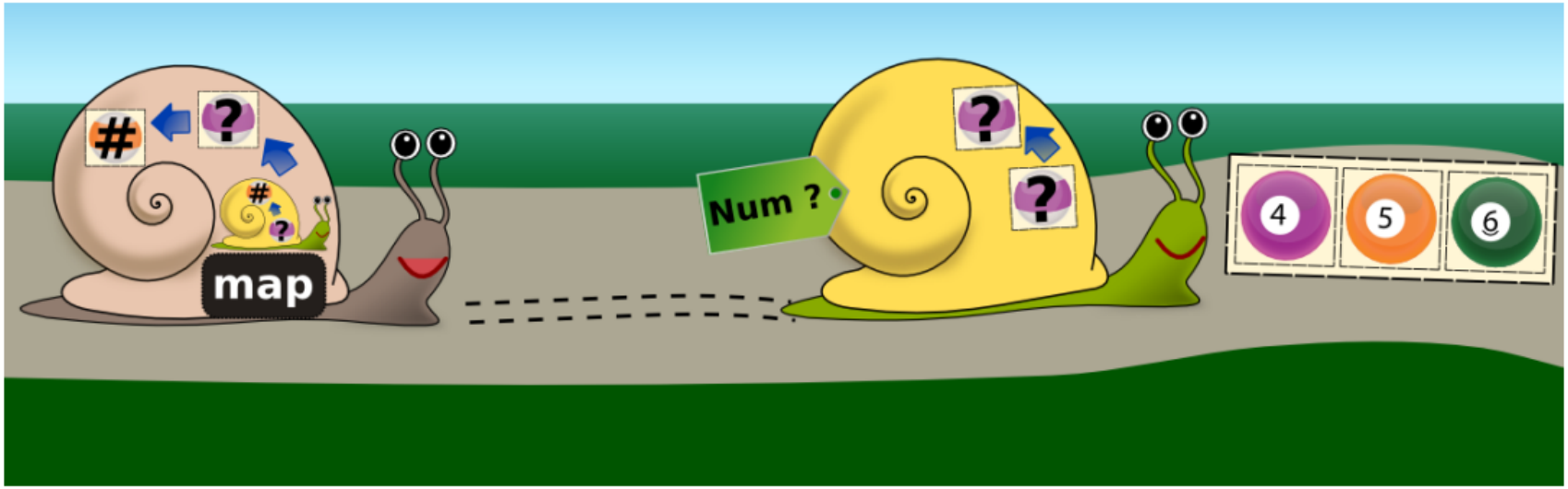
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs

ghci> :type catcon [1,2,3]
catcon [1,2,3] :: Num a => [a] -> [a]
ghci> :type map (catcon [1,2,3])
map (catcon [1,2,3]) :: Num a => [[a]] -> [[a]]
ghci> map (catcon [1,2,3]) [[4],[5],[6]]
```



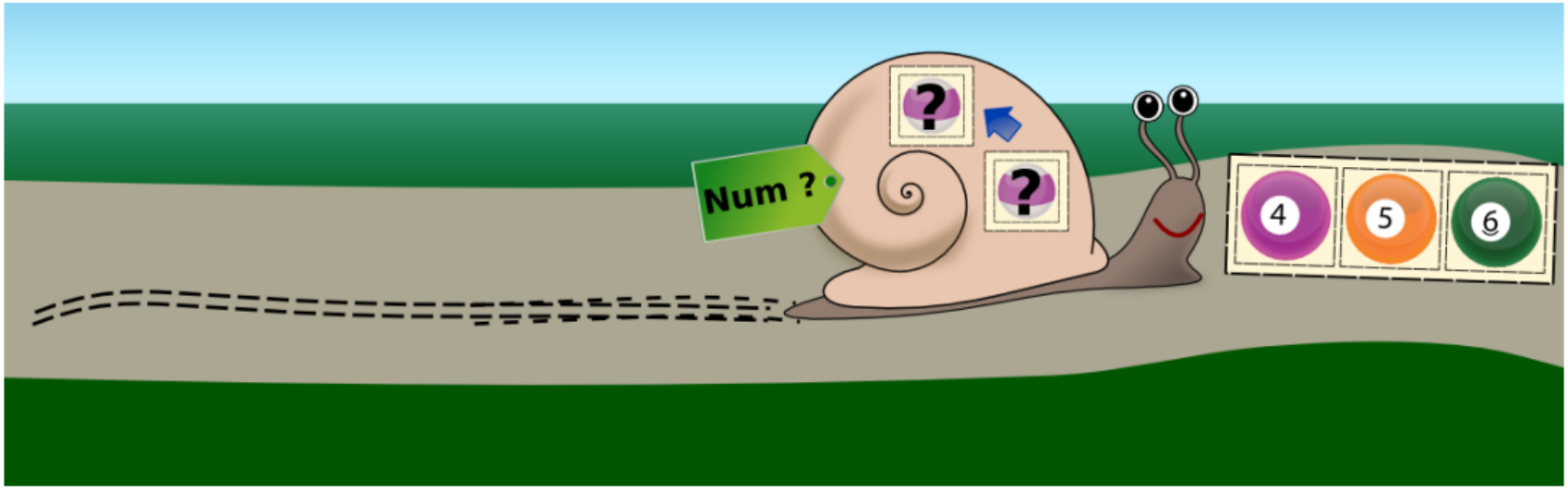
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs

ghci> :type catcon [1,2,3]
catcon [1,2,3] :: Num a => [a] -> [a]
ghci> :type map (catcon [1,2,3])
map (catcon [1,2,3]) :: Num a => [[a]] -> [[a]]
ghci> map (\list -> catcon [1,2,3] list) [[4],[5],[6]]
```



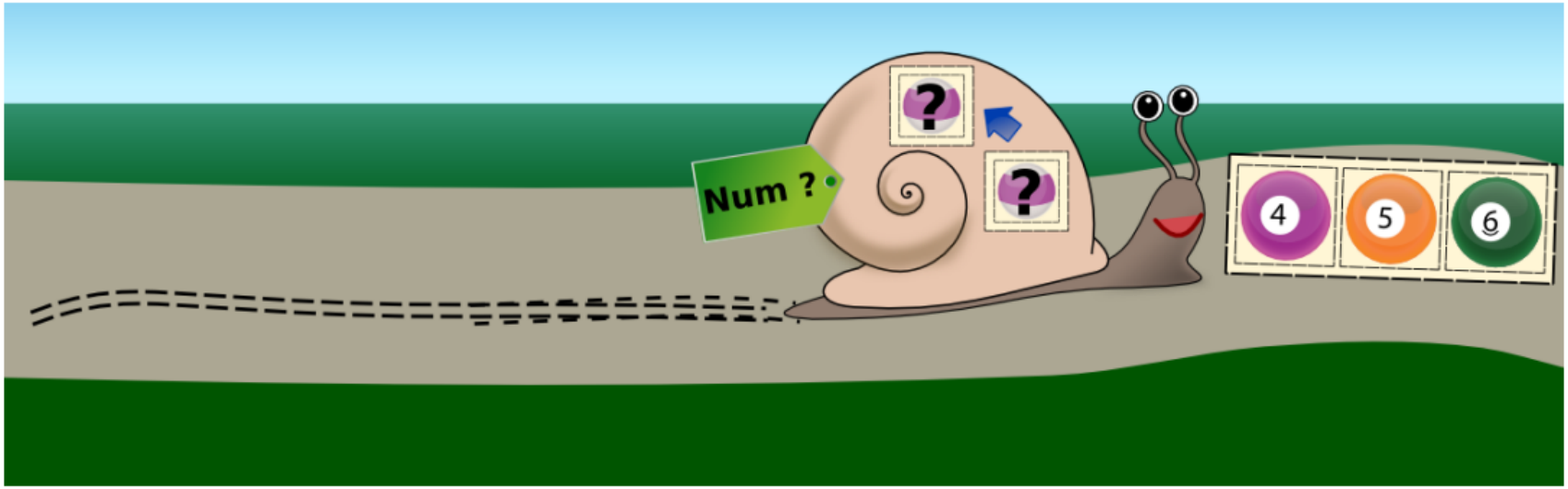
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs

ghci> :type catcon [1,2,3]
catcon [1,2,3] :: Num a => [a] -> [a]
ghci> :type map (catcon [1,2,3])
map (catcon [1,2,3]) :: Num a => [[a]] -> [[a]]
ghci> map (\list -> catcon [1,2,3] list) [[4],[5],[6]]
```



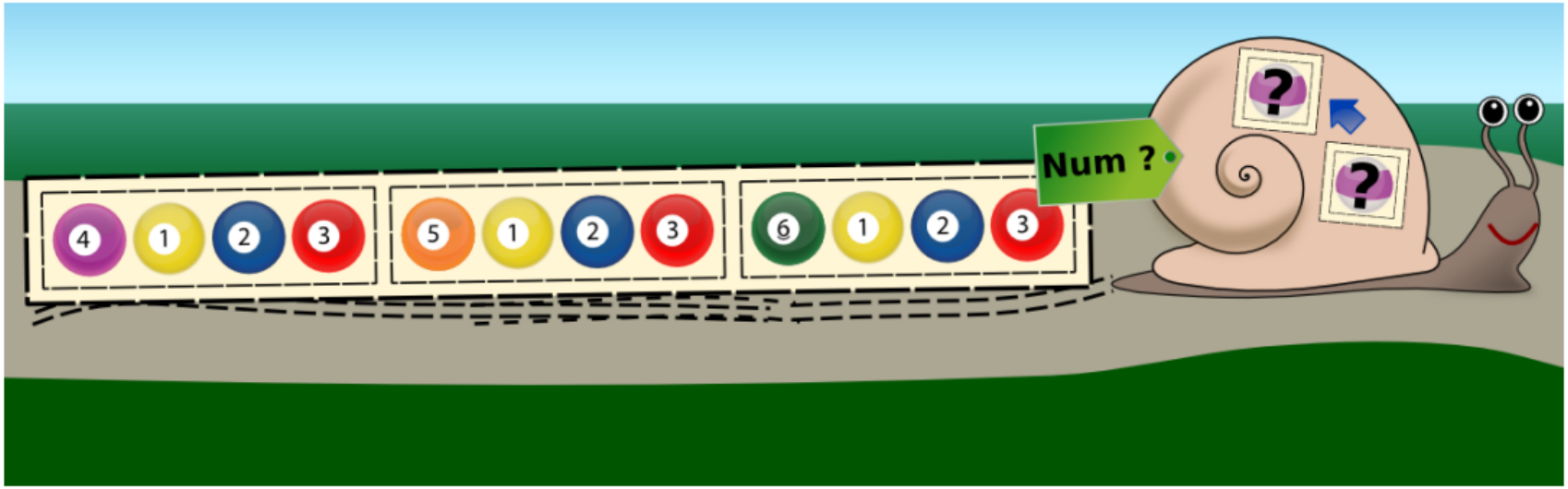
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs

ghci> :type catcon [1,2,3]
catcon [1,2,3] :: Num a => [a] -> [a]
ghci> :type map (catcon [1,2,3])
map (catcon [1,2,3]) :: Num a => [[a]] -> [[a]]
ghci> (\lists -> map (\list -> catcon [1,2,3] list) lists) [[4],[5],[6]]
```



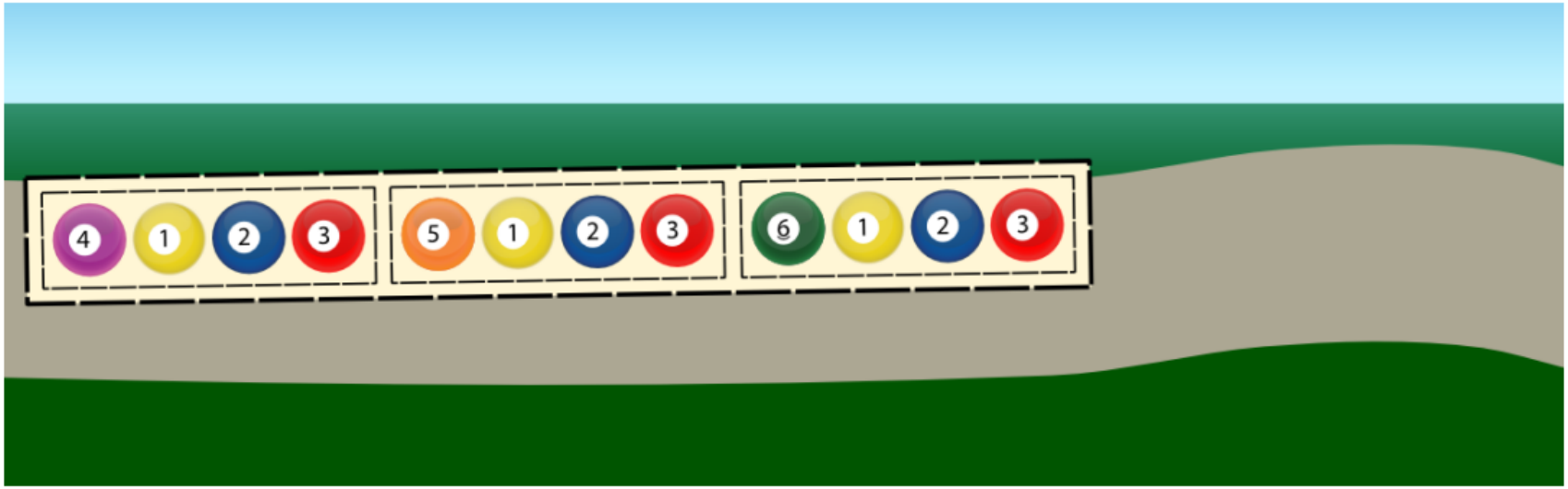
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs

ghci> :type catcon [1,2,3]
catcon [1,2,3] :: Num a => [a] -> [a]
ghci> :type map (catcon [1,2,3])
map (catcon [1,2,3]) :: Num a => [[a]] -> [[a]]
ghci> (\lists -> map (\list -> catcon [1,2,3] list) lists) [[4],[5],[6]]
```



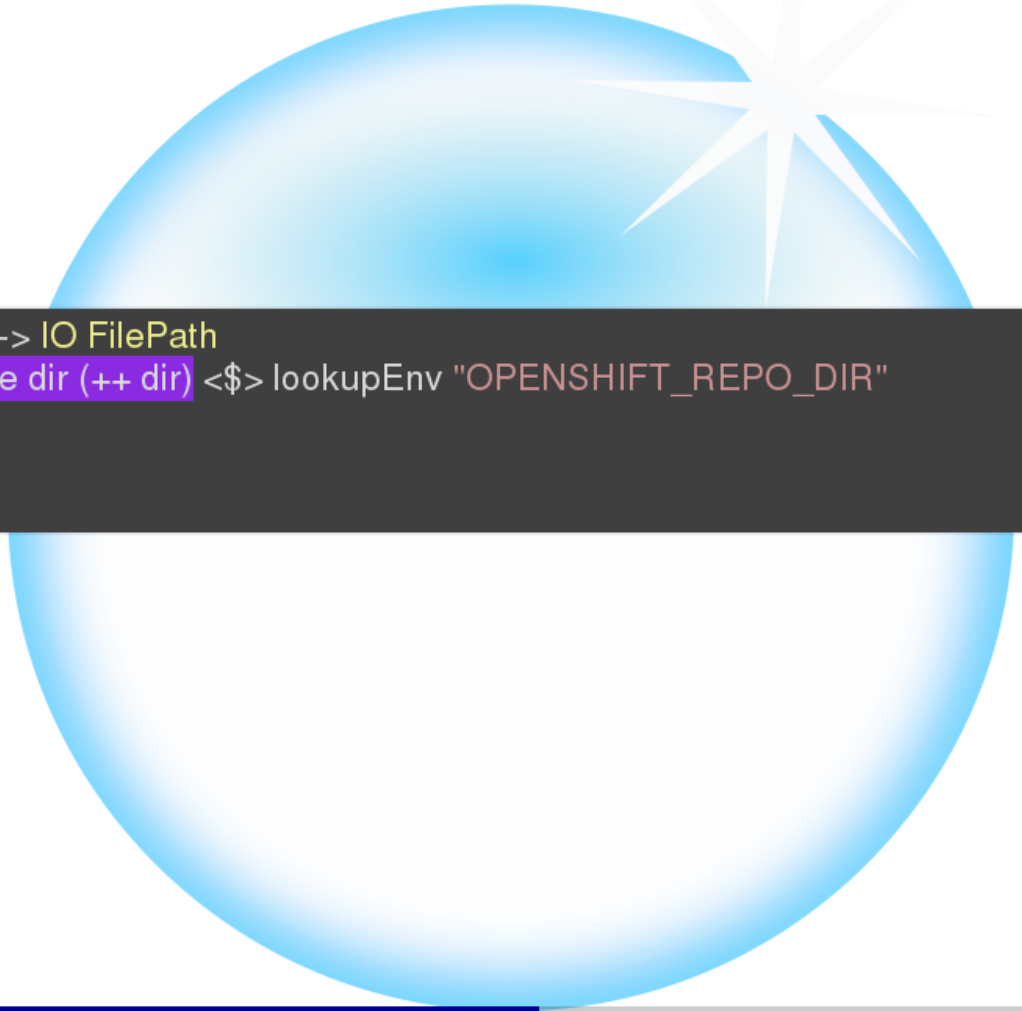
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs

ghci> :type catcon [1,2,3]
catcon [1,2,3] :: Num a => [a] -> [a]
ghci> :type map (catcon [1,2,3])
map (catcon [1,2,3]) :: Num a => [[a]] -> [[a]]
ghci> (\lists -> map (\list -> catcon [1,2,3] list) lists) [[4],[5],[6]]
[[4,1,2,3],[5,1,2,3],[6,1,2,3]]
```



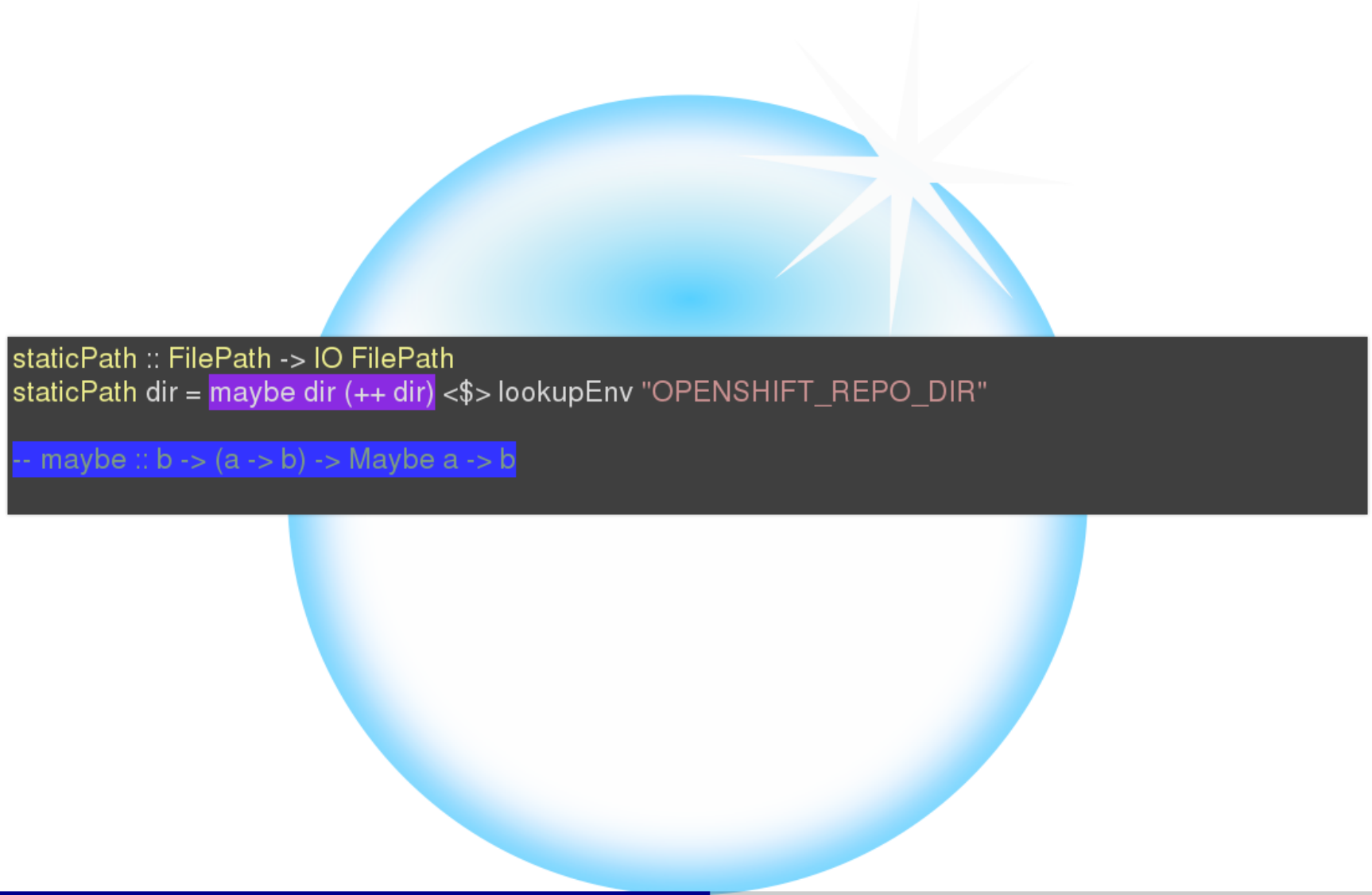
```
catcon :: [a] -> [a] -> [a]
catcon xs ys = ys ++ xs

ghci> :type catcon [1,2,3]
catcon [1,2,3] :: Num a => [a] -> [a]
ghci> :type map (catcon [1,2,3])
map (catcon [1,2,3]) :: Num a => [[a]] -> [[a]]
ghci> (\lists -> map (\list -> catcon [1,2,3] list) lists) [[4],[5],[6]]
[[4,1,2,3],[5,1,2,3],[6,1,2,3]]
```

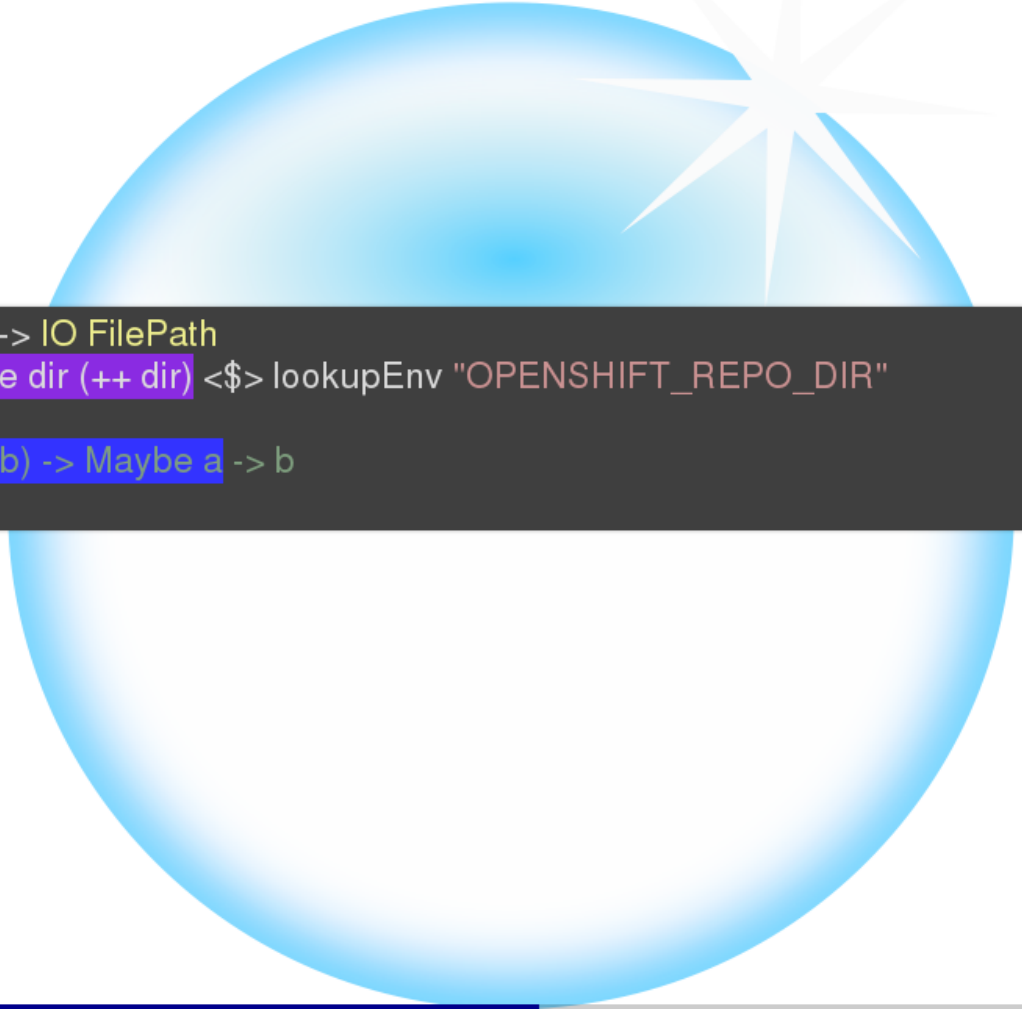


```
staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"
```

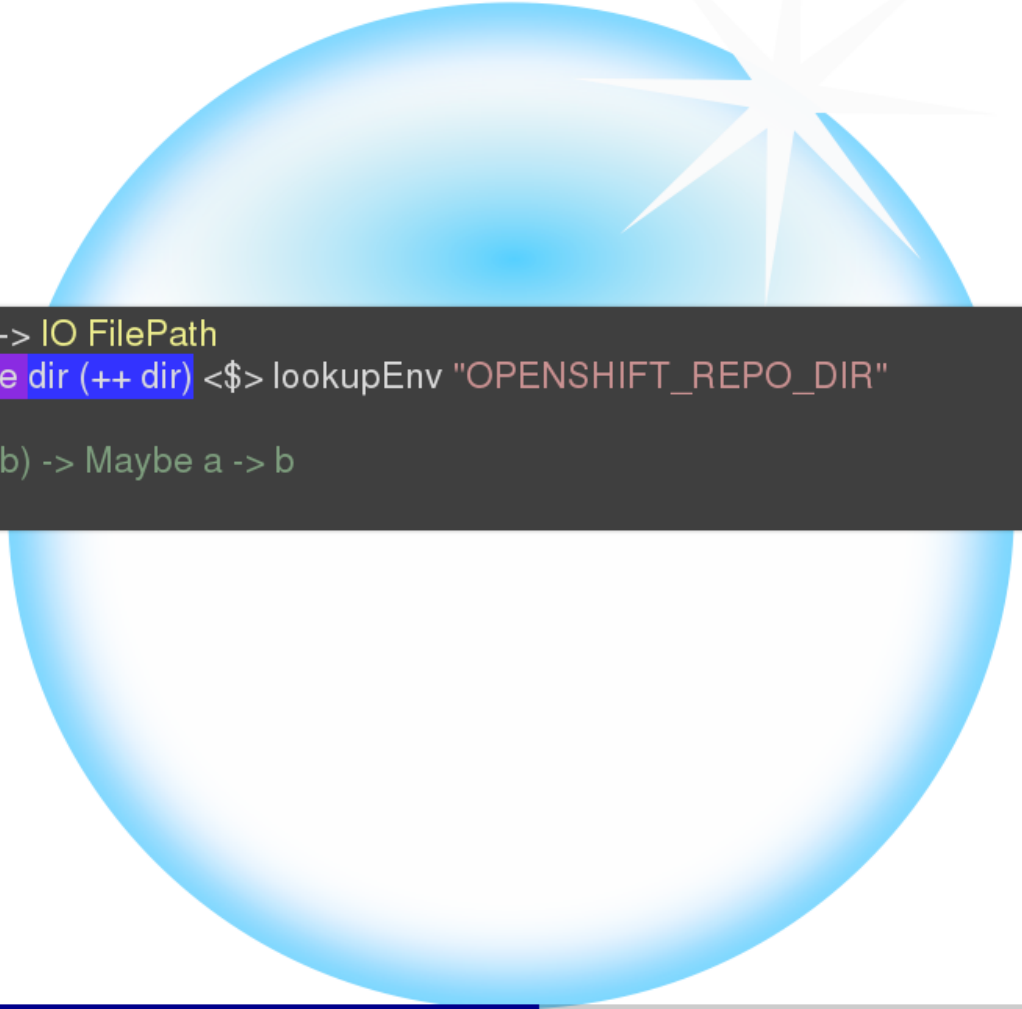


```
staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"
```

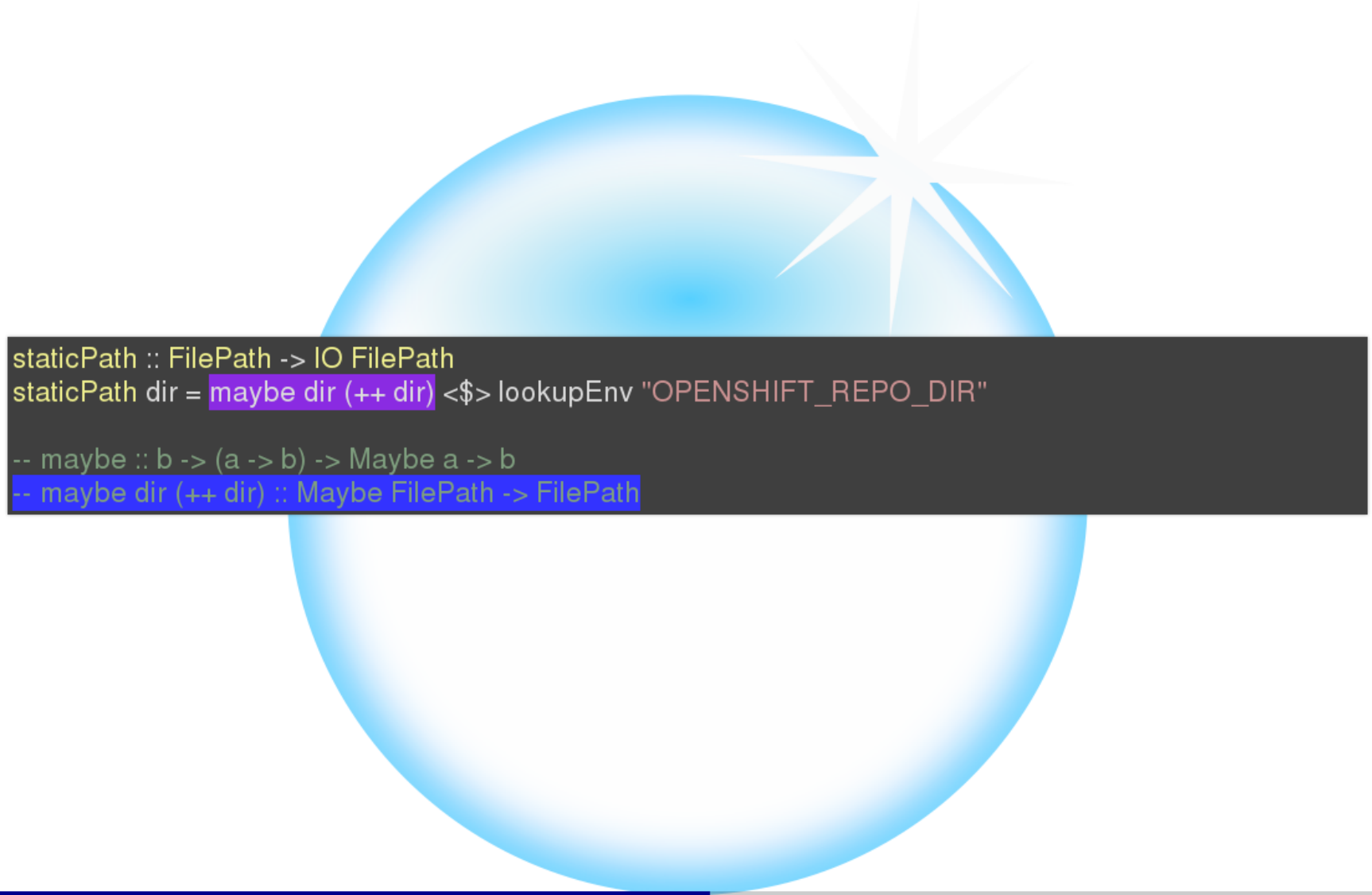

staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"
-- maybe :: b -> (a -> b) -> Maybe a -> b

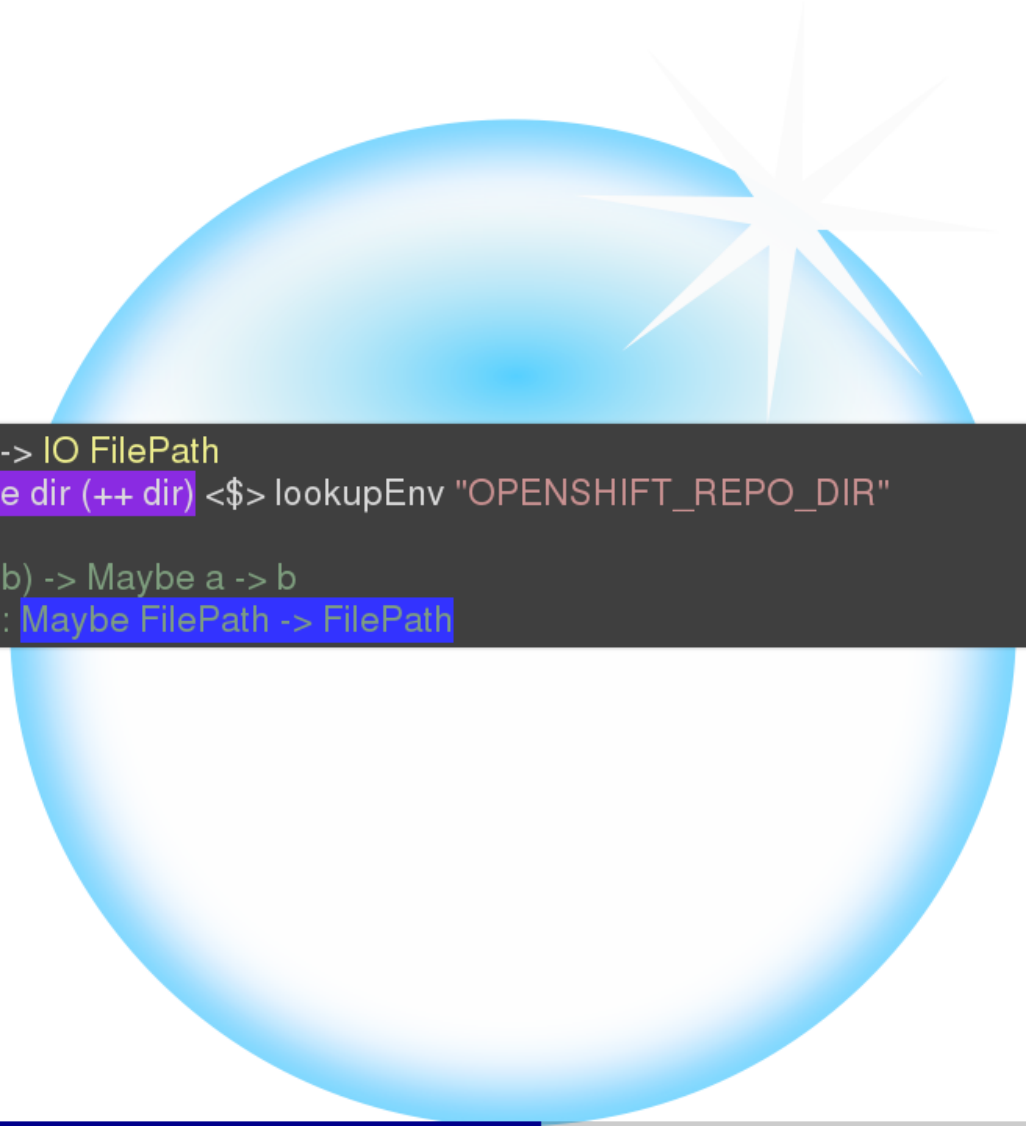


```haskell
staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"

-- maybe :: b -> (a -> b) -> Maybe a -> b
```



```haskell
staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"

-- maybe :: b -> (a -> b) -> Maybe a -> b
```


staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"
-- maybe :: b -> (a -> b) -> Maybe a -> b
-- maybe dir (++ dir) :: Maybe FilePath -> FilePath



```haskell
staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"

-- maybe :: b -> (a -> b) -> Maybe a -> b
-- maybe dir (++ dir) :: Maybe FilePath -> FilePath
```

Closure

WHAT: An instance of a function together with the environment in which it was created

WHY: Allows functions to access variables from their context even when it no longer exists; can make code more concise

```
ghci> (\ch n -> (\str -> take n (repeat ch) ++ str)) 'a' 5 "ha"
```

```
ghci> (\ch n -> (\str -> take n (repeat ch) ++ str)) 'a' 5 "ha"
```

```
ghci> (\ch n -> (\str -> take n (repeat ch) ++ str)) 'a' 5 "ha"
```

```
ghci> (\ch n -> (\str -> take n (repeat ch) ++ str)) 'a' 5 "ha"
```

```
ghci> (\ch n -> (\str -> take n (repeat ch) ++ str)) 'a' 5 "ha"
```

```
ghci> (\ch n -> (\str -> take n (repeat ch) ++ str)) 'a' 5 "ha"
```

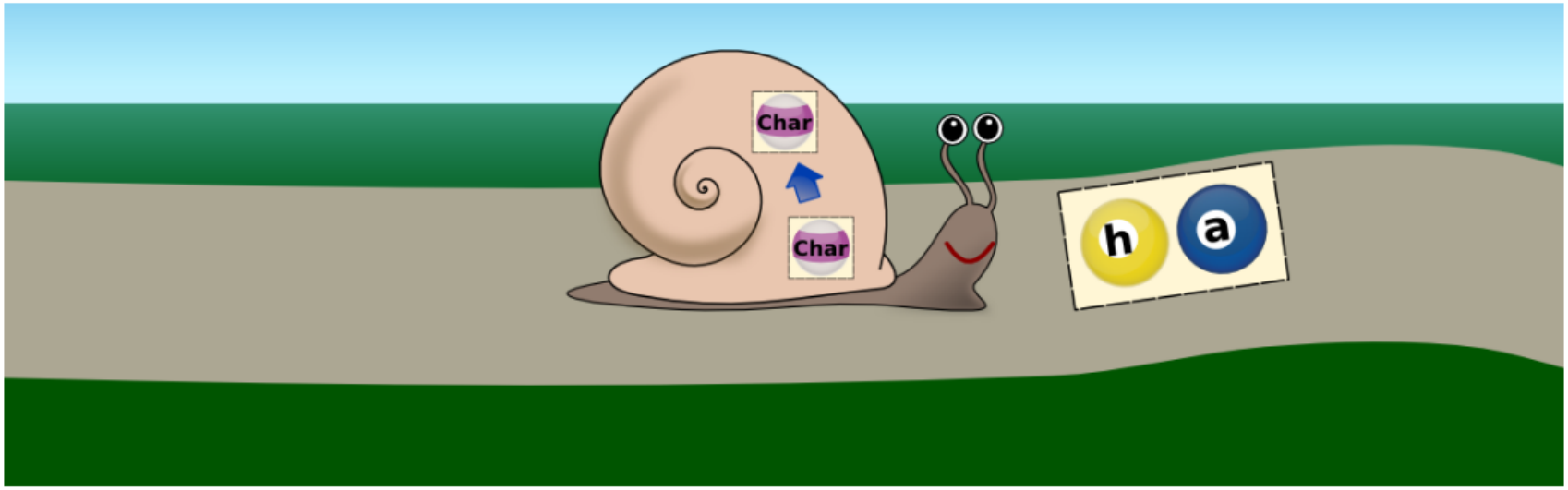
```
ghci> (\ch n -> (\str -> take n (repeat ch) ++ str)) 'a' 5 "ha"
```

```
ghci> (\ch n -> (\str -> take n (repeat ch) ++ str)) 'a' 5 "ha"
```

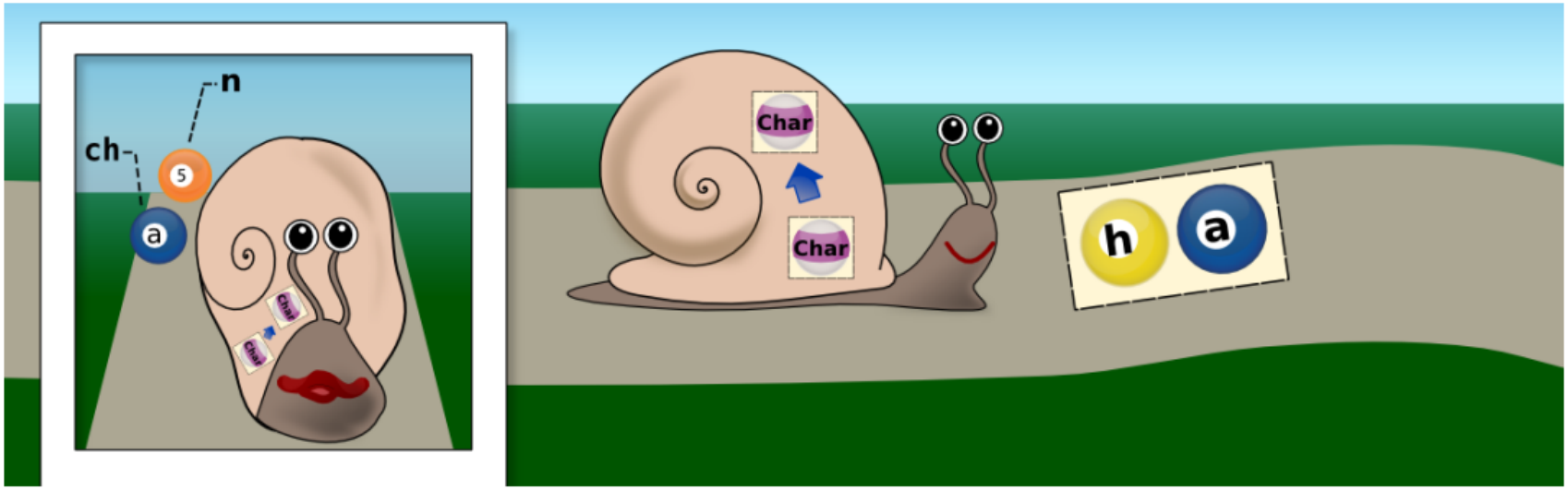
```
ghci> (\ch n -> (\str -> take n (repeat ch) ++ str)) 'a' 5 "ha"
```

```
ghci> (\ch n -> (\str -> take n (repeat ch) ++ str)) 'a' 5 "ha"
"aaaaaha"
```

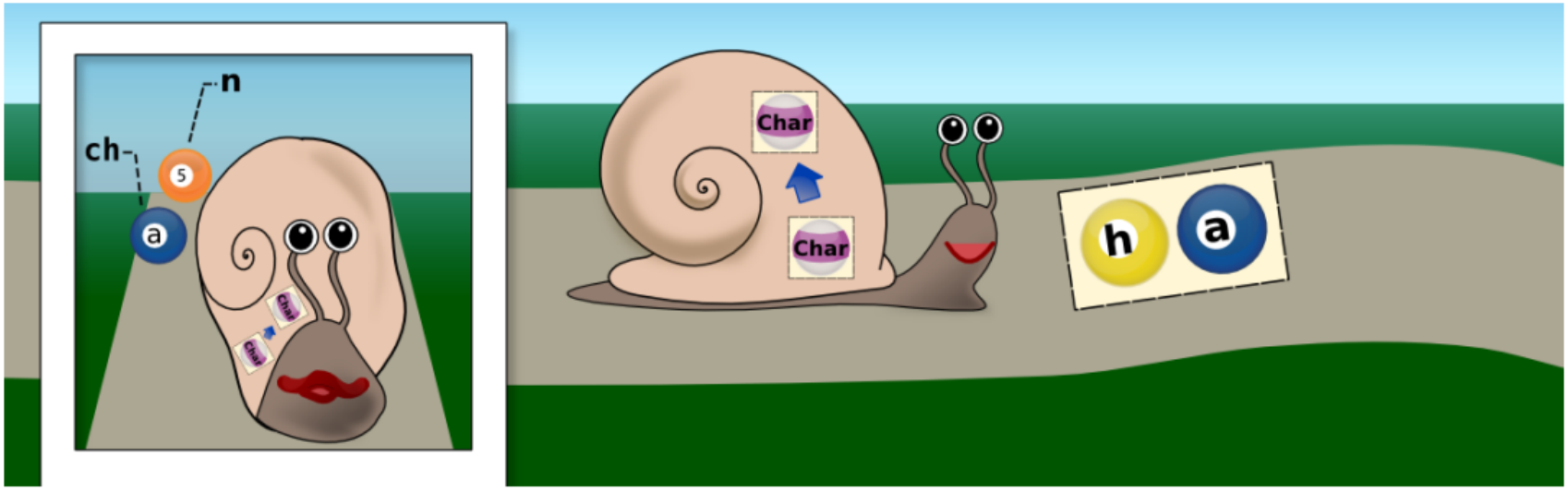
```
ghci> (\ch n -> (\str -> take n (repeat ch) ++ str)) 'a' 5 "ha"
"aaaaaha"
```



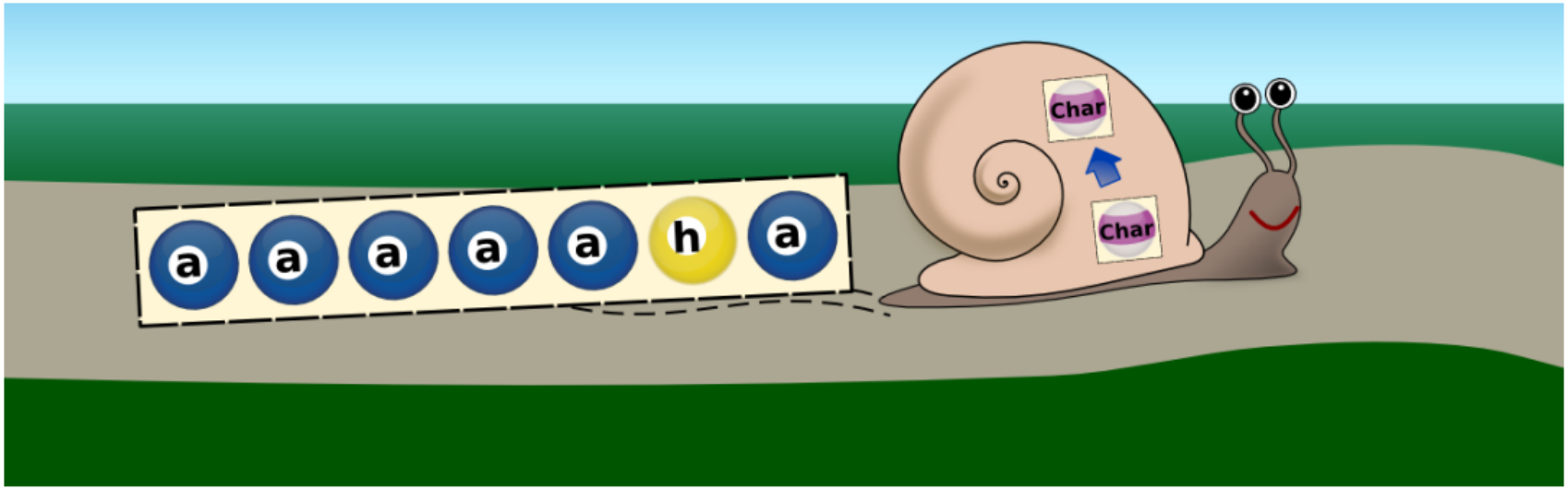
```
ghci> (\ch n -> (\str -> take n (repeat ch) ++ str)) 'a' 5 "ha"
"aaaaaha"
```



```
ghci> (\ch n -> (\str -> take n (repeat ch) ++ str)) 'a' 5 "ha"
"aaaaaha"
```



```
ghci> (\ch n -> (\str -> take n (repeat ch) ++ str)) 'a' 5 "ha"
"aaaaaha"
```




(\defs -> liftIO (shuffled defs) >>= (\sdefs -> renderGame (ws^.name) defs sdefs))


(\defs -> liftIO (shuffled defs) >>= (\sdefs -> renderGame (ws^.name) defs sdefs))


(\defs -> liftIO (shuffled defs) >>= (\sdefs -> renderGame (ws^.name) defs sdefs))


(\defs -> liftIO (shuffled defs) >>= (\sdefs -> renderGame (ws^.name) defs sdefs))

Functor

WHAT: Structure that can be mapped over

WHY: Identify common pattern; use abstraction to reduce code duplication

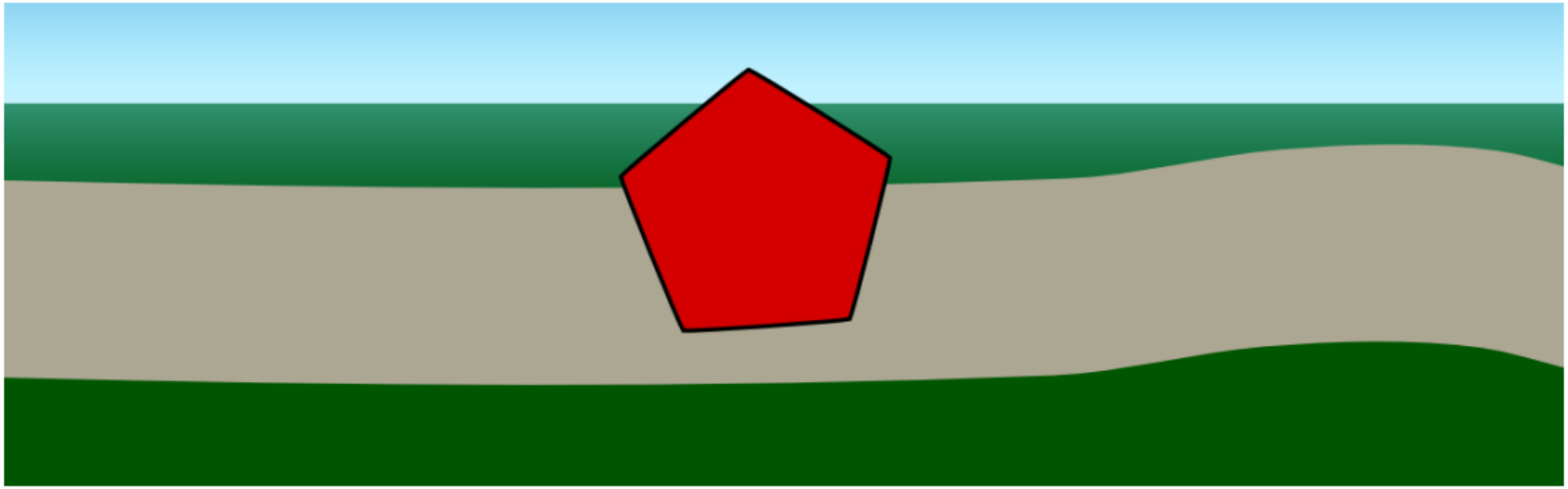
```
ghci> :info Functor
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  ...
```

```
ghci> :info Functor
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  ...
```

```
interface Functor<F> {
  <A, B> F<B> fmap(Function<A, B> f, F<A> a);
  ...
}
```

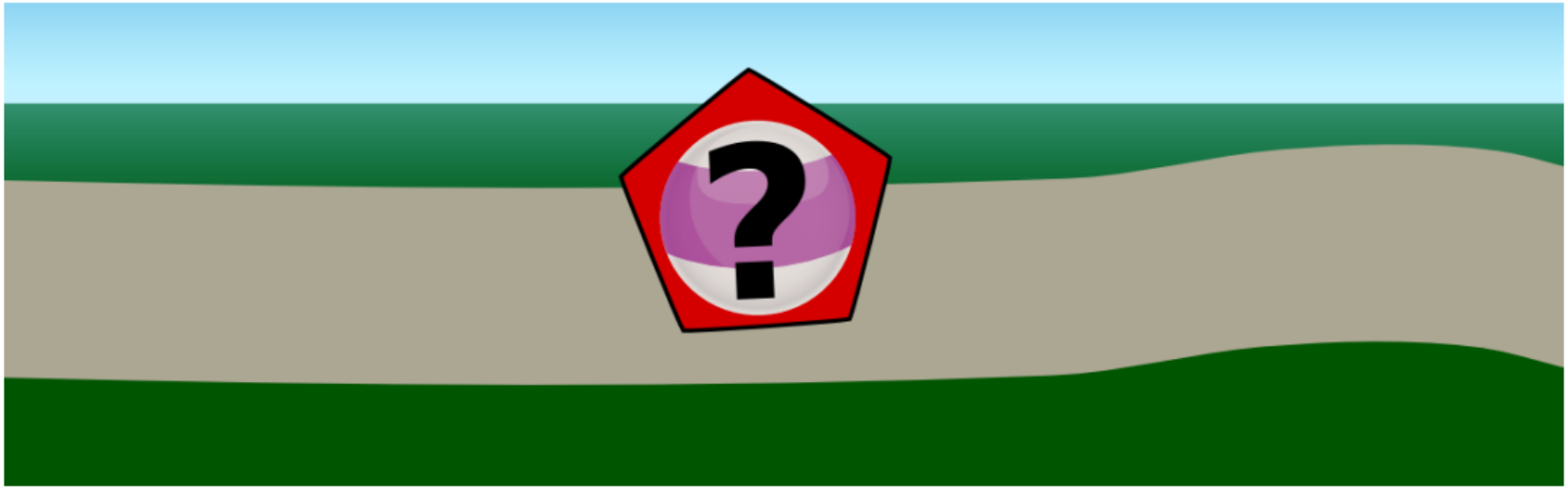
```
ghci> :info Functor
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  ...
```

```
interface Functor<F> {
  <A, B> F<B> fmap(Function<A, B> f, F<A> a);
  ...
}
```



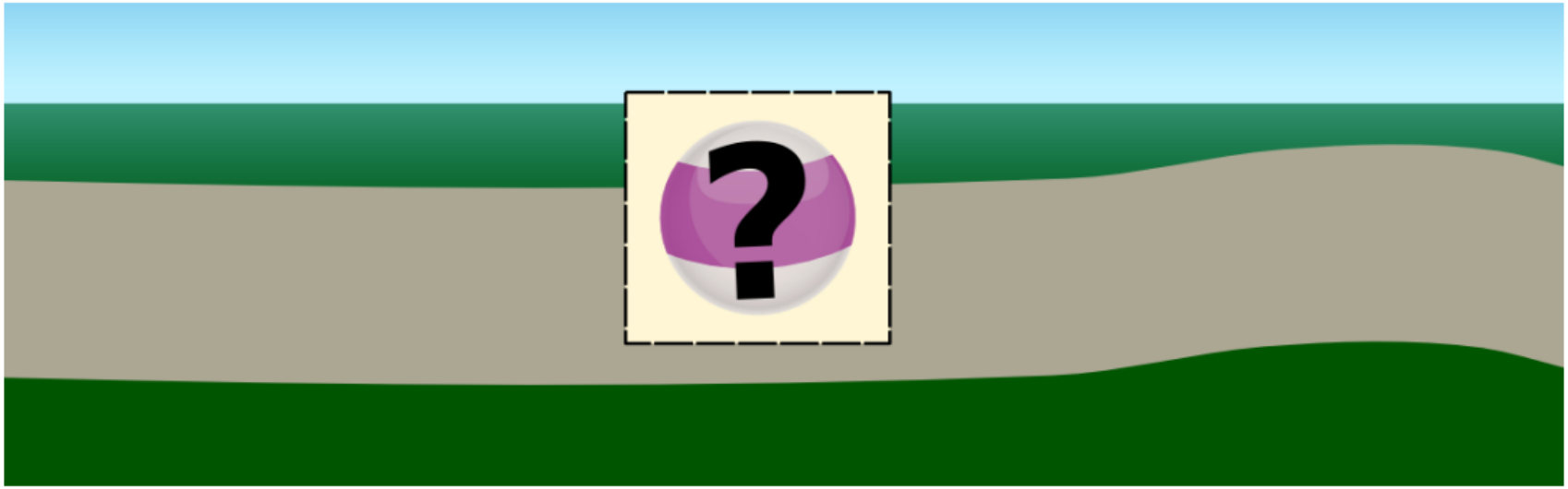
```
ghci> :info Functor
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  ...
```

```
interface Functor<F> {
  <A, B> F<B> fmap(Function<A, B> f, F<A> a);
  ...
}
```



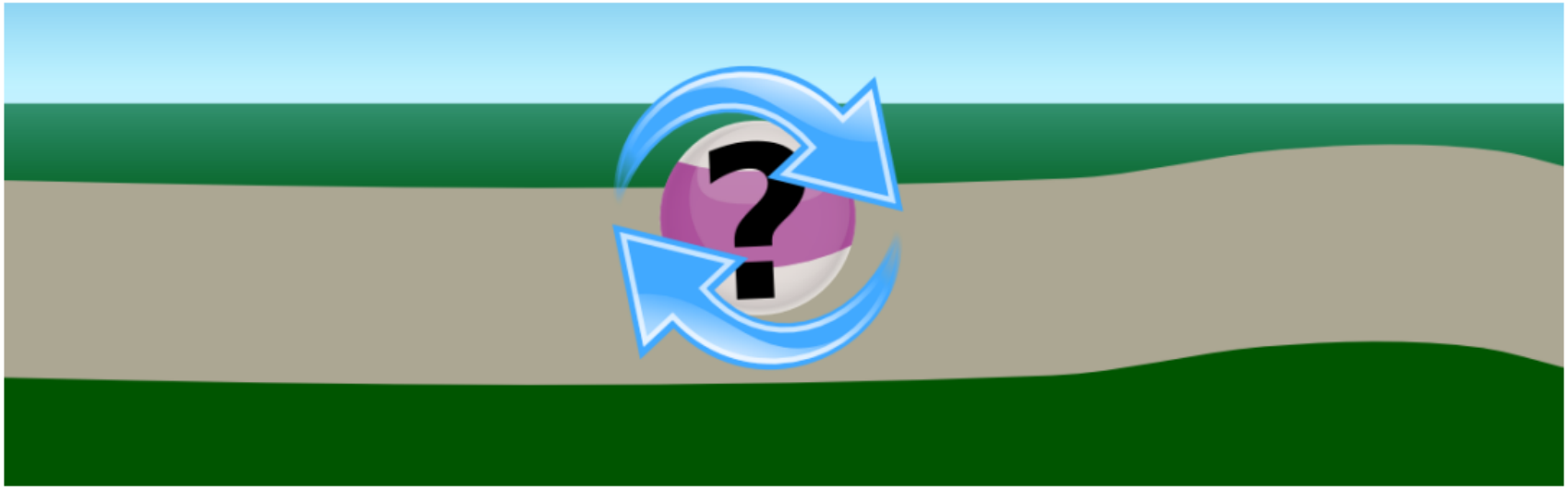
```
ghci> :info Functor
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  ...
```

```
interface Functor<F> {
  <A, B> F<B> fmap(Function<A, B> f, F<A> a);
  ...
}
```



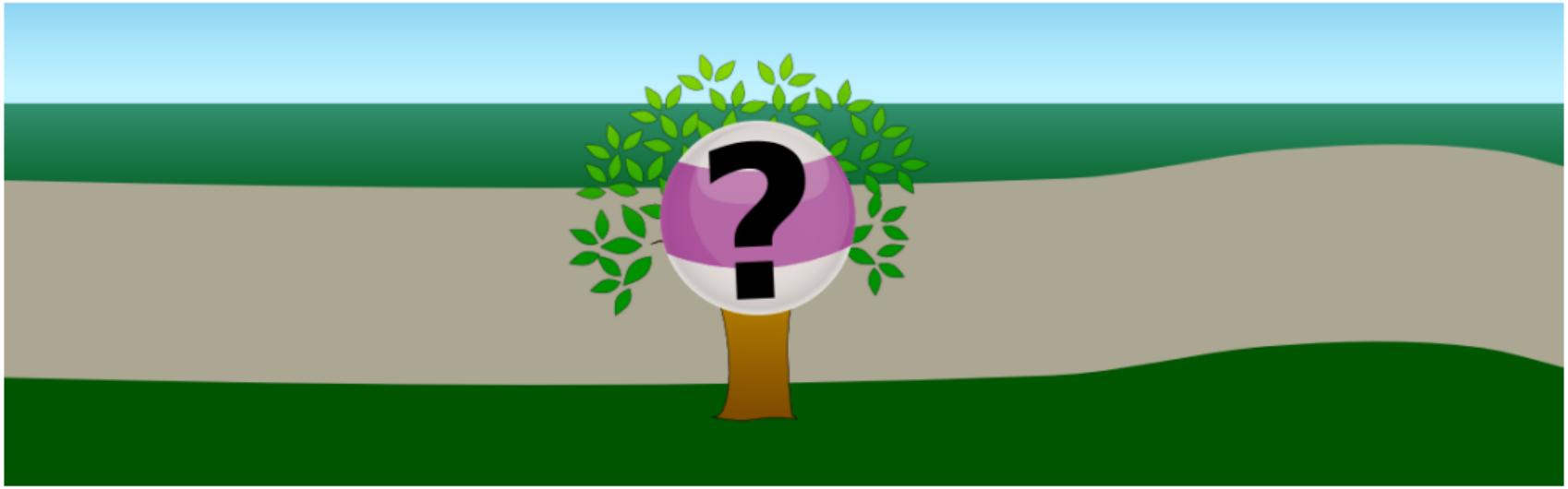
```
ghci> :info Functor
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  ...
```

```
interface Functor<F> {
  <A, B> F<B> fmap(Function<A, B> f, F<A> a);
  ...
}
```



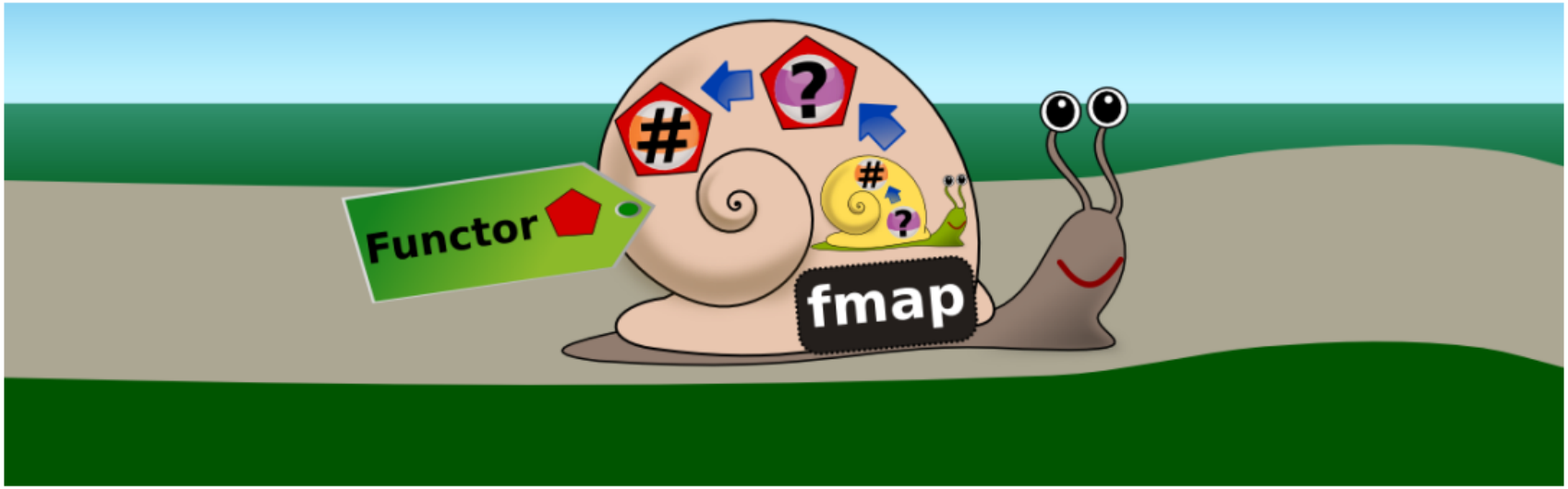
```
ghci> :info Functor
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  ...
```

```
interface Functor<F> {
  <A, B> F<B> fmap(Function<A, B> f, F<A> a);
  ...
}
```

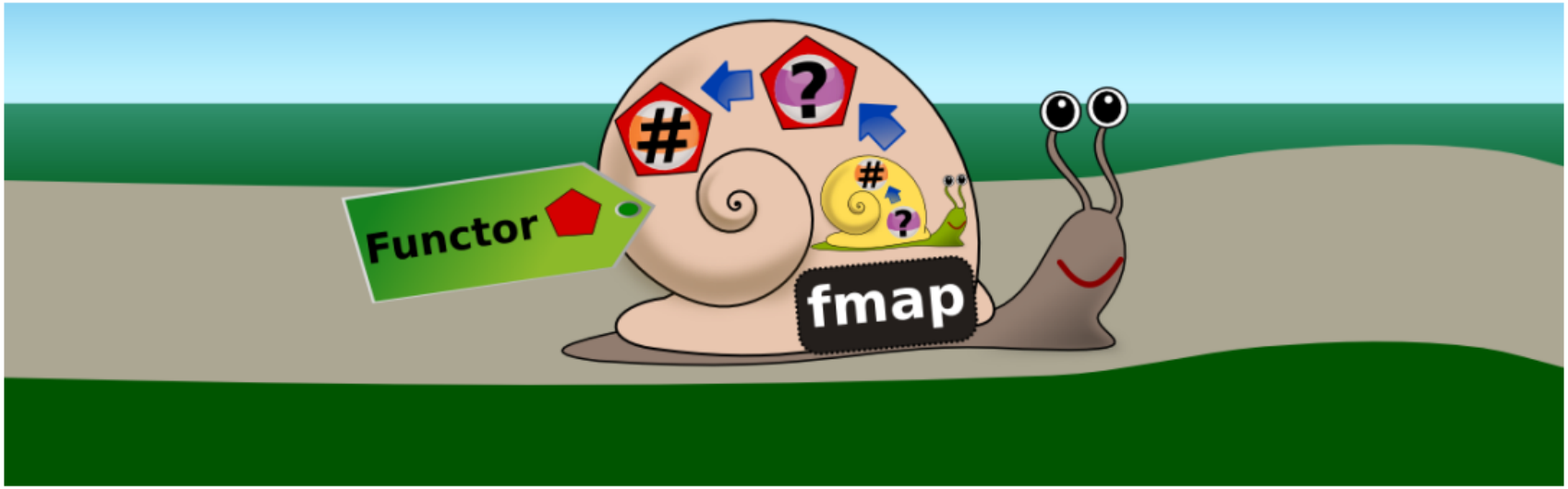


```
ghci> :info Functor
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  ...
```

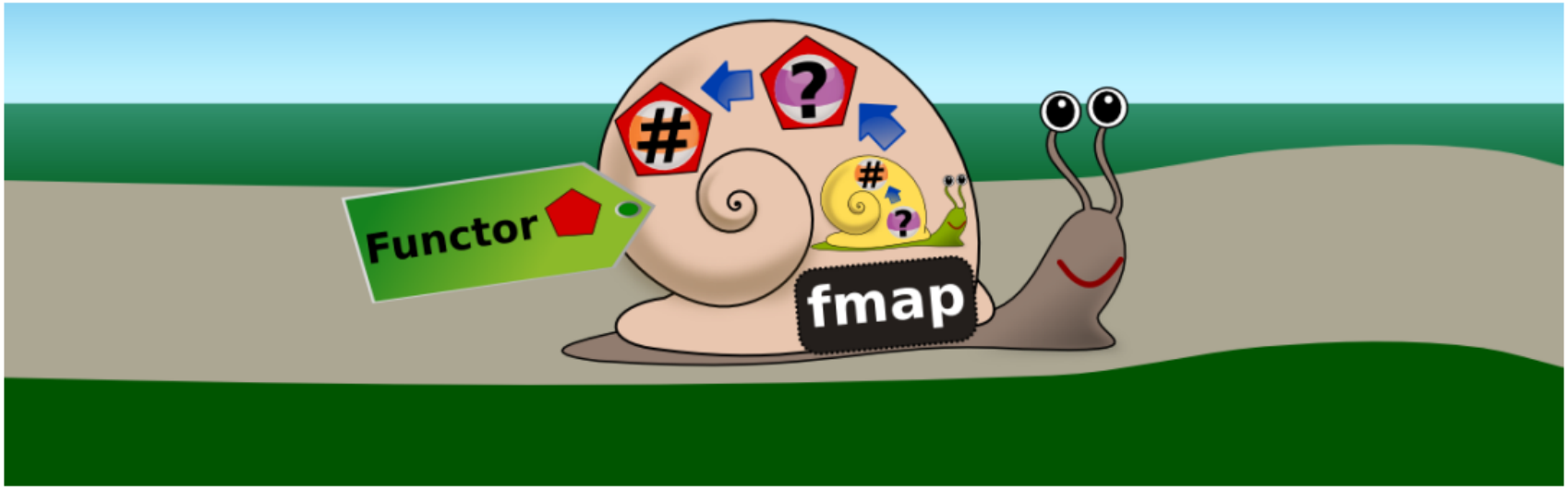
```
interface Functor<F> {
  <A, B> F<B> fmap(Function<A, B> f, F<A> a);
  ...
}
```

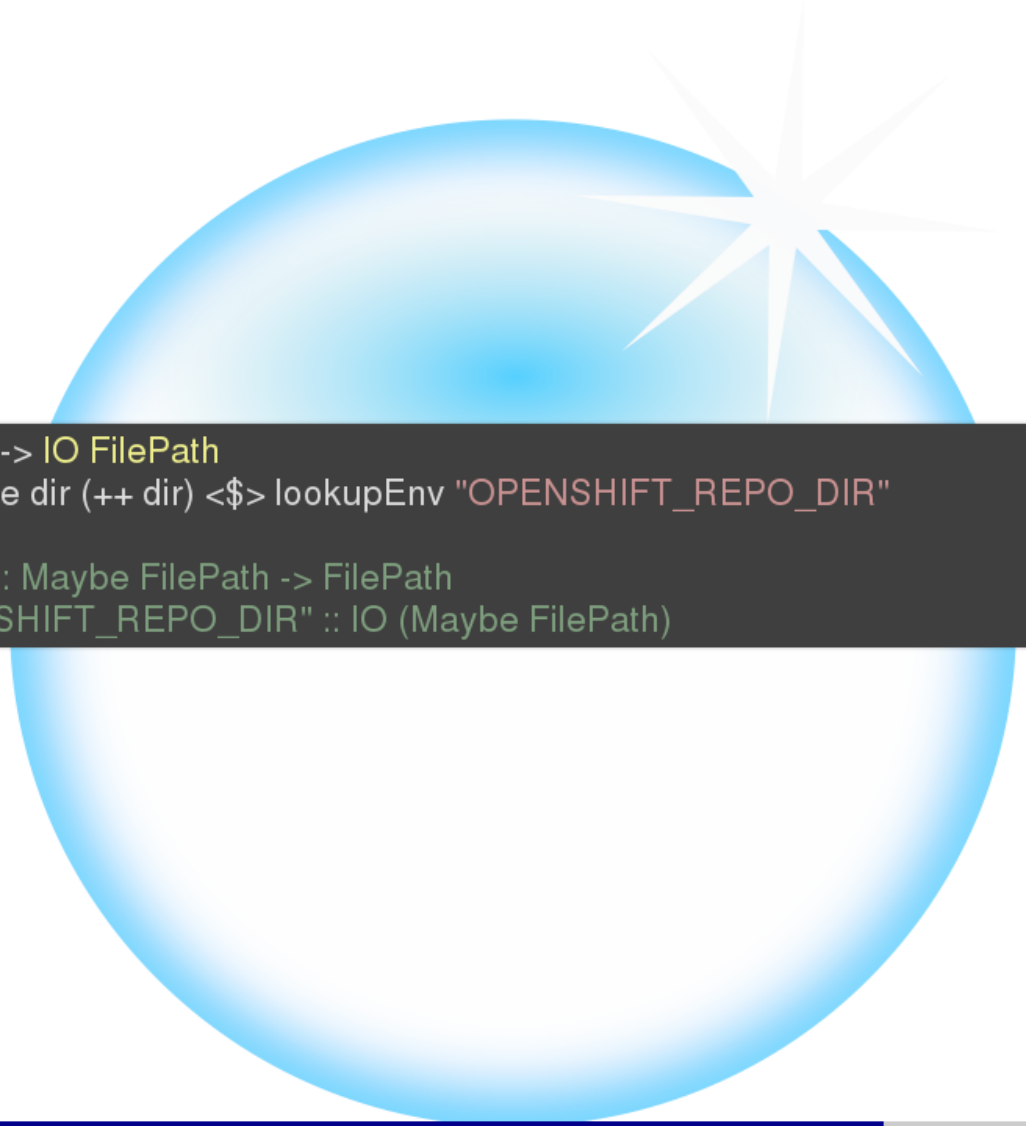


```
ghci> :info Functor
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  ...
ghci> fmap (*2) [1,2,3]
[2,4,6]
```

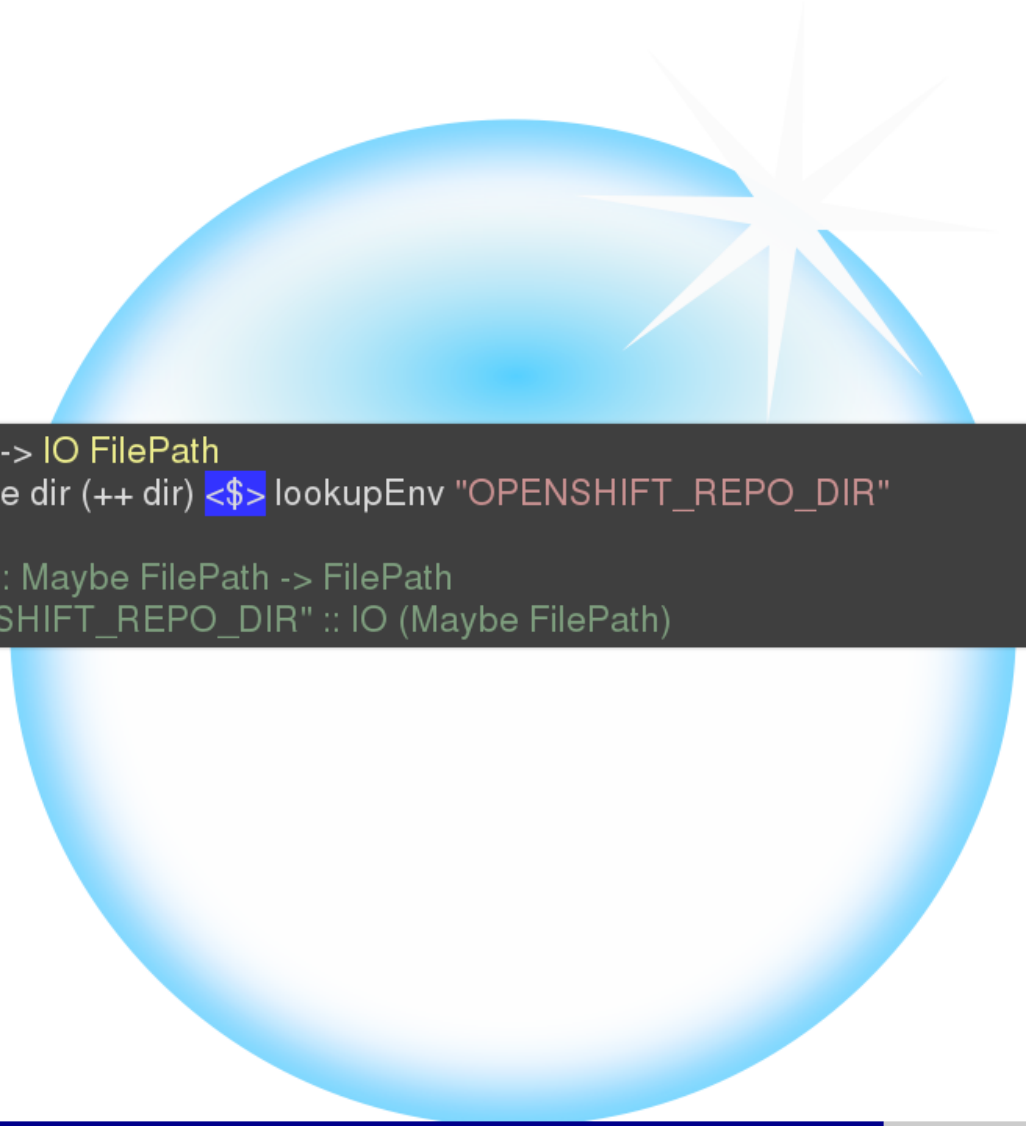


```
ghci> :info Functor
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  ...
ghci> fmap (*2) [1,2,3]
[2,4,6]
ghci> fmap reverse getLine
foo
"oof"
```





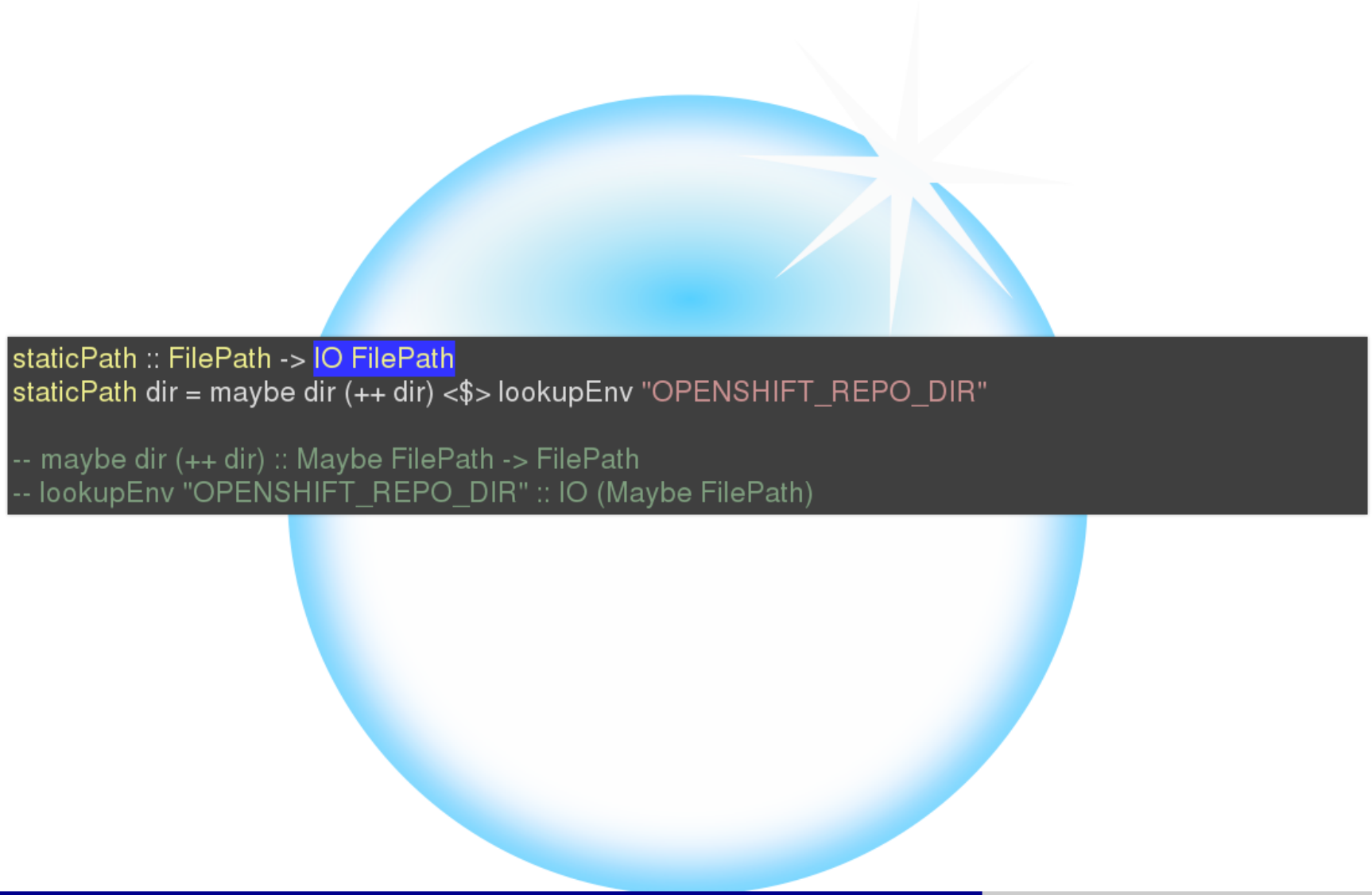
```
staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"

-- maybe dir (++ dir) :: Maybe FilePath -> FilePath
-- lookupEnv "OPENSHIFT_REPO_DIR" :: IO (Maybe FilePath)
```



```haskell
staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"

-- maybe dir (++ dir) :: Maybe FilePath -> FilePath
-- lookupEnv "OPENSHIFT_REPO_DIR" :: IO (Maybe FilePath)
```

```haskell
staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"

-- maybe dir (++ dir) :: Maybe FilePath -> FilePath
-- lookupEnv "OPENSHIFT_REPO_DIR" :: IO (Maybe FilePath)
```


staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"
-- maybe dir (++ dir) :: Maybe FilePath -> FilePath
-- lookupEnv "OPENSHIFT_REPO_DIR" :: IO (Maybe FilePath)


staticPath :: FilePath -> IO FilePath
staticPath dir = maybe dir (++ dir) <$> lookupEnv "OPENSHIFT_REPO_DIR"
-- maybe dir (++ dir) :: Maybe FilePath -> FilePath
-- lookupEnv "OPENSHIFT_REPO_DIR" :: IO (Maybe FilePath)

Monad

WHAT: Structure that puts values in a computational context and implements functions that facilitate the chaining of these computations

WHY: A large number of very useful functions can be written once for any type that fits the pattern

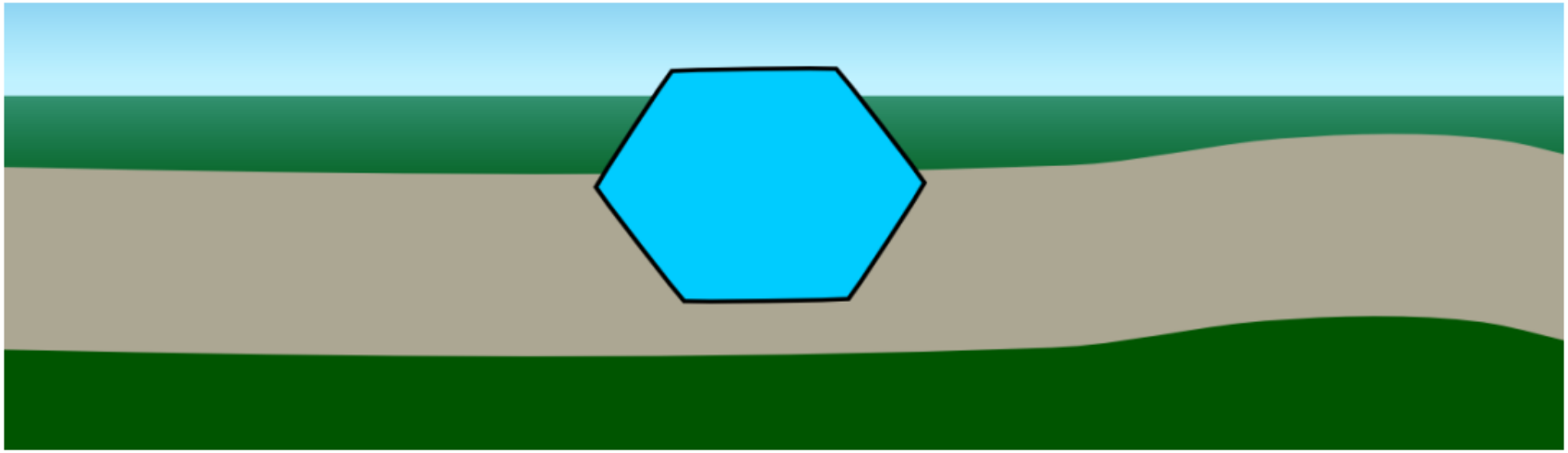
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
```

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

```
interface Monad<M> {
  <A, B> M<B> bind(M<A> m, Function<A, M<B>> f);
  <A> M<A> return(A a);
  ...
}
```

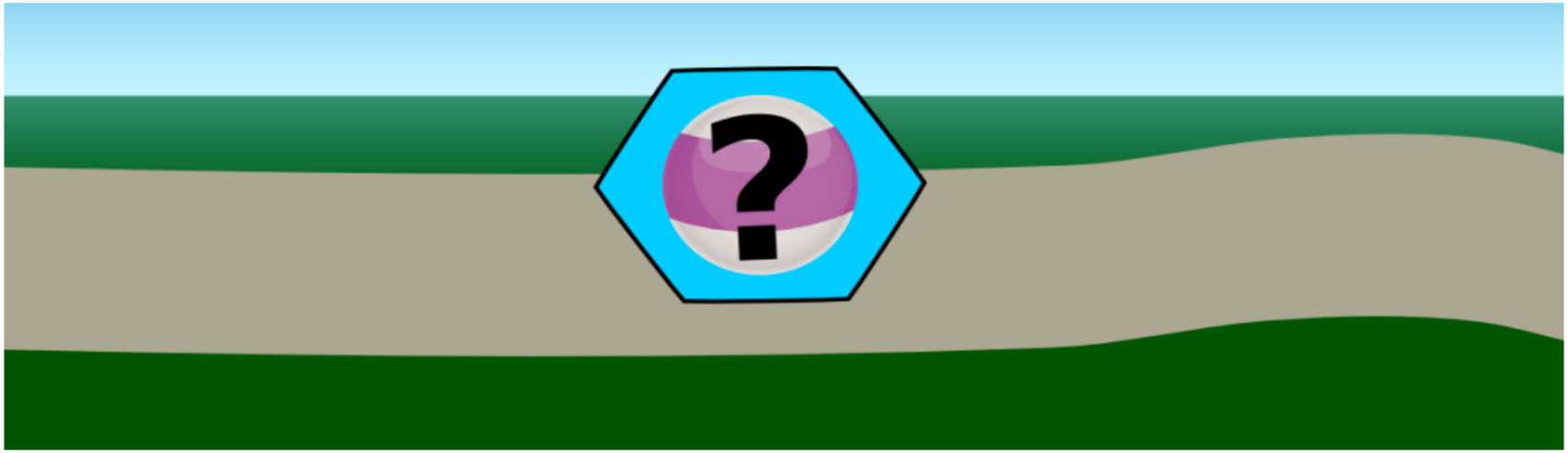
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
```

```
interface Monad<M> {
  <A, B> M<B> bind(M<A> m, Function<A, M<B>> f);
  <A> M<A> return(A a);
  ...
}
```



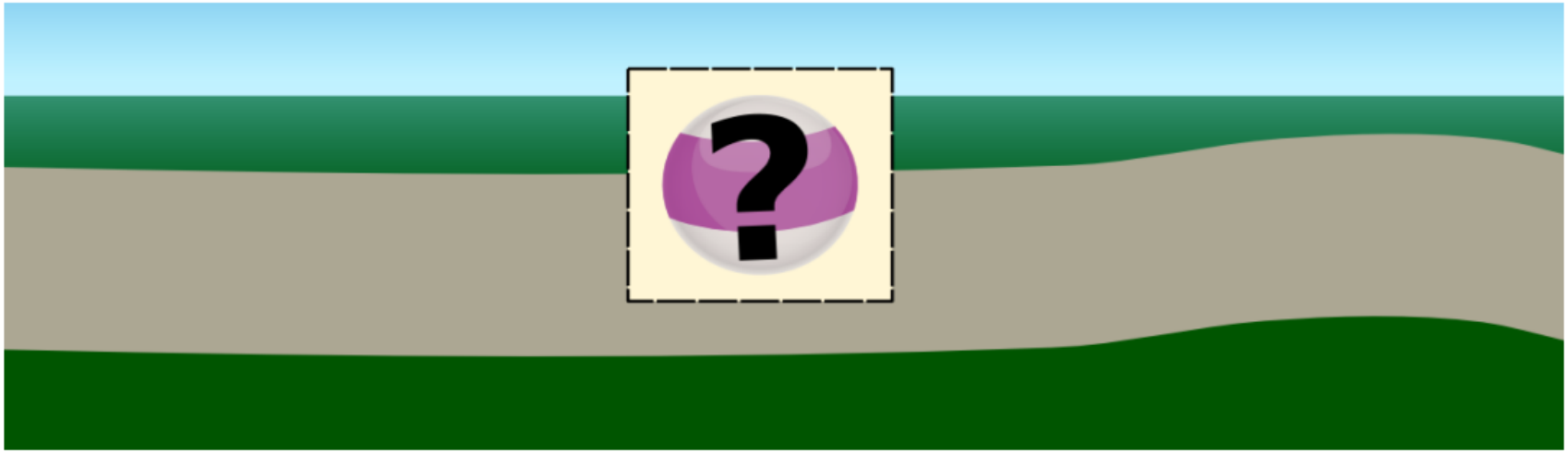
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
```

```
interface Monad<M> {
  <A, B> M<B> bind(M<A> m, Function<A, M<B>> f);
  <A> M<A> return(A a);
  ...
}
```



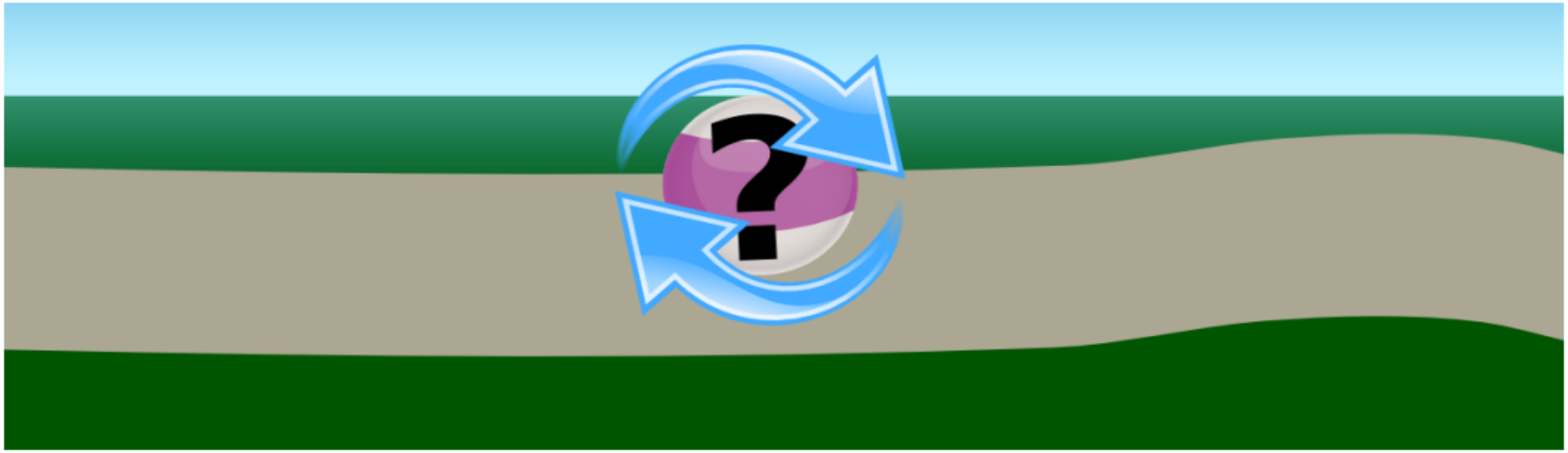
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
```

```
interface Monad<M> {
  <A, B> M<B> bind(M<A> m, Function<A, M<B>> f);
  <A> M<A> return(A a);
  ...
}
```

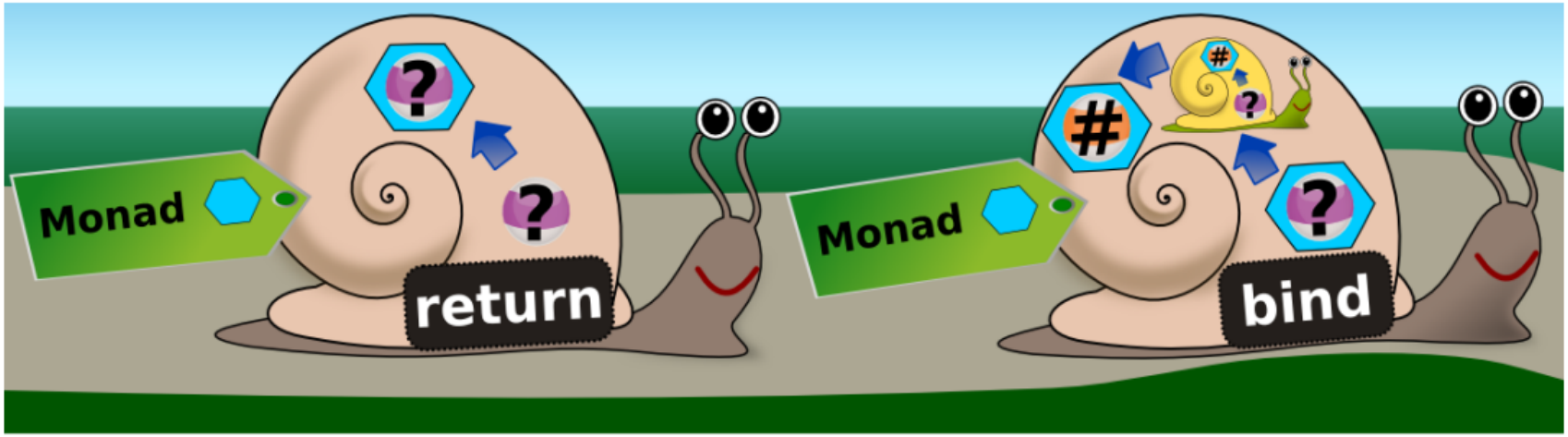


```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
```

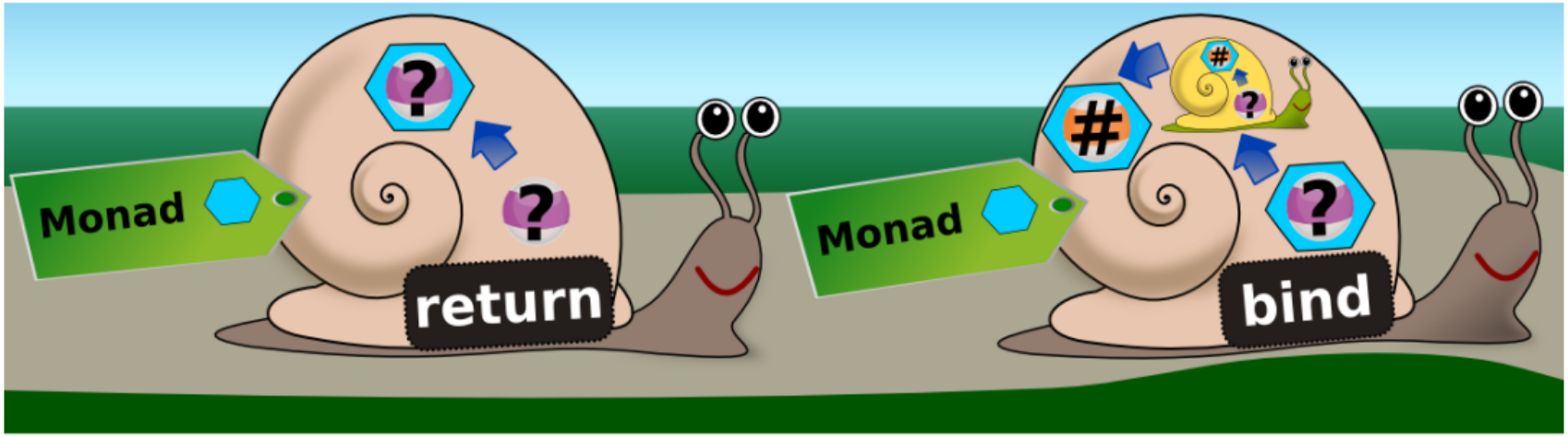
```
interface Monad<M> {
  <A, B> M<B> bind(M<A> m, Function<A, M<B>> f);
  <A> M<A> return(A a);
  ...
}
```



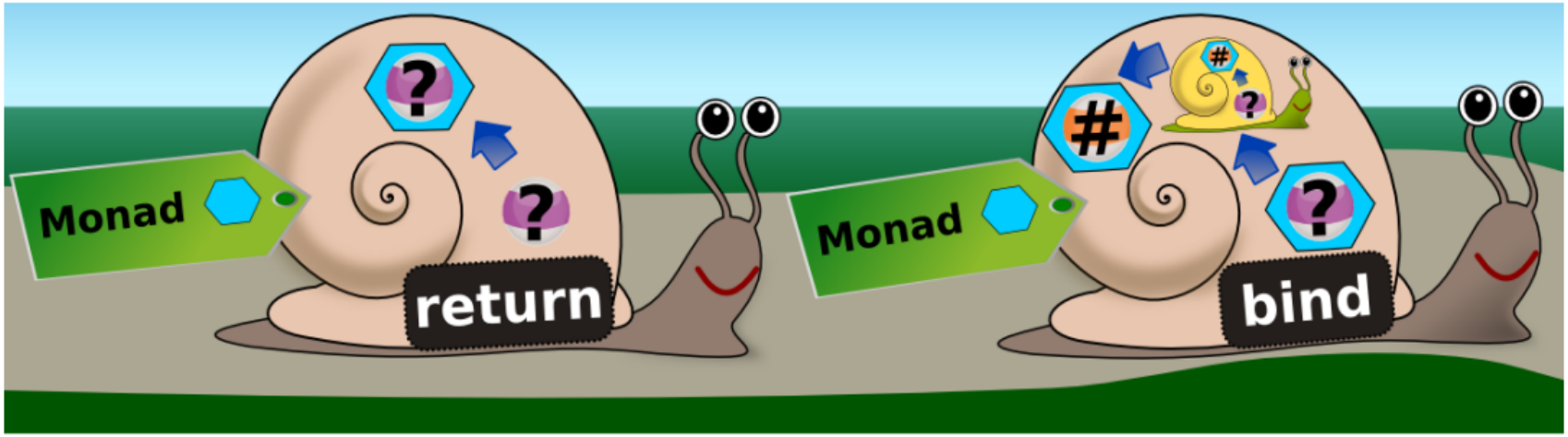
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
```



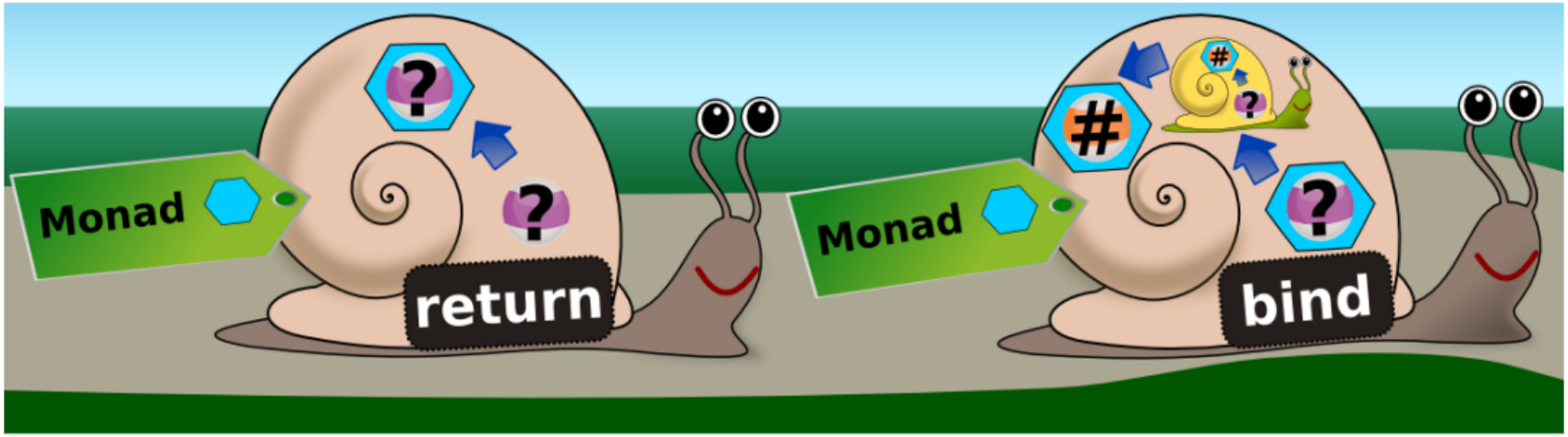
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
ghci> ["walk","nap","fight"] >>= (\t -> return ("cat" ++ t))
```



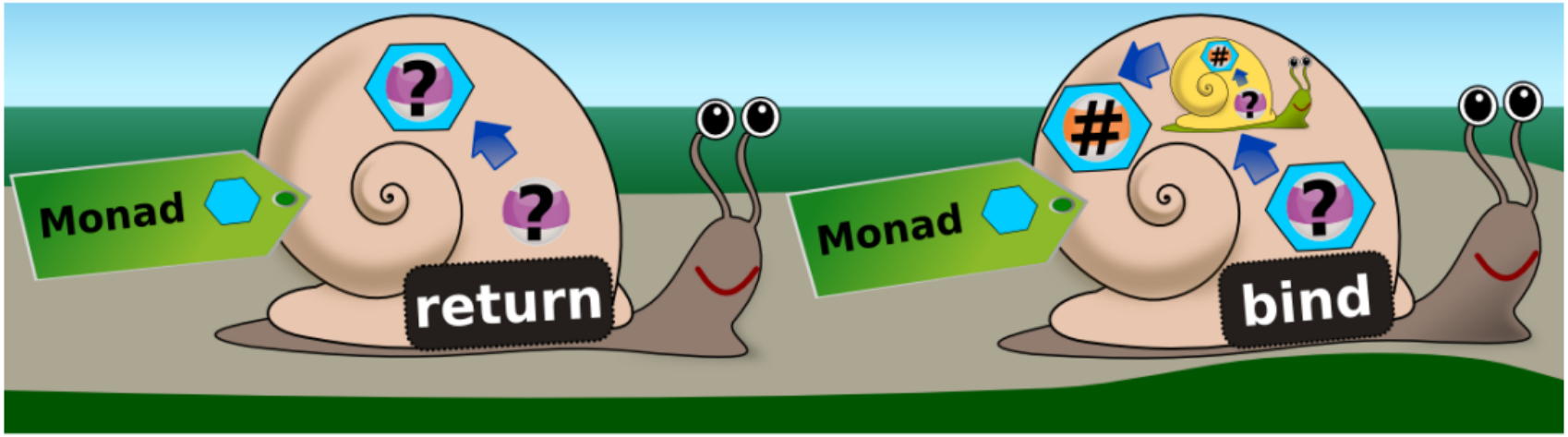
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
ghci> ["walk","nap","fight"] >>= (\t -> return ("cat" ++ t))
```



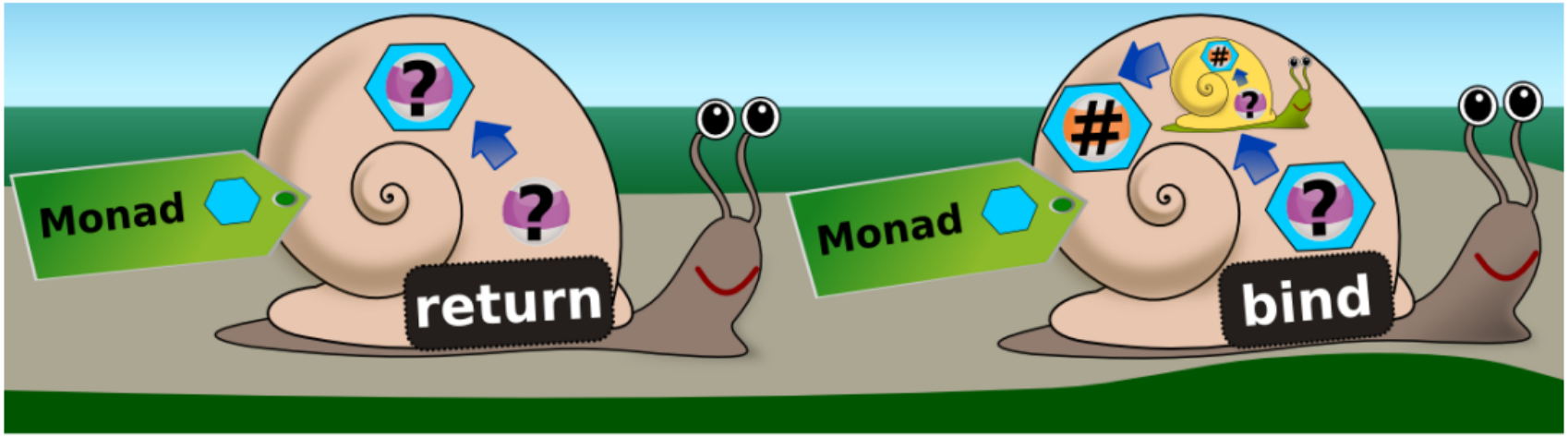
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
ghci> ["walk","nap","fight"] >>= (\t -> return ("cat" ++ t))
```



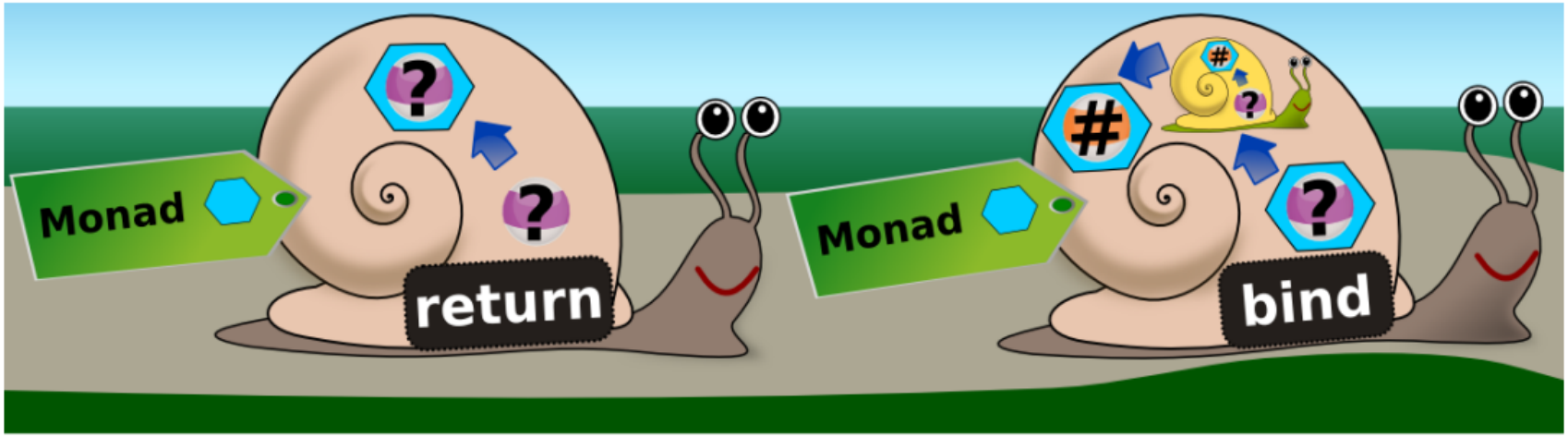
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
ghci> ["walk","nap","fight"] >>= (\t -> return ("cat" ++ t))
```



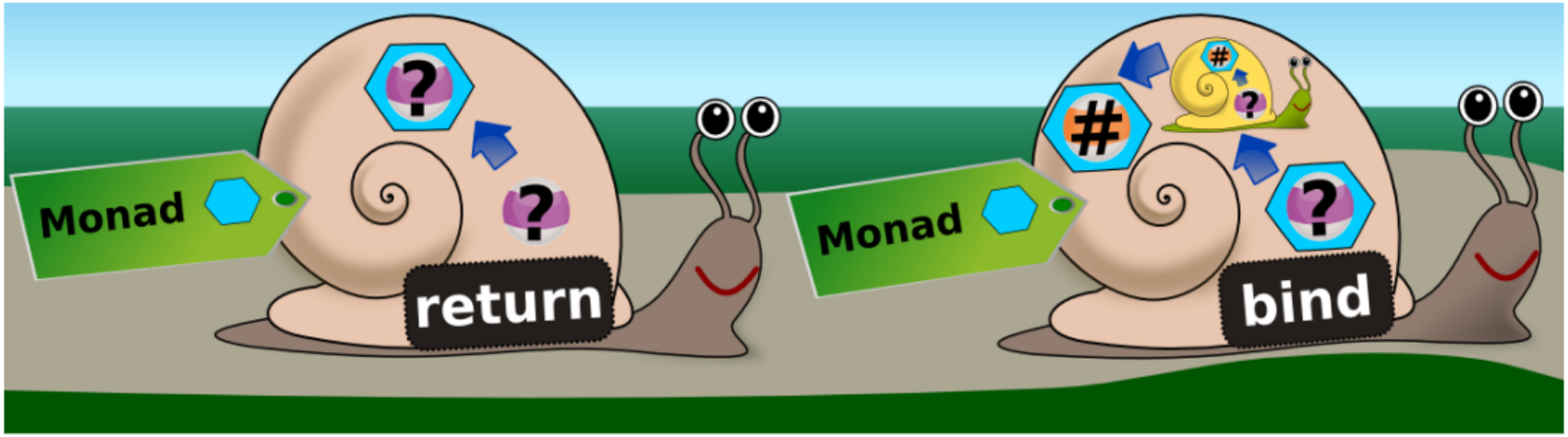
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
ghci> ["walk","nap","fight"] >>= (\t -> return ("cat" ++ t))
["catwalk","catnap","catfight"]
```



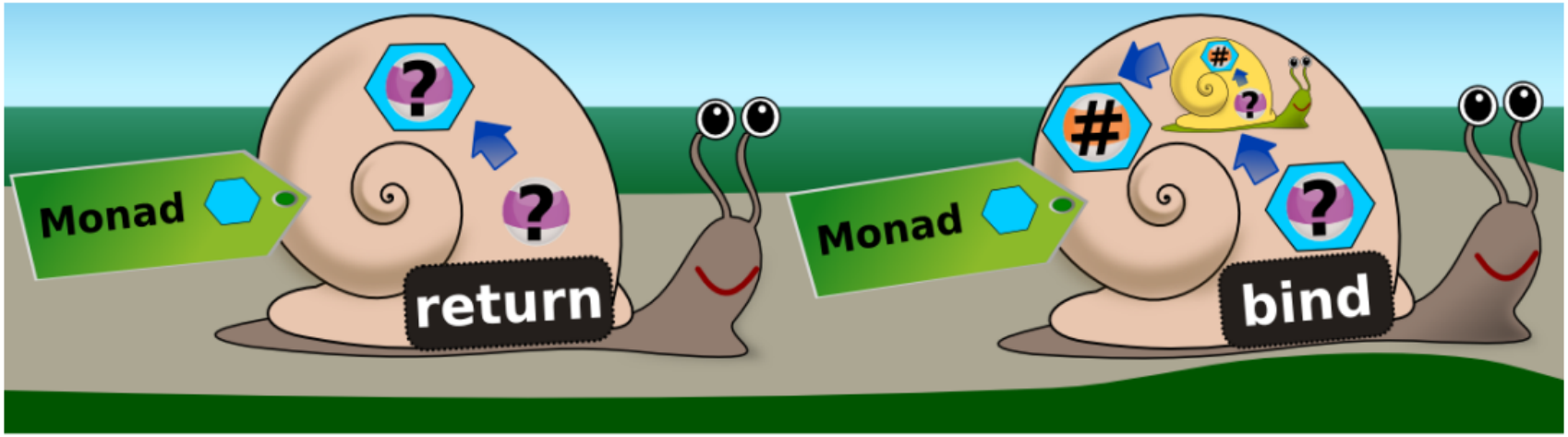
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
ghci> ["walk","nap","fight"] >>= (\t -> return ("cat" ++ t))
["catwalk","catnap","catfight"]
ghci> ["walk","nap","fight"] >>= (\t -> ["cat","dog"] >>= (\a -> return (a ++ t)))
```



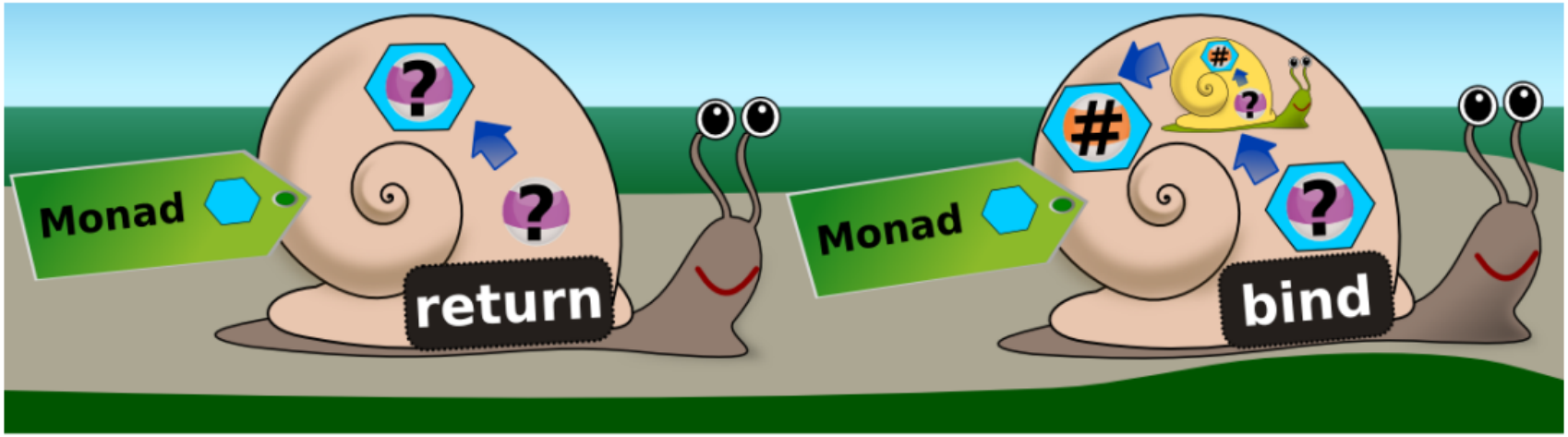
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
ghci> ["walk","nap","fight"] >>= (\t -> return ("cat" ++ t))
["catwalk","catnap","catfight"]
ghci> ["walk","nap","fight"] >>= (\t -> ["cat","dog"] >>= (\a -> return (a ++ t)))
```



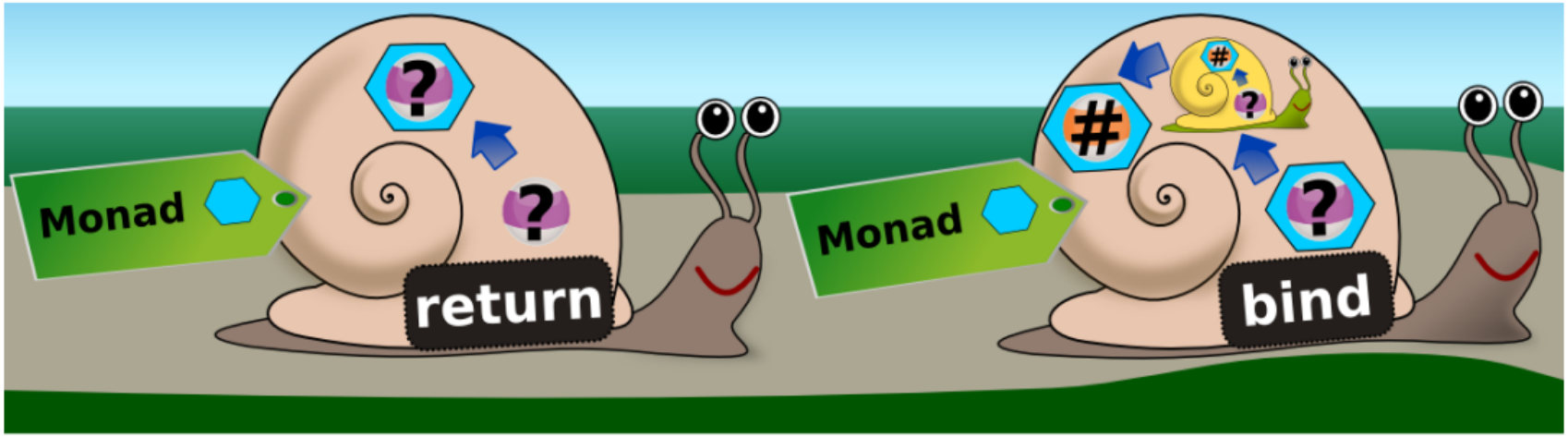
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
ghci> ["walk","nap","fight"] >>= (\t -> return ("cat" ++ t))
["catwalk","catnap","catfight"]
ghci> ["walk","nap","fight"] >>= (\t -> ["cat","dog"] >>= (\a -> return (a ++ t)))
```



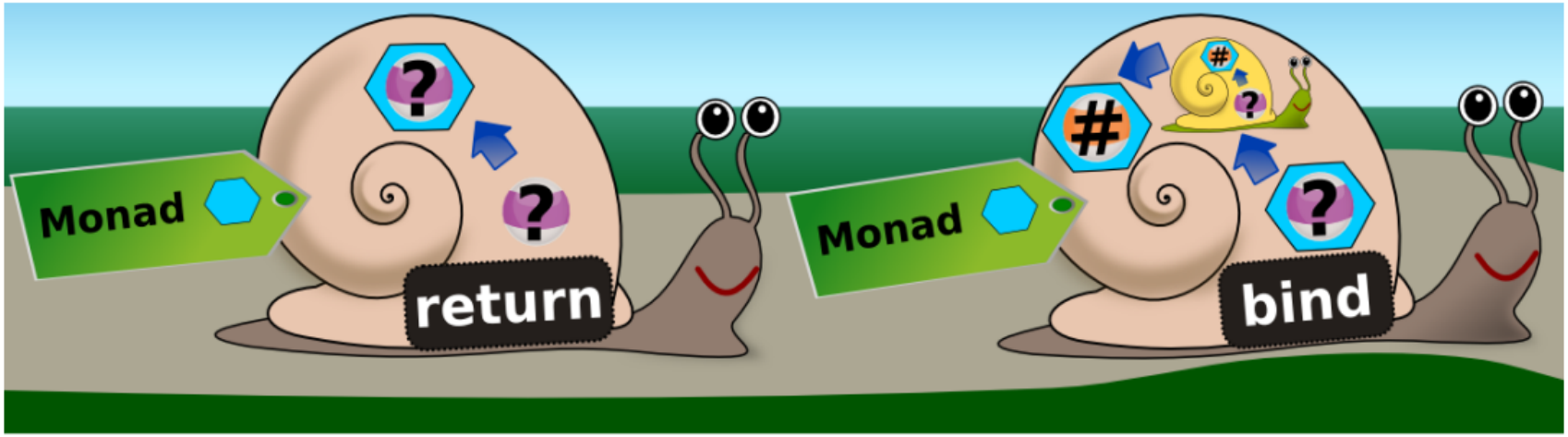
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
ghci> ["walk","nap","fight"] >>= (\t -> return ("cat" ++ t))
["catwalk","catnap","catfight"]
ghci> ["walk","nap","fight"] >>= (\t -> ["cat","dog"] >>= (\a -> return (a ++ t)))
```



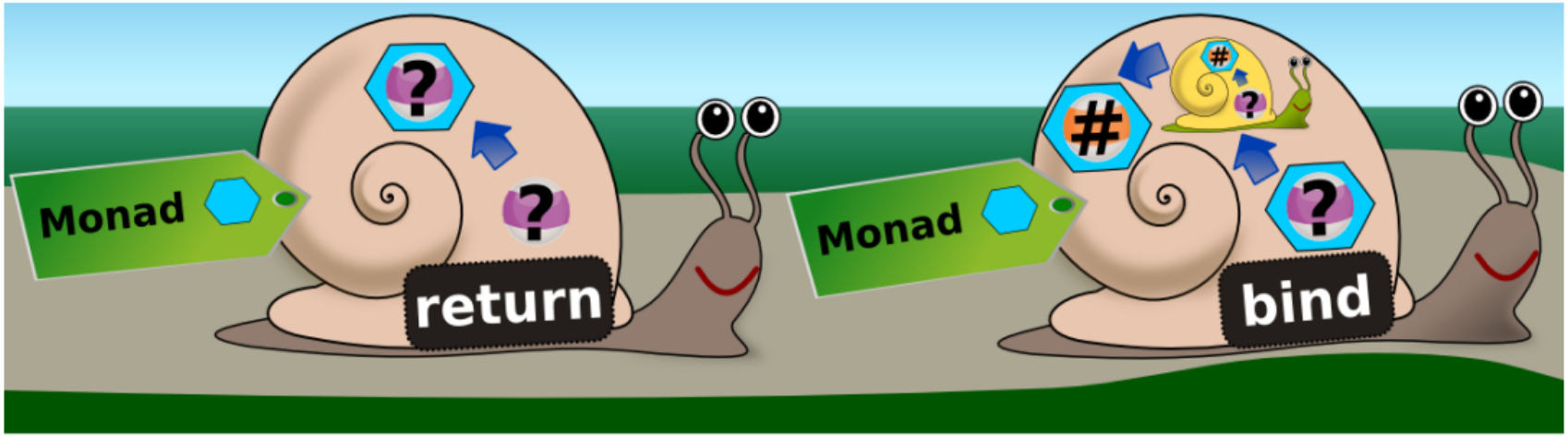
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
ghci> ["walk","nap","fight"] >>= (\t -> return ("cat" ++ t))
["catwalk","catnap","catfight"]
ghci> ["walk","nap","fight"] >>= (\t -> ["cat","dog"] >>= (\a -> return (a ++ t)))
```



```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
ghci> ["walk","nap","fight"] >>= (\t -> return ("cat" ++ t))
["catwalk","catnap","catfight"]
ghci> ["walk","nap","fight"] >>= (\t -> ["cat","dog"] >>= (\a -> return (a ++ t)))
```

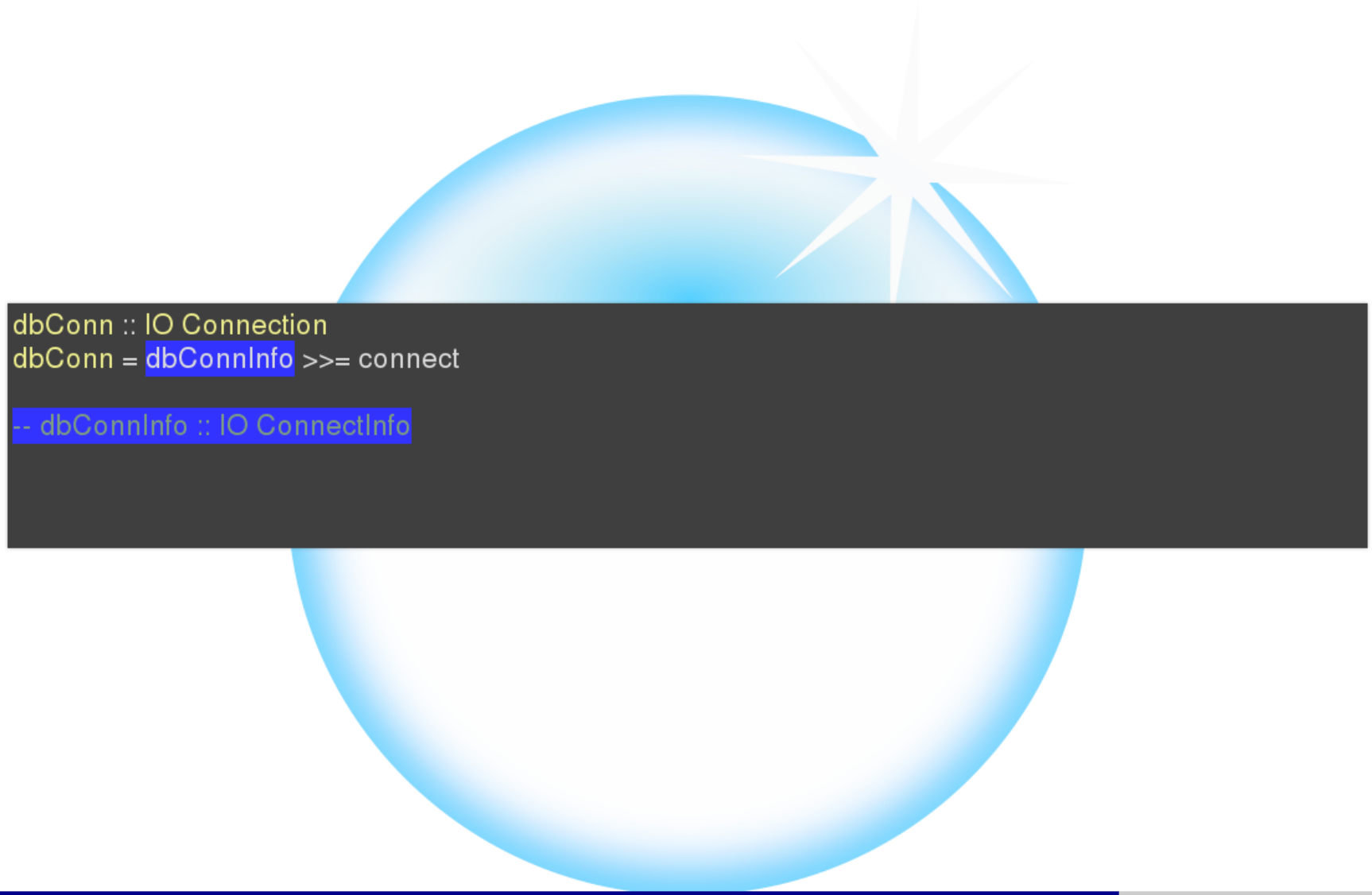


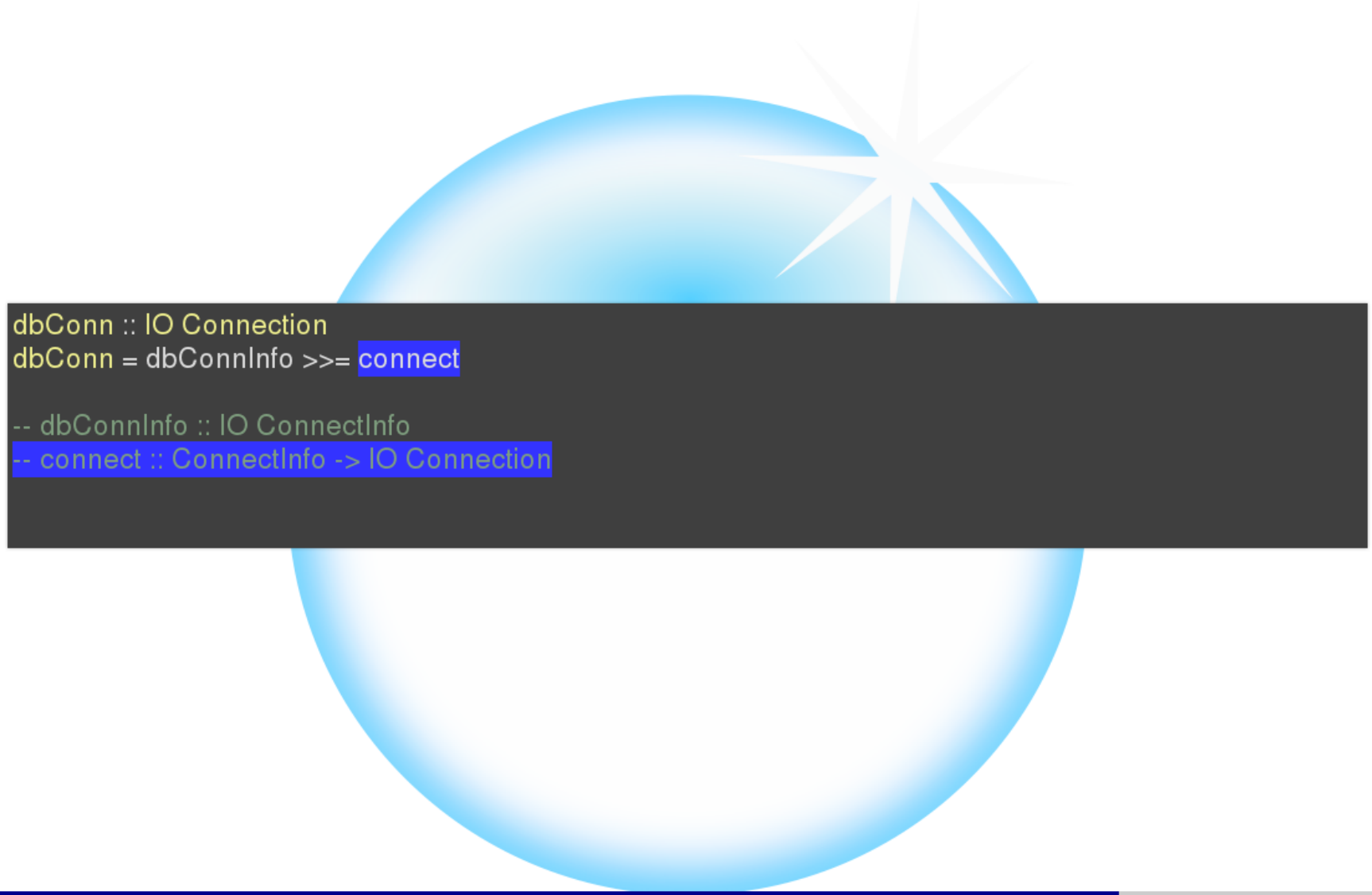
```
ghci> :info Monad
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
ghci> ["walk","nap","fight"] >>= (\t -> return ("cat" ++ t))
["catwalk","catnap","catfight"]
ghci> ["walk","nap","fight"] >>= (\t -> ["cat","dog"] >>= (\a -> return (a ++ t)))
["catwalk","dogwalk","catnap","dognap","catfight","dogfight"]
```

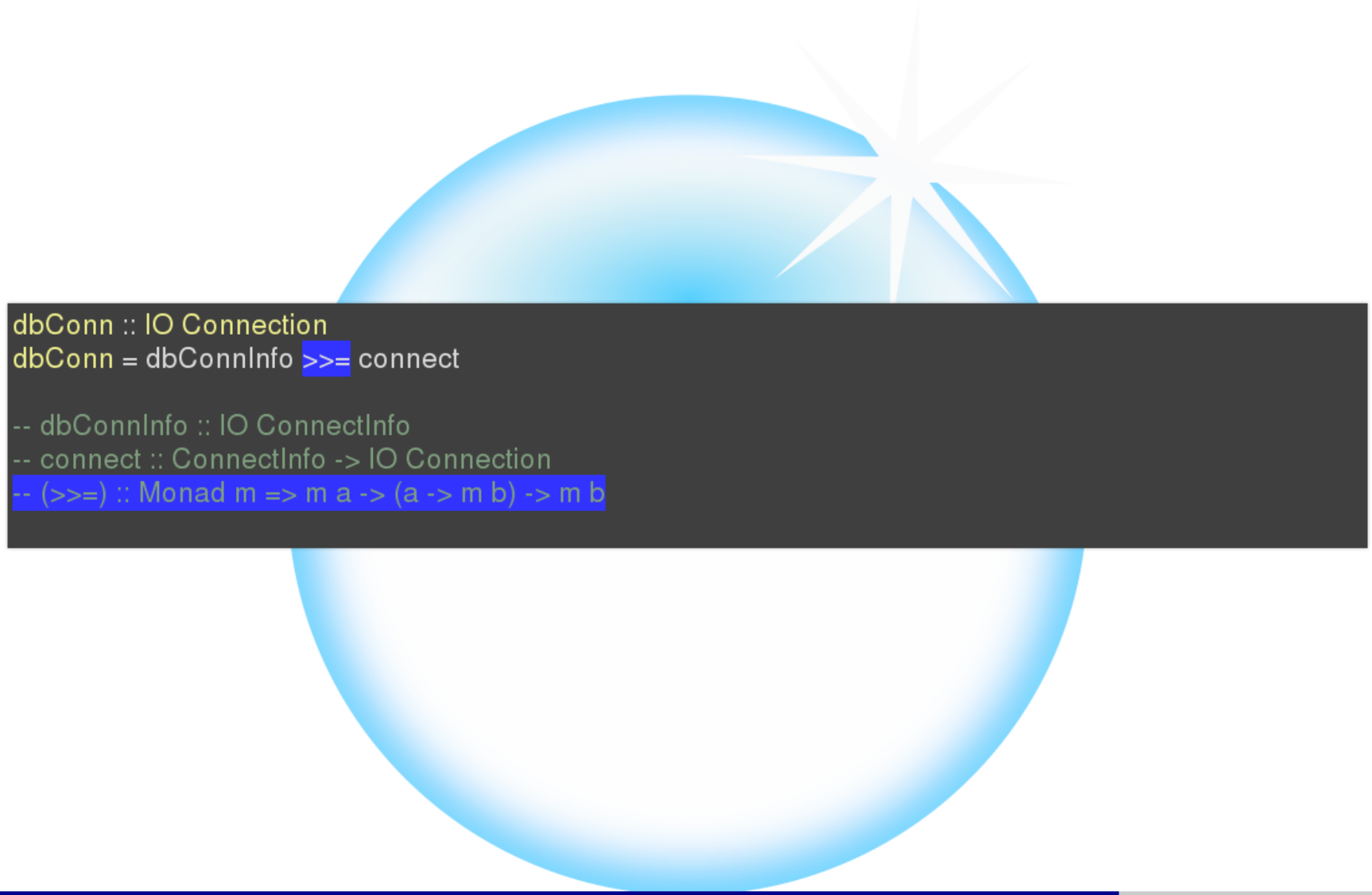


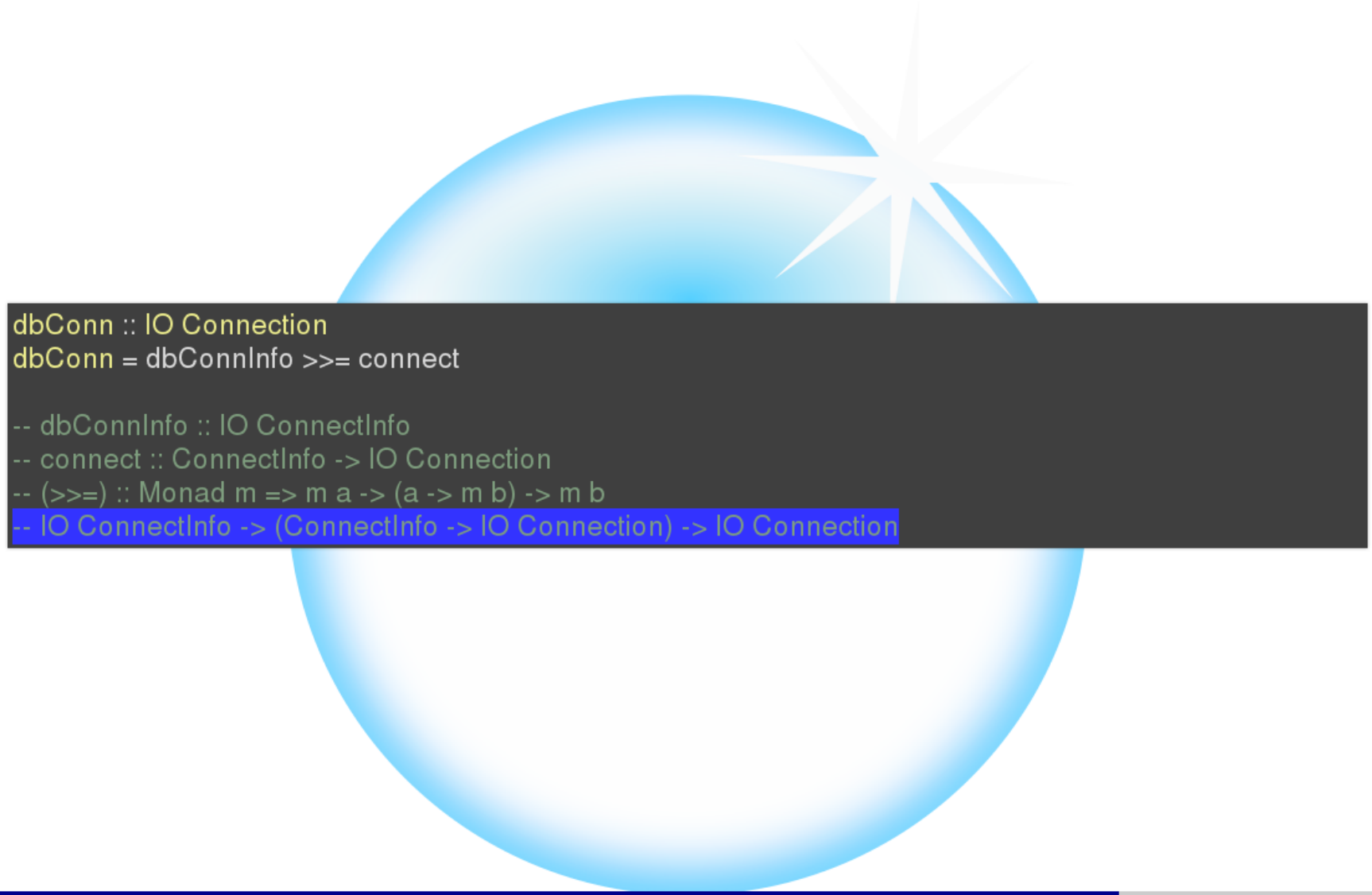

dbConn :: IO Connection
dbConn = dbConnInfo >>= connect

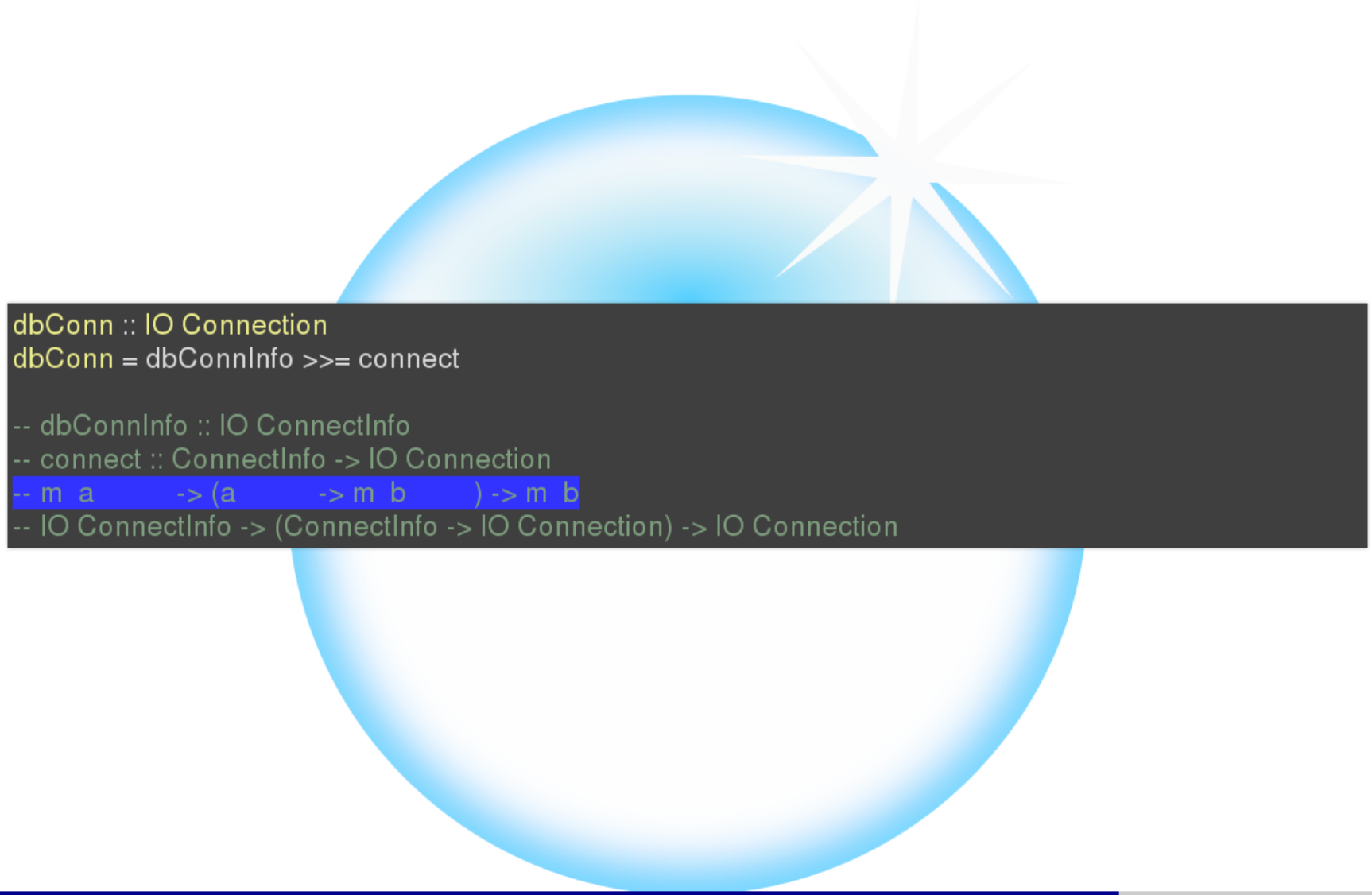
```
dbConn :: IO Connection
dbConn = dbConnInfo >>= connect
```

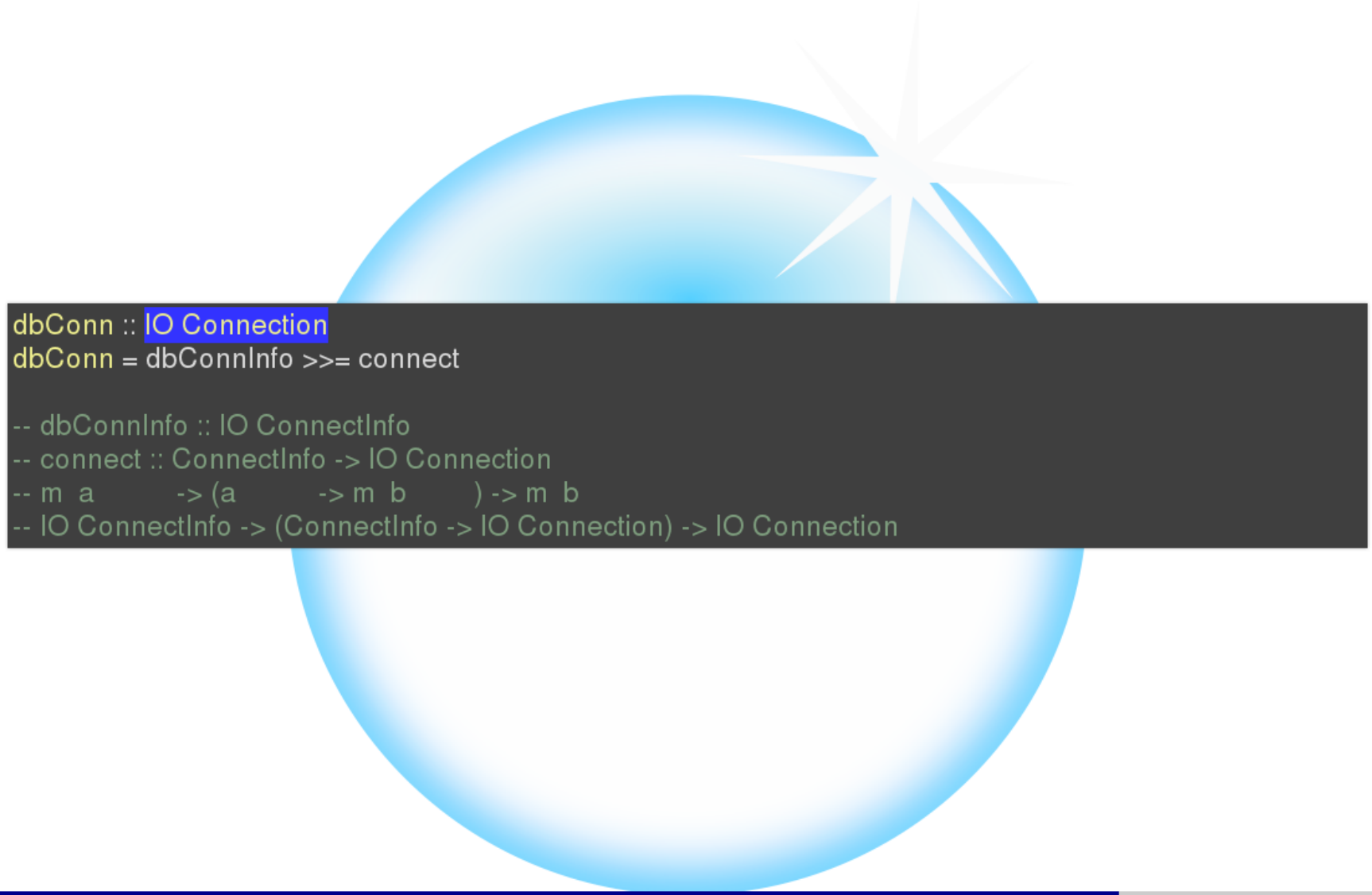

dbConn :: IO Connection
dbConn = dbConnInfo >>= connect
-- dbConnInfo :: IO ConnectInfo


dbConn :: IO Connection
dbConn = dbConnInfo >>= connect
-- dbConnInfo :: IO ConnectInfo
-- connect :: ConnectInfo -> IO Connection


dbConn :: IO Connection
dbConn = dbConnInfo >>= connect
-- dbConnInfo :: IO ConnectInfo
-- connect :: ConnectInfo -> IO Connection
-- (>>=) :: Monad m => m a -> (a -> m b) -> m b


dbConn :: IO Connection
dbConn = dbConnInfo >>= connect
-- dbConnInfo :: IO ConnectInfo
-- connect :: ConnectInfo -> IO Connection
-- (>>=) :: Monad m => m a -> (a -> m b) -> m b
-- IO ConnectInfo -> (ConnectInfo -> IO Connection) -> IO Connection


dbConn :: IO Connection
dbConn = dbConnInfo >>= connect
-- dbConnInfo :: IO ConnectInfo
-- connect :: ConnectInfo -> IO Connection
-- m a -> (a -> m b) -> m b
-- IO ConnectInfo -> (ConnectInfo -> IO Connection) -> IO Connection


dbConn :: IO Connection
dbConn = dbConnInfo >>= connect
-- dbConnInfo :: IO ConnectInfo
-- connect :: ConnectInfo -> IO Connection
-- m a -> (a -> m b) -> m b
-- IO ConnectInfo -> (ConnectInfo -> IO Connection) -> IO Connection

Lens

WHAT: Data structure for accessing and mutating values in a data type

WHY: Provides concise, functional way of drilling down into data types to get or set

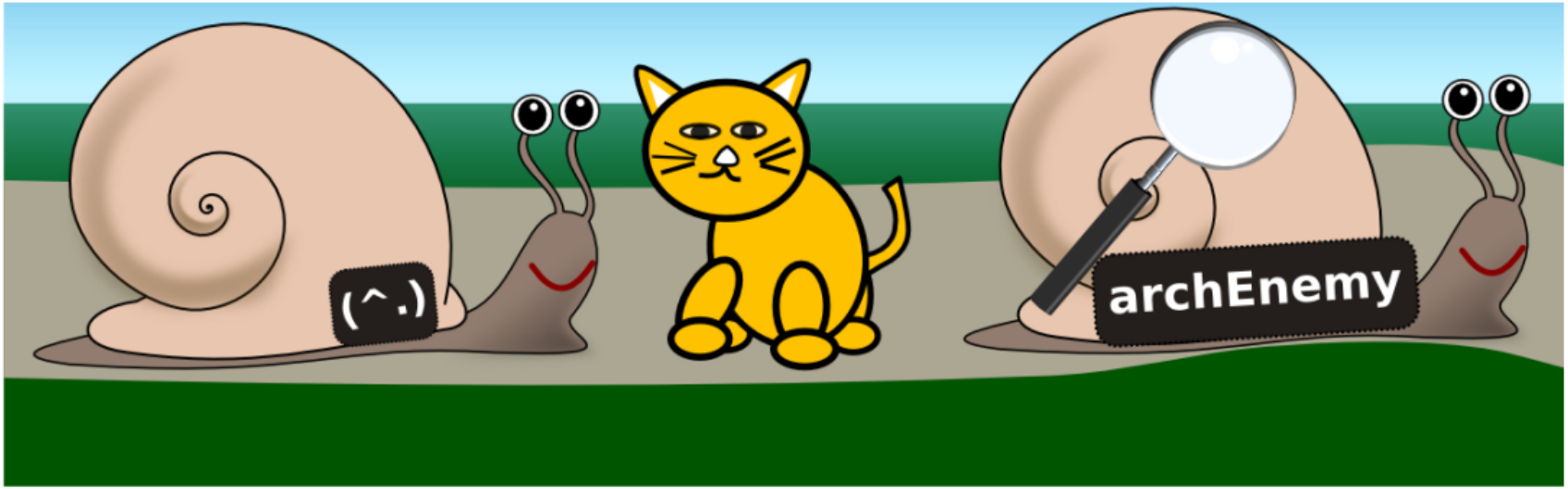
```haskell
-- data Lens target field =
--     Lens {
--         get :: target -> field,
--         set :: target -> field -> target
--     }
```

```
-- data Lens target field =
--     Lens {
--         get :: target -> field,
--         set :: target -> field -> target
--     }
ghci> :info Lens
type Lens s t a b = Functor f => (a -> f b) -> s -> f t
...
```

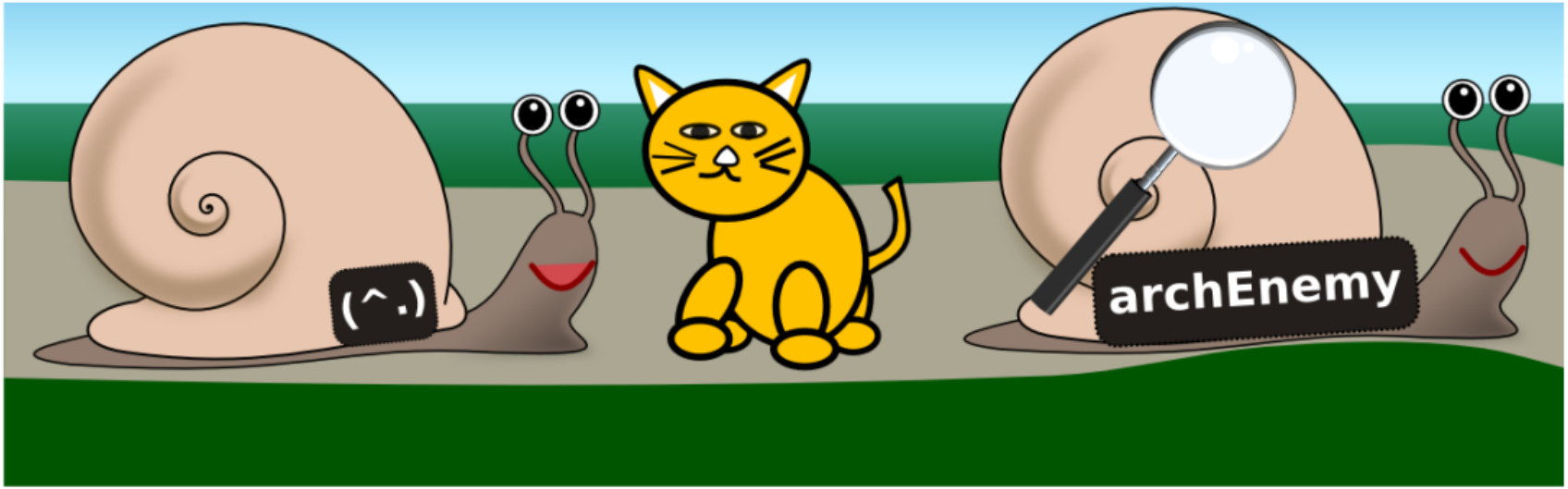
```
-- dog = Dog { _name = Name { _firstName = "Scooby", _lastName = "Da" } }
-- cat = Cat { _archEnemy = dog }
-- mouse = Mouse { _food = "cheddar" }
```

```
-- dog = Dog { _name = Name { _firstName = "Scooby", _lastName = "Da" } }
-- cat = Cat { _archEnemy = dog }
-- mouse = Mouse { _food = "cheddar" }

ghci> cat^.archEnemy^.name^.firstName
```

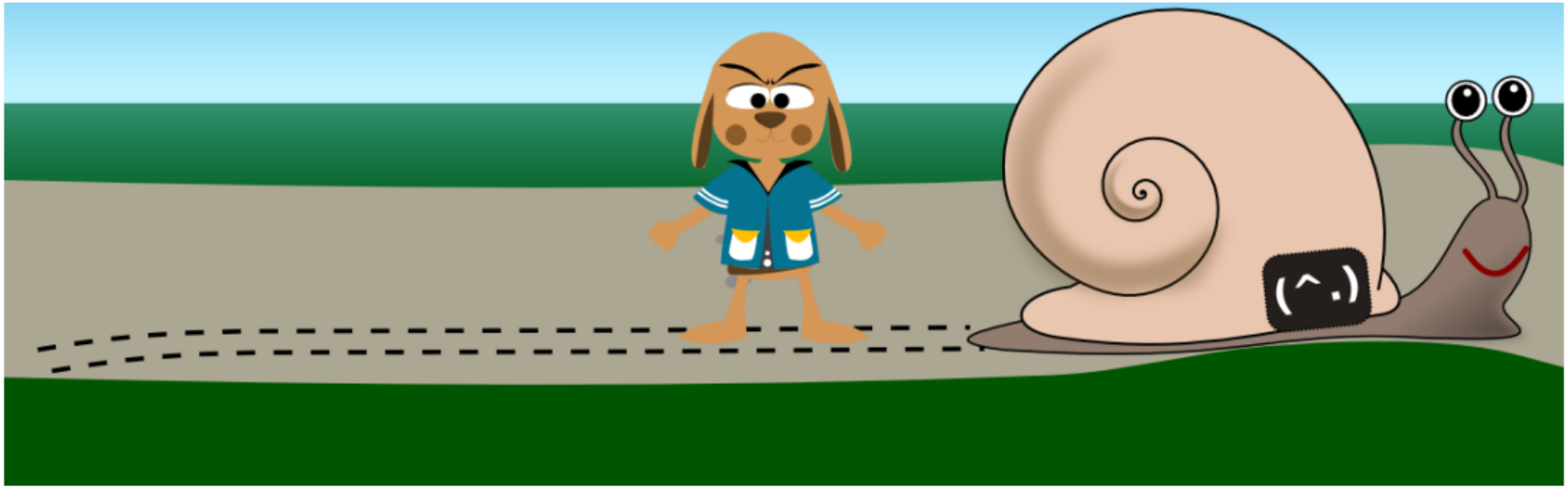
```
-- dog = Dog { _name = Name { _firstName = "Scooby", _lastName = "Da" } }
-- cat = Cat { _archEnemy = dog }
-- mouse = Mouse { _food = "cheddar" }

ghci> cat^.archEnemy^.name^.firstName
```



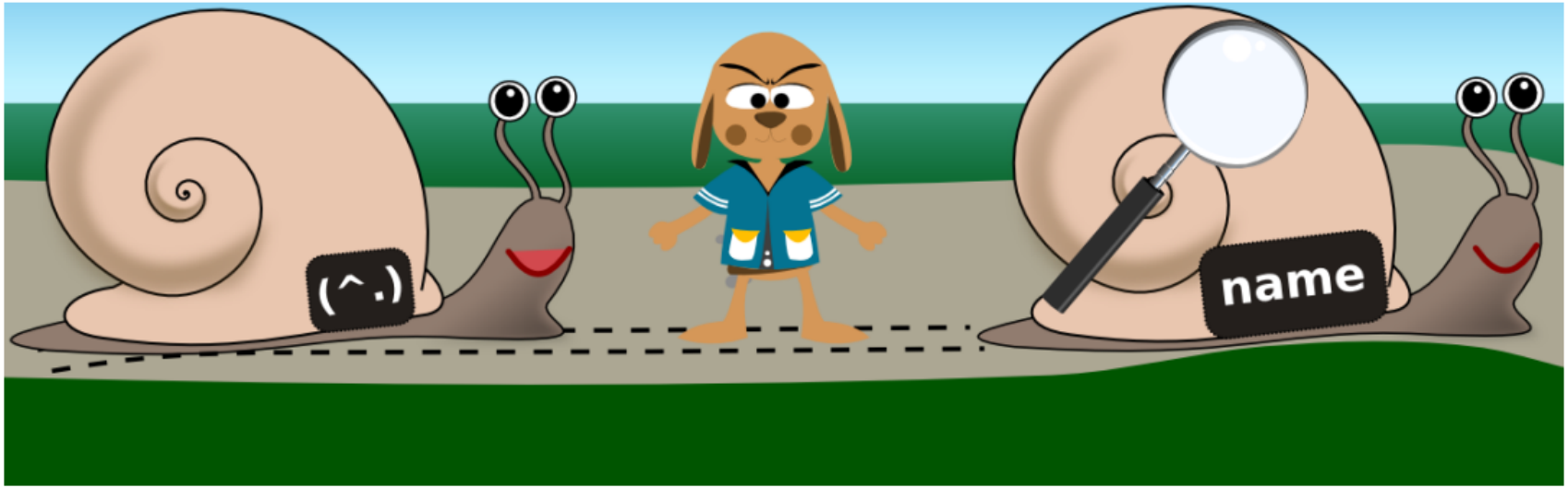
```
-- dog = Dog { _name = Name { _firstName = "Scooby", _lastName = "Da" } }
-- cat = Cat { _archEnemy = dog }
-- mouse = Mouse { _food = "cheddar" }

ghci> cat^.archEnemy^.name^.firstName
```



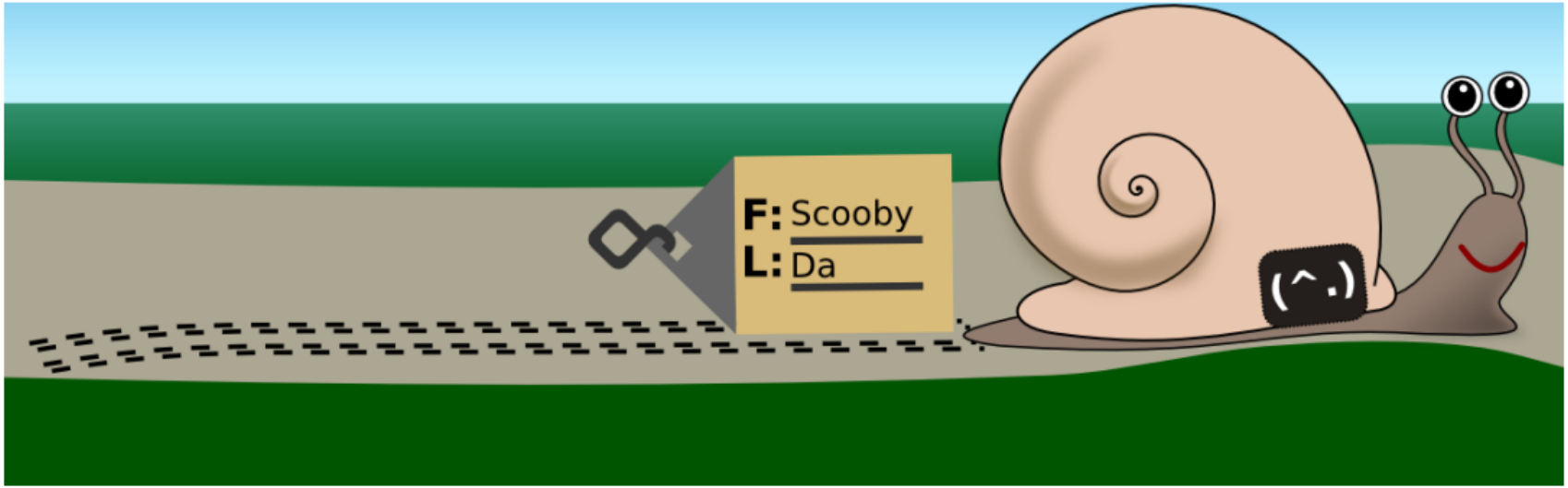
```
-- dog = Dog { _name = Name { _firstName = "Scooby", _lastName = "Da" } }
-- cat = Cat { _archEnemy = dog }
-- mouse = Mouse { _food = "cheddar" }

ghci> cat^.archEnemy^.name^.firstName
```



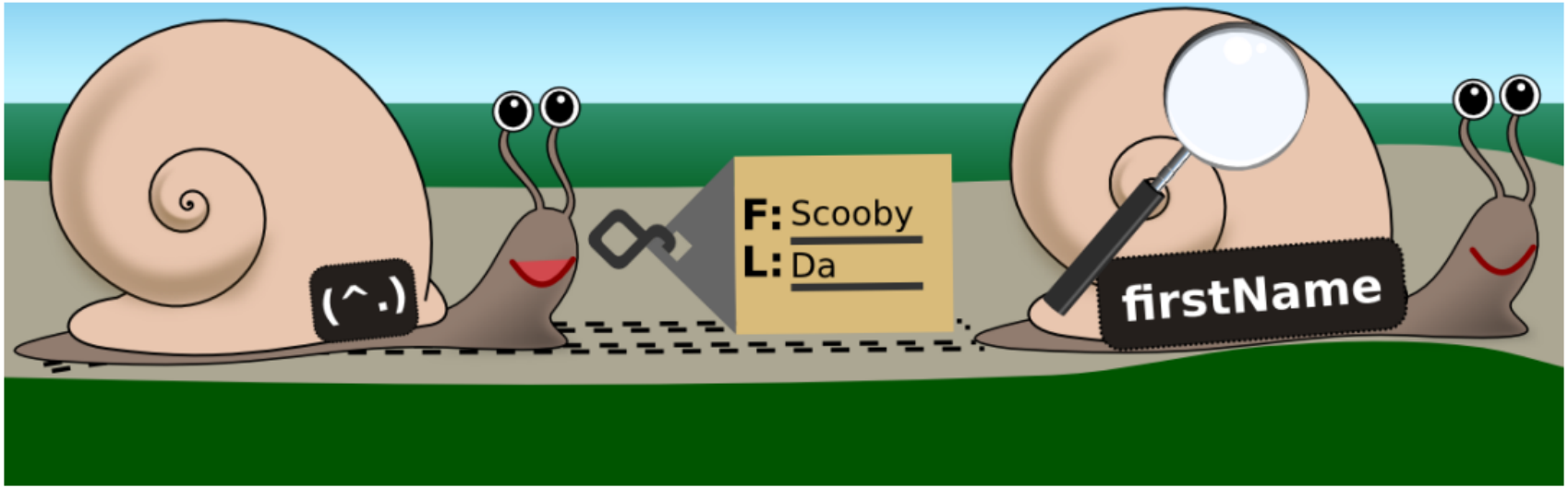
```
-- dog = Dog { _name = Name { _firstName = "Scooby", _lastName = "Da" } }
-- cat = Cat { _archEnemy = dog }
-- mouse = Mouse { _food = "cheddar" }

ghci> cat^.archEnemy^.name^.firstName
```



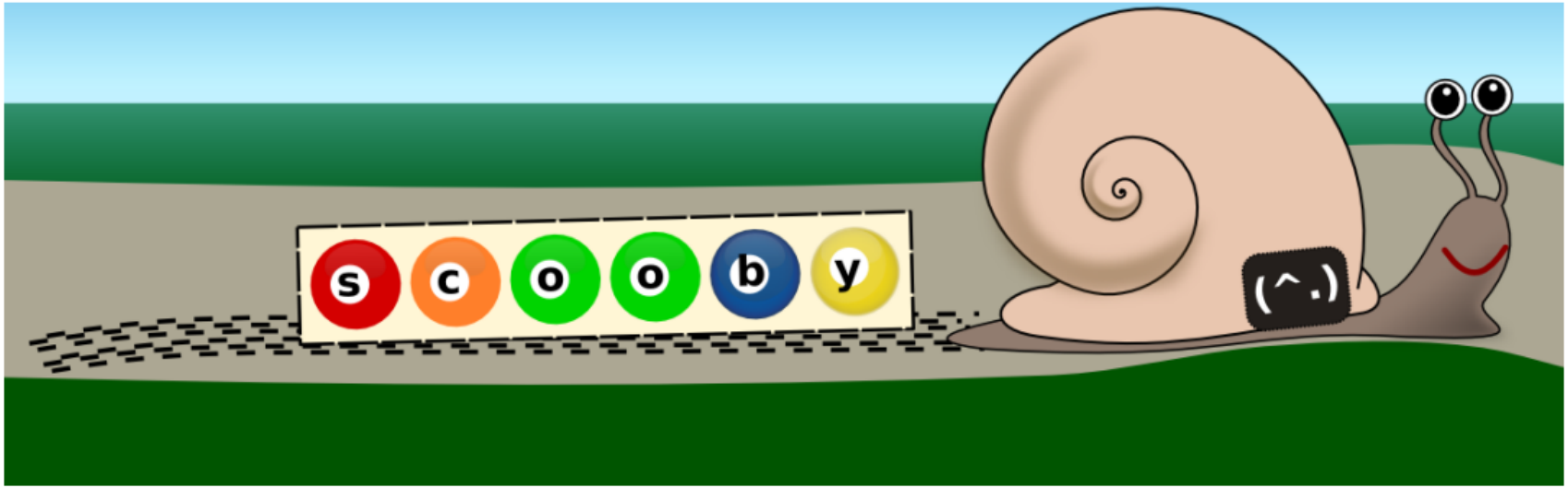
```
-- dog = Dog { _name = Name { _firstName = "Scooby", _lastName = "Da" } }
-- cat = Cat { _archEnemy = dog }
-- mouse = Mouse { _food = "cheddar" }

ghci> cat^.archEnemy^.name^.firstName
```



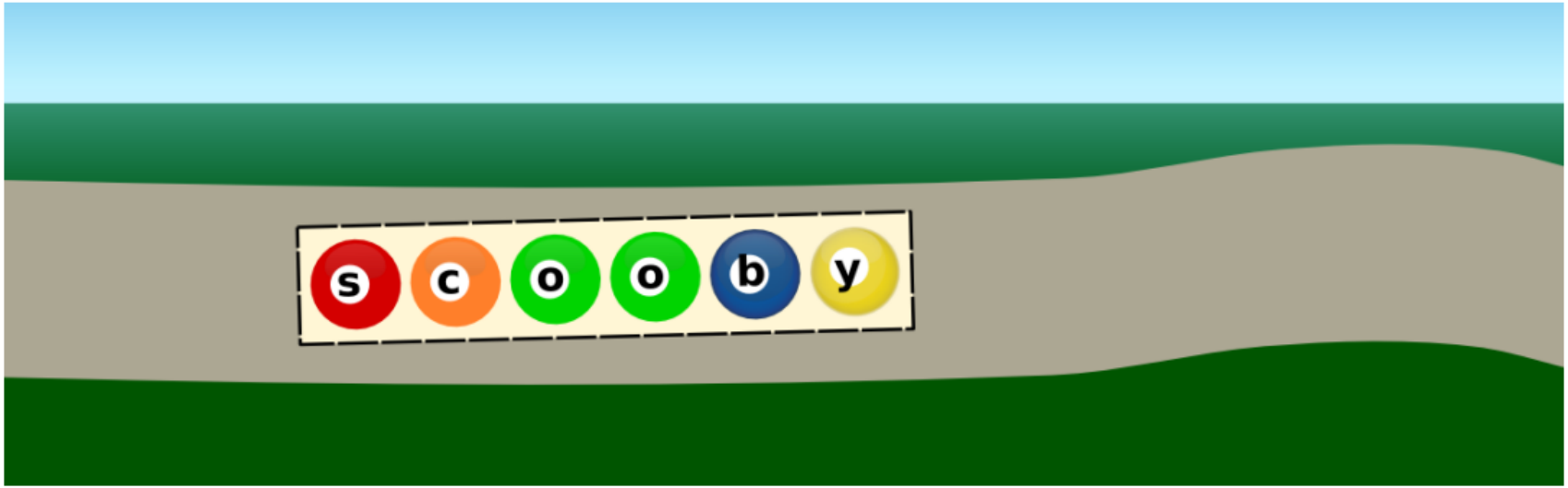
```
-- dog = Dog { _name = Name { _firstName = "Scooby", _lastName = "Da" } }
-- cat = Cat { _archEnemy = dog }
-- mouse = Mouse { _food = "cheddar" }

ghci> cat^.archEnemy^.name^.firstName
```



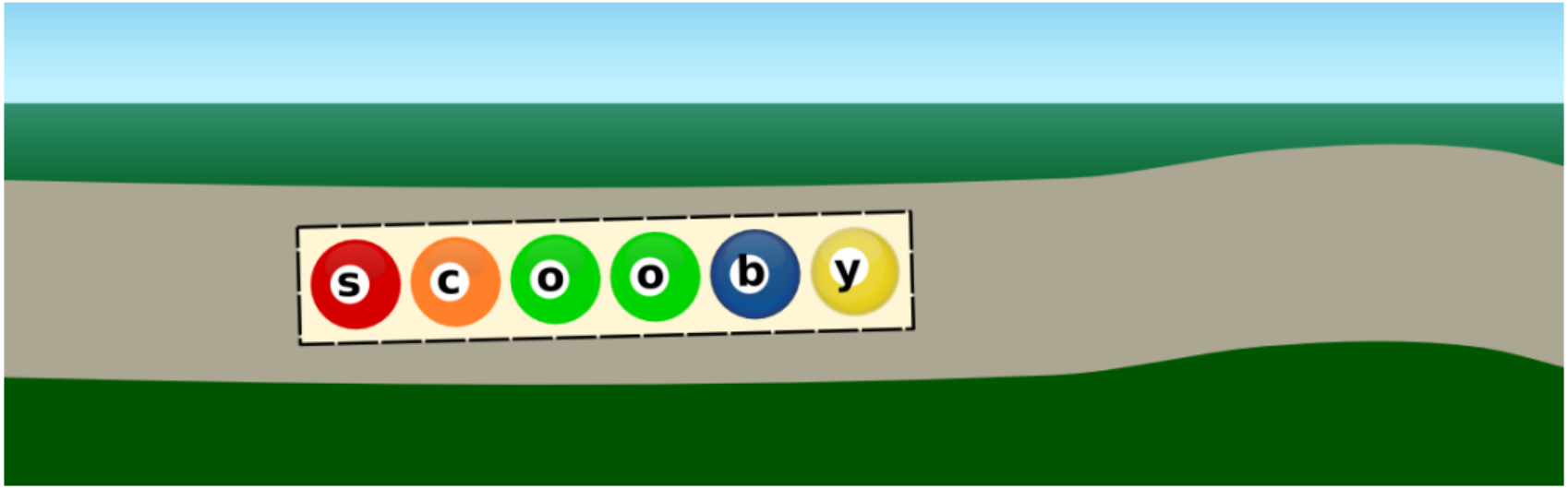
```
-- dog = Dog { _name = Name { _firstName = "Scooby", _lastName = "Da" } }
-- cat = Cat { _archEnemy = dog }
-- mouse = Mouse { _food = "cheddar" }

ghci> cat^.archEnemy^.name^.firstName
"Scooby"
```



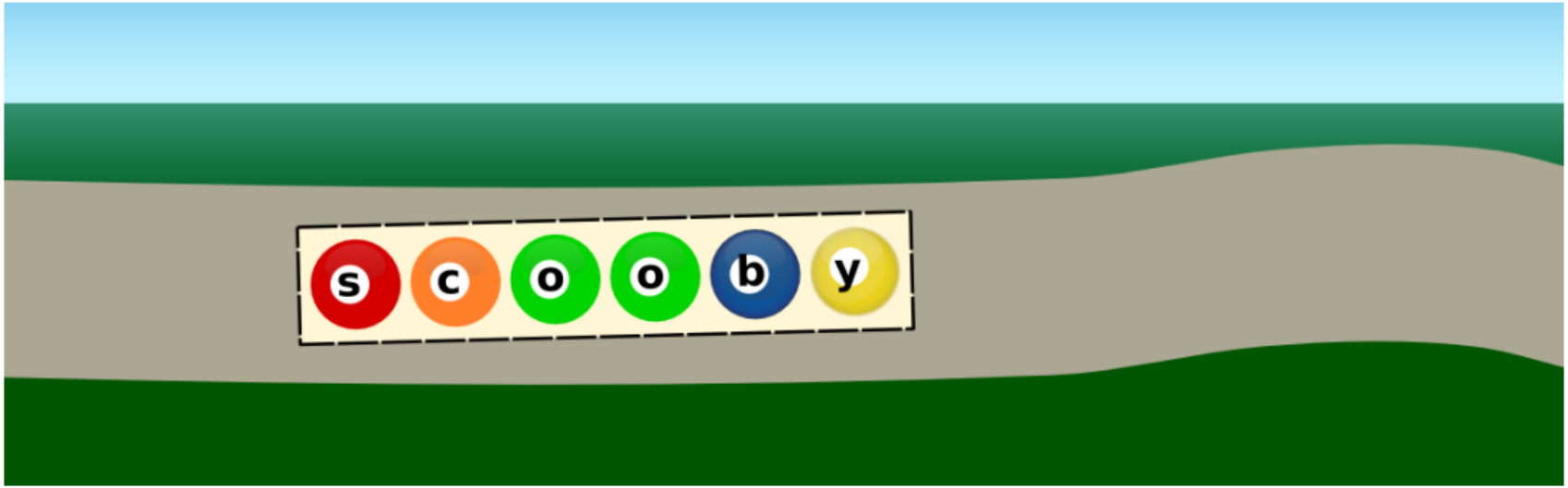
```
-- dog = Dog { _name = Name { _firstName = "Scooby", _lastName = "Da" } }
-- cat = Cat { _archEnemy = dog }
-- mouse = Mouse { _food = "cheddar" }

ghci> cat^.archEnemy^.name^.firstName
"Scooby"
```

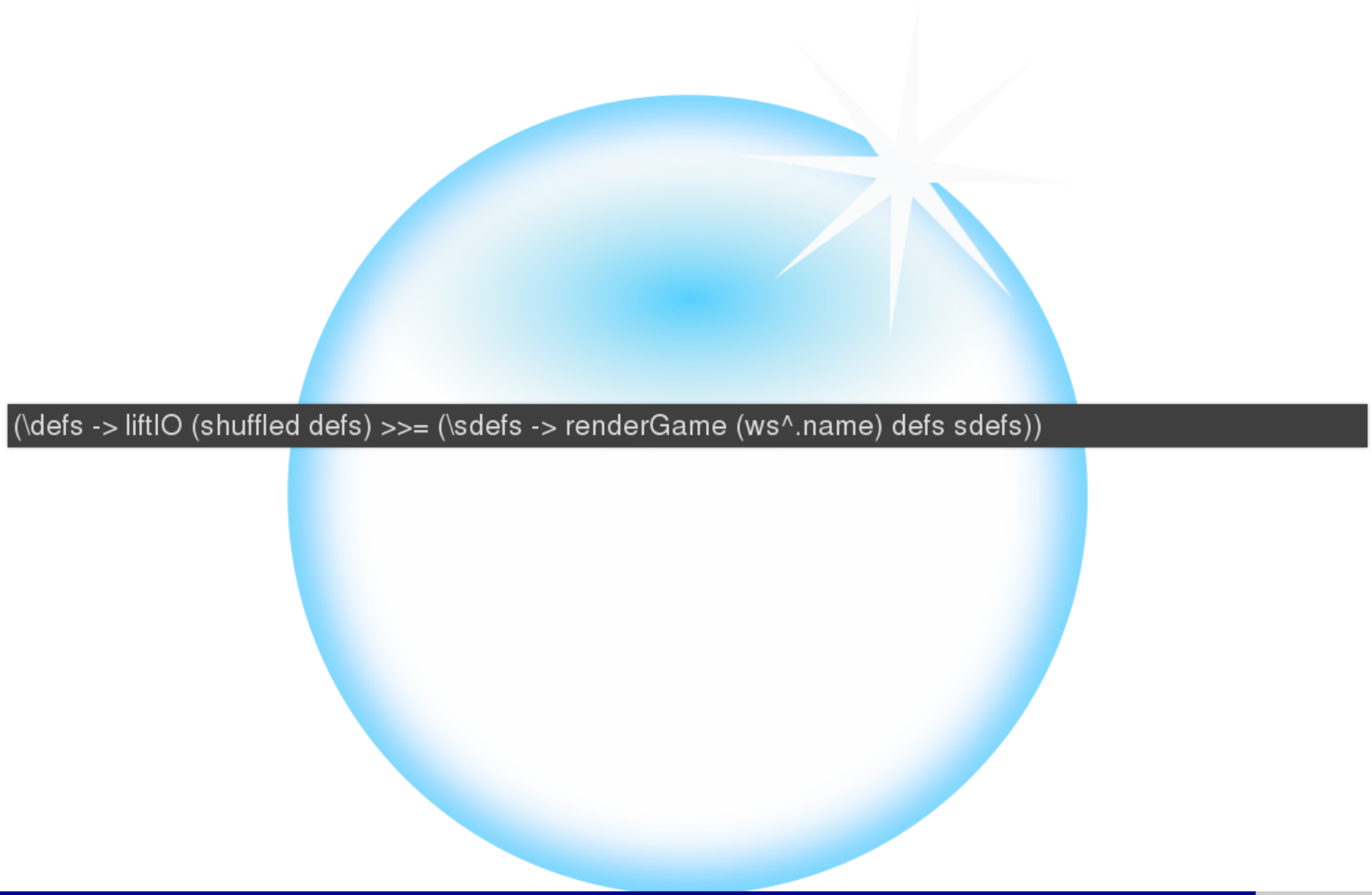


```
-- dog = Dog { _name = Name { _firstName = "Scooby", _lastName = "Da" } }
-- cat = Cat { _archEnemy = dog }
-- mouse = Mouse { _food = "cheddar" }

ghci> cat^.archEnemy^.name^.firstName
"Scooby"
ghci> set archEnemy mouse cat
```



```
-- dog = Dog { _name = Name { _firstName = "Scooby", _lastName = "Da" } }
-- cat = Cat { _archEnemy = dog }
-- mouse = Mouse { _food = "cheddar" }

ghci> cat^.archEnemy^.name^.firstName
"Scooby"
ghci> set archEnemy mouse cat
Cat {_archEnemy = Mouse {_food = "cheddar"}}
```




(\defs -> liftIO (shuffled defs) >>= (\sdefs -> renderGame (ws^.name) defs sdefs))


(\defs -> liftIO (shuffled defs) >>= (\sdefs -> renderGame (ws^.name) defs sdefs))


Conclusion

References, Resources, and Credits

References and Resources

- Abrahamson, J., A Little Lens Starter Tutorial
- Functional Talks
- Gonzalez, G., The Functor Design Pattern
- Haskell Learning Path
- Haskell Wiki on Functional Programming
- Hughes, J., Why Functional Programming Matters
- Jones, M. P., Functional Programming with Overloading and Higher-Order Polymorphism
- Kmett, E., Lenses, Folds, and Traversals
- Kmett, E., The Lens Package
- Lipovača, M., Learn You a Haskell For Great Good
- Meijer, E., Functional Programming Fundamentals Series
- Miller, K., Monads to the Rescue
- Morris, T., A Modern History of Lenses
- Morris, T., Comonads, Applicative Functors, Monads and Other Principled Things
- Morris, T. & Hibberd, M., NICTA Functional Programming Course
- O'Sullivan, B., Stewart, D., & Goerzen, J., Real World Haskell
- Wubble Source Code
- Yorgey, B., Introduction to Haskell
- Yorgey, B., Typeclassopedia

Image Credits

- Book Snake [cropped] (Alan Levine, CC BY-SA 2.0)
- Matrix (El-Sobreviviente, CC BY 3.0)
- Old Idea [reworked] (Fran Para Carrion, CC BY-SA 2.0)
- Open ClipArt

Coder Decoder

http://decoder.codemiller.com

Katie Miller (@codemiller)
OpenShift Developer Advocate at Red Hat