Quiz

Ivan Trepakov

NSU Sys.Pro

Quiz 1

Evaluate expression

```haskell
succ (2 - pred 1)

drop 2 (take 4 "Haskell") > map succ "cat"

sum (map fromEnum (enumFrom False))

map (\x -> x * 2) [1,3..10] ++ [100,1000]
```

Guess type signature

```haskell
max "Haskell"

map fst

map (take 2)
```

Quiz 2

Guess type signature

```
(++) [True]
```

```
zip [0..]
```

```
map filter
```

Guess type signature

```
alph = 'a' : alph
```

```
foo = zipWith (:)
```

```
f = f f
```

```
y g = g (y g)
```

Quiz 3

Guess type signature

```
z x y = zip x (concat y)

concatMap f x = concat (map f x)

f = 0 : 1 : zipWith (+) f (tail f)
```

Guess the function(s)

```
_ :: a -> a

_ :: a -> b

_ :: a -> [a] -> [a]

_ :: [a] -> Maybe (a, [a])
```

Quiz 4

Guess the function(s)

```
_ :: a -> b -> a

_ :: (a -> b -> c) -> b -> a -> c

_ :: ((a, b) -> c) -> a -> b -> c

_ :: (a -> b -> c) -> (a, b) -> c
```

Guess the function(s)

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

_ :: (b -> c) -> (a -> b) -> a -> c

_ :: (b -> a -> b) -> b -> [a] -> b

_ :: (a -> b -> b) -> b -> [a] -> b
```

Quiz 5

Evaluate expression

```
(2^) . (3+) $ 4

map ($2) [(*2), (^3), (1+)]

(++ "!") . reverse $ "abc"
```

Guess type signature

```
flip const

const undefined

foldr (:) []

((filter even .) .)
```

Quiz 6

Guess type signature

```
filter (const True)

foldl' (flip (:)) []

map (,)
```

How many distinguishable *total* functions?

```
_ :: a -> a

_ :: (a, a) -> (a, a)

_ :: a -> a -> Bool

_ :: Eq a => a -> a -> Bool
```

Quiz 7

How many distinguishable *total* functions?

```
_ :: a -> b -> a

_ :: a -> a -> a

_ :: (b -> c) -> (a -> b) -> (a -> c)
```

How many distinguishable *total* functions?

```
_ :: a -> Maybe a

_ :: a -> Maybe b

_ :: (a -> Maybe b) -> [a] -> [b]
```

Quiz 8

How many distinguishable *total* functions?

```
_ :: Bool -> Bool -> Bool

_ :: Bool -> a -> a

_ :: [a] -> a
```

Guess the function(s)

```
_ :: (a -> a -> Bool) -> [a] -> [[a]]

_ :: (a -> a -> Ordering) -> [a] -> [a]

_ :: Ord a => (b -> a) -> b -> b -> Ordering
```

Quiz 9

Evaluate expression

```
foldr (++) "S" ["foo", "bar"]

foldl (++) "S" ["foo", "bar"]

foldr ($) "S" [(++ "foo"), (++ "bar")]

mconcat [(++ "foo"), (++ "bar")] "S"
```

Guess the Semigroup(s)

```
instance Semigroup (Maybe a) where
  (<>) :: Maybe a -> Maybe a -> Maybe a

newtype X a = X a
instance Semigroup a => Semigroup (X a) where
  (<>) :: X a -> X a -> X a

newtype Y a = Y a
instance Semigroup (Y a) where
  (<>) :: Y a -> Y a -> Y a
```

Quiz 10

Guess the kind

```haskell
data A a b = A a b

data B a b = B (a b)

data C a b = C (b (a b))

data D a b = D (a (b a))
```

Guess foldMap Monoid

```haskell
product :: (Foldable t, Num a) =>
                          t a -> a


any :: Foldable t =>
            (a -> Bool) -> t a -> Bool


elem :: (Foldable t, Eq a) =>
                      a -> t a -> Bool
```

Quiz 11

Guess foldMap Monoid

```haskell
find :: Foldable t =>
            (a -> Bool) -> t a -> Maybe a

safeMaximum :: (Foldable t, Ord a) =>
                          t a -> Maybe a

safeMaximumBy :: Foldable t =>
    (a -> a -> Ordering) -> t a -> Maybe a
```

Guess type signature

```haskell
map (Just .) [even, odd]

map (const) [even, odd]

map (const .) [even, odd]
```

Quiz 12

Guess the function(s)

```
_ :: Foldable t => (a -> [b]) -> t a -> [b]

_ :: Foldable t => t Bool -> Bool

_ :: Foldable t => (a -> Bool) -> t a -> [a]
```

Guess type signature

```
foldMap . foldMap

foldMap . (flip foldMap)

foldMap . map
```

Quiz 13

Guess type signature

```
liftA2 (==)

liftA2 (take)

liftA2 (.)
```

Guess type signature

```
liftA2 ($)

liftA2 (fmap)

liftA2 (liftA2)
```

Quiz 14

Guess type signature

```
liftA

pure . pure

pure pure
```

Guess the function(s)

```
_ :: Functor f => a -> f b -> f a

_ :: Applicative f =>
         f a -> f (a -> b) -> f b

_ :: Applicative f =>
              Bool -> f () -> f ()
```

Quiz 15

Evaluate expression

```
traverse Just [1..5]

sequenceA $ sequenceA
  [Just "hello", Just "world!"]

sequenceA_ $ sequenceA
  [Just "hello", Just "world!"]
```

Evaluate expression

```
for_ [1..5] Just

for (Just "foo") $ const [1..5]

for [1..3] (flip take [1..])
```

Q&A