In [1]:

```
{-# LANGUAGE OverloadedStrings #-}

import Network.Wreq
import Data.Aeson.Lens
import Control.Lens
```

In [7]:

```
r <- get "https://api.github.com/search/repositories?q=language:haskell&sort=stars"
```

In [9]:

```
print5 = mapM_ print . take 5

print5 $ r ^.. responseBody . key "items" . values . key "full_name" . _String
```

```
"jgm/pandoc"
"begriffs/postgrest"
"koalaman/shellcheck"
"purescript/purescript"
"elm-lang/elm-compiler"
```

In [10]:

```
import qualified Data.Text.IO as T

geonamesUsername <- T.readFile ".geonames-username"
```

In [11]:

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

import Data.String (IsString)
import qualified Data.Text as T

newtype GithubRepo = GithubRepo { unGithubRepo :: T.Text } deriving (IsString, Show
newtype GithubUser = GithubUser { unGithubUser :: T.Text } deriving (IsString, Show
newtype CountryCode = CountryCode { unCountryCode :: T.Text } deriving (IsString, S
newtype CountryName = CountryName { unCountryName :: T.Text } deriving (IsString, S
newtype CountryPopulation = CountryPopulation { unCountryPopulation :: Int }
    deriving (Num, Enum, Eq, Ord, Real, Integral, Show)

data Country = Country
    { countryName :: CountryName
    , countryPopulation :: CountryPopulation
    } deriving Show
```

In [12]:

```
import Data.Semigroup

githubAPI :: String
geonamesAPI :: String

githubAPI = "https://api.github.com"
geonamesAPI = "http://api.geonames.org"

githubAPISearchRepos :: String
githubAPIRepos :: GithubRepo -> String
githubAPIUsers :: GithubUser -> String
githubAPIRepoContributors :: GithubRepo -> String
geonamesAPISearch :: String
geonamesAPICountryInfo :: String

githubAPISearchRepos = githubAPI <> "/search/repositories"
githubAPIRepos (GithubRepo repo) = githubAPI <> "/repos/" <> T.unpack repo
githubAPIUsers (GithubUser user) = githubAPI <> "/users/" <> T.unpack user
githubAPIRepoContributors repo = githubAPIRepos repo <> "/contributors"
geonamesAPISearch = geonamesAPI <> "/searchJSON"
geonamesAPICountryInfo = geonamesAPI <> "/countryInfoJSON"
```

In [13]:

```
import Control.Monad

geonamesError r = r ^? responseBody . key "status" . to show

findCountryCode :: T.Text -> IO (Maybe CountryCode)
findCountryCode location = do
    r <- getWith opts geonamesAPISearch
    forM_ (geonamesError r) $ \e ->
        fail ("At " <> T.unpack location <> ": " <> e)
    pure $ CountryCode <$>
        r ^? responseBody
            . key "geonames"
            . nth 0
            . key "countryCode"
            . _String
  where
    opts =
        defaults
            & param "q" .~ [location]
            & param "username" .~ [geonamesUsername]
```

In [15]:

```
findCountryCode "Schweiz"
```

JustCountryCode {unCountryCode = "CH"}

In [16]:

```
findCountryCode "England"
```

JustCountryCode {unCountryCode = "GB"}

In [14]:

```
:i Country
```

In [18]:

```
countryByCountryCode :: CountryCode -> IO (Maybe Country)
countryByCountryCode (CountryCode code) = do
    r <- getWith opts geonamesAPICountryInfo
    forM_ (geonamesError r) $ \e ->
        fail ("At " <> T.unpack code <> ": " <> e)
    pure $ Country
        <$> (fmap CountryName $ r
                ^? responseBody
                . key "geonames"
                . nth 0
                . key "countryName"
                . _String)
        <*> (fmap CountryPopulation $ r
                ^? responseBody
                . key "geonames"
                . nth 0
                . key "population"
                . _String
                . to (read . T.unpack))
  where
    opts =
      defaults
          & param "country" .~ [ code ]
          & param "username" .~ [ geonamesUsername ]
```

In [19]:

```
countryByCountryCode "CH"
```

JustCountry {countryName = CountryName {unCountryName = "Switzerland"}, countryPopulation = CountryPopulation {unCountryPopulation = 7581000}}

In [20]:

```
import qualified Data.ByteString as BS

githubAuth <- oauth2Token <$> BS.readFile ".github-api-token"
```

In [21]:

```
import Control.Monad.Trans.Maybe

githubUserCountry :: GithubUser -> IO (Maybe Country)
githubUserCountry user = runMaybeT $ do
    location <- MaybeT $ getLocation <$> fetchUser
    code <- MaybeT $ findCountryCode location
    MaybeT $ countryByCountryCode code
  where
    fetchUser = getWith opts $ githubAPIUsers user
    getLocation r = r ^? responseBody . key "location" . _String
    opts = defaults & auth ?~ githubAuth
```

In [24]:

```
githubUserCountry "aherrmann"
```

Nothing

In [25]:

```
{-# LANGUAGE RankNTypes #-}

import Data.Conduit
import Data.Function (fix)
import Data.ByteString.Lens
import qualified Data.ByteString.Lazy as BL

import Control.Monad.IO.Class (liftIO)
import qualified Data.Conduit.Combinators as C

getAllWith :: Options -> String -> Producer IO (Response BL.ByteString)
getAllWith opts = fix $ \loop url -> do
      r <- liftIO (getWith opts url)
      yield r
      mapM_ loop $ r ^? responseLink "rel" "next" . linkURL . unpackedChars

topRepos :: T.Text -> Producer IO GithubRepo
topRepos language =
    getAllWith opts githubAPISearchRepos
    .| awaitForever (C.yieldMany . fmap GithubRepo . getFullName)
  where
    getFullName r =
      r ^.. responseBody . key "items" . values . key "full_name" . _String
    opts = defaults
        & param "q" .~ ["language:" <> language]
        & param "sort" .~ ["stars"]
        & param "per_page" .~ ["100"]
        & auth ?~ githubAuth
```

In [27]:

```
topRepos "haskell" $$ C.take 5 .| C.mapM_ print
```

```
GithubRepo {unGithubRepo = "jgm/pandoc"}
GithubRepo {unGithubRepo = "begriffs/postgrest"}
GithubRepo {unGithubRepo = "koalaman/shellcheck"}
GithubRepo {unGithubRepo = "purescript/purescript"}
GithubRepo {unGithubRepo = "elm-lang/elm-compiler"}
```

In [28]:

```
repoContributors :: GithubRepo -> Producer IO GithubUser
repoContributors repo =
    getAllWith opts (githubAPIRepoContributors repo)
    .| awaitForever (C.yieldMany . fmap GithubUser . getLogin)
  where
    getLogin r = r ^.. responseBody .values.key "login" . _String
    opts = defaults
        & param "per_page" .~ ["100"]
        & auth ?~ githubAuth
```

In [30]:

```
repoContributors "jgm/pandoc" $$ C.take 10 .| C.mapM_ print
```

```
GithubUser {unGithubUser = "jgm"}
GithubUser {unGithubUser = "jkr"}
GithubUser {unGithubUser = "tarleb"}
GithubUser {unGithubUser = "labdsf"}
GithubUser {unGithubUser = "mpickering"}
GithubUser {unGithubUser = "mb21"}
GithubUser {unGithubUser = "lierdakil"}
GithubUser {unGithubUser = "adunning"}
GithubUser {unGithubUser = "claremacrae"}
GithubUser {unGithubUser = "ickc"}
```

In [41]:

```
topRepos "haskell" $$ awaitForever repoContributors .| C.take 1 .| C.mapM_ print
```

```
GithubUser {unGithubUser = "jgm"}
```

In [33]:

```
{-# LANGUAGE LambdaCase #-}

import qualified Data.HashSet as Set
import Data.Hashable (Hashable(..))

accumulateUniques :: (Eq a, Hashable a) => Int -> Sink a IO (Set.HashSet a)
accumulateUniques n = go mempty
    where
      go acc = await >>= \case
          Just x | Set.size acc < n -> go (Set.insert x acc)
          _ -> pure acc
```

In [39]:

```
C.yieldMany ["foo", "baz", "baz", "bar", error "NO!"] $$ accumulateUniques 3
```

```
fromList ["baz","foo","bar"]
```

In [40]:

```
{-# LANGUAGE StandaloneDeriving #-}

deriving instance Eq GithubUser
deriving instance Hashable GithubUser
```

In [42]:

```
users <- topRepos "haskell" $$ awaitForever repoContributors .| accumulateUniques 5

Set.toList users
```

```
[GithubUser {unGithubUser = "jkr"},GithubUser {unGithubUser = "tarle
b"},GithubUser {unGithubUser = "jgm"},GithubUser {unGithubUser = "mpic
kering"},GithubUser {unGithubUser = "labdsf"}]
```

In [43]:

```
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE ViewPatterns #-}

import qualified Data.Aeson as Aeson
import GHC.Generics (Generic)
import qualified Data.Binary as B
import qualified Database.LevelDB as DB
import Data.Default
import Control.Monad.Trans.Resource

deriving instance Generic GithubUser
deriving instance Generic Country
deriving instance Generic CountryName
deriving instance Generic CountryPopulation

instance B.Binary GithubUser
instance B.Binary CountryName
instance B.Binary CountryPopulation
instance B.Binary Country

cached :: (B.Binary a, B.Binary b) => (a -> IO b) -> ((a -> ResourceT IO b) -> Reso
cached fetch act = runResourceT $ do
    db <- DB.open ".cache" def {DB.createIfMissing = True}
    act $ \key'@(BL.toStrict . B.encode -> key) -> do
        (fmap $ B.decode . BL.fromStrict) <$> DB.get db def key >>= \case
            Just x -> pure x
            Nothing -> do
                x <- liftIO $ fetch key'
                DB.put db def key (BL.toStrict $ B.encode x)
                pure x
```

In [45]:

```
:t githubUserCountry
```

```
githubUserCountry :: GithubUser -> IO (Maybe Country)
```

In [46]:

```
:i Country
```

In [44]:

```
import qualified Data.HashMap.Strict as Map
import Control.Monad

deriving instance Eq CountryPopulation
deriving instance Eq CountryName
deriving instance Eq Country
instance Hashable Country where hashWithSalt s = hashWithSalt s . unCountryName . c

userCountries :: Set.HashSet GithubUser -> IO (Map.HashMap Country Int)
userCountries cs = cached githubUserCountry $ \githubUserCountry' ->
  foldM
    (\m u -> githubUserCountry' u >>= \case
        Nothing -> pure m
        Just c -> pure $ Map.insertWith (+) c 1 m
    ) Map.empty (Set.toList cs)
```

In [47]:

```
import Graphics.Rendering.Chart
import Graphics.Rendering.Chart.Backend.Cairo
import Data.Default.Class
import Control.Lens

chart :: String -> [(Country, Double)] -> Renderable ()
chart title cs = toRenderable layout
  where
    layout = pie_title .~ title
           $ pie_plot . pie_data .~ map pitem values
           $ def
    values = take 10
           . (ix 0. _3 .~ 20)
           . fmap (\(c, val) -> (T.unpack $ unCountryName $ countryName c, val, 0))
            $ cs
    pitem (s,v,o) = pitem_value .~ v
                  $ pitem_label .~ s
                  $ pitem_offset .~ o
                  $ def
```

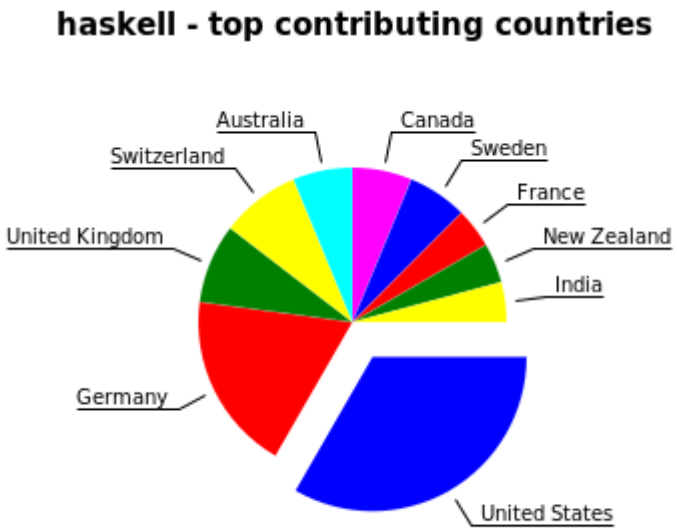
In [48]:

```
import Data.Bifunctor
import Data.List
```

In [49]:

```
let lang = "haskell"
    title = "top contributing countries"

users <- topRepos lang $$ awaitForever repoContributors .| accumulateUniques 100

countries <- userCountries users

putStrLn $ "Users: " <> show (Set.size users)
putStrLn $ "Countries: " <> show (Map.size countries)

chart (lang <> " - " <> title)
    . sortOn (negate . snd)
    . fmap (second fromIntegral)
    $ Map.toList countries
```

```
Users: 100

Countries: 22
```



In [51]:

```
let lang = "haskell"
    title = "top contributing countries"

users <- topRepos lang $$ awaitForever repoContributors .| accumulateUniques 3000

countries <- userCountries users

putStrLn $ "Users: " <> show (Set.size users)
putStrLn $ "Countries: " <> show (Map.size countries)

chart (lang <> " - " <> title)
    . sortOn (negate . snd)
    . fmap (\(c, n) -> (c, n / fromIntegral (countryPopulation c)))
    . filter  ((> 1000000) fromIntegral)

    . fmap (second fromIntegral)
    $ Map.toList countries
```

Functor law

Found:

```
fmap (\ (c, n) -> (c, n / fromIntegral (countryPopulation c))) .
  fmap (second fromIntegral)
```

Why Not:

```
fmap
  ((\ (c, n) -> (c, n / fromIntegral (countryPopulation c))) .
     second fromIntegral)
```

```
Users: 3000
```

```
Countries: 74
```

In [52]:

```
chart (lang <> " - " <> title)
    . sortOn (negate . snd)
    . fmap (\(c, n) -> (c, n / fromIntegral (countryPopulation c)))
    . filter  ((> 1000000) . countryPopulation . fst)

    . fmap (second fromIntegral)
    $ Map.toList countries
```