Haskell - Módulos

André Casagranda
André Zanghelini
Felipe Losso
Rafael Oliveira

8 de Junho, 2022

Sumário

- Motivação
- Principais Conceitos
- Sintaxe e Semântica
- Relação com Outras Linguagens
- Exemplos
- Conclusões

Organização

- Agrupamento de funções relacionadas;
- Gerenciar funções que possam ter o mesmo nome;
- Criar tipos abstratos de dados;

Organização

- Agrupamento de funções relacionadas;
- Gerenciar funções que possam ter o mesmo nome;
- Criar tipos abstratos de dados;

Reuso

- Código mais gerenciável;
- Uso eficiente de partes do código que tenha algum propósito específico (evita redundância);
- Consistência do código;

Principais Conceitos

Namespace control: permite o reuso de nomes, desde que sejam únicos para o mesmo módulo.

Agrupamento de entidades relacionadas: variáveis, funções, tipos, etc.

Não são first-class: não podem ser passados como valores.

Particionar programas: servem para dividir o programa em agrupamentos lógicos que façam sentido.

obs.: Um módulo por arquivo
(restrição do GHC e não da linguagem)

Principais Conceitos

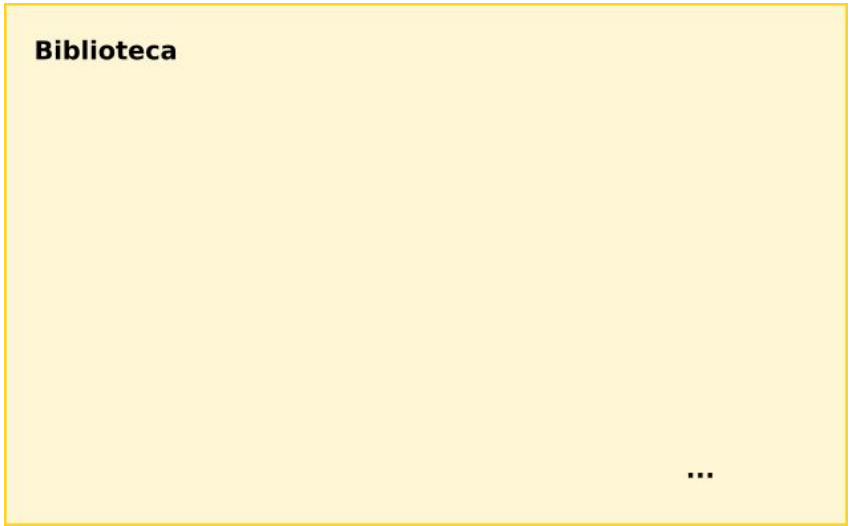
Importação: para um módulo ser usado ele precisa ser importado.

Bibliotecas: são coleções de módulos.

Principais Conceitos

Importação: para um módulo ser usado ele precisa ser importado.

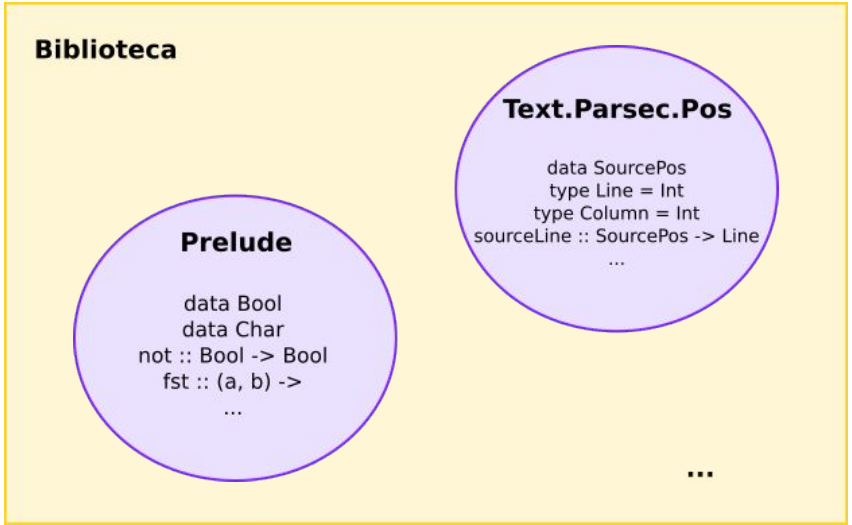
Bibliotecas: são coleções de módulos.



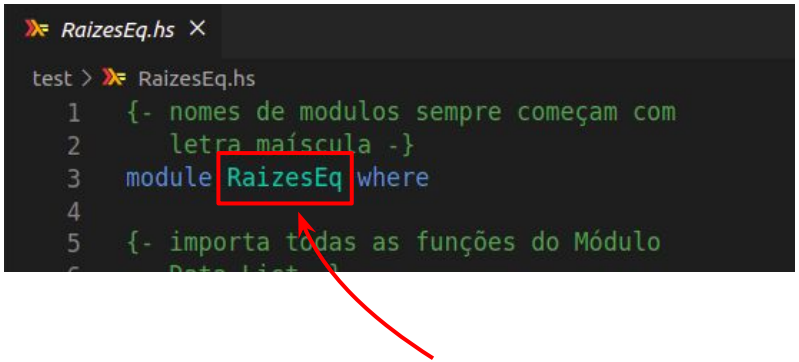
Principais Conceitos

Importação: para um módulo ser usado ele precisa ser importado.

Bibliotecas: são coleções de módulos.

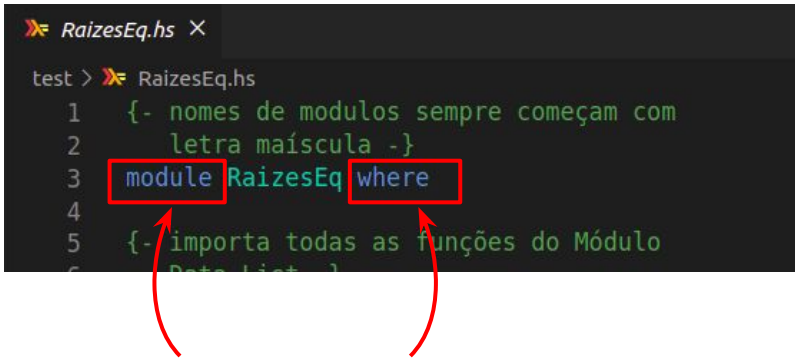


Definição de um módulo



- Os nomes dos módulos são alfanuméricos e devem sempre **começar com uma letra maiúscula**.

Definição de um módulo



- Os nomes dos módulos são alfanuméricos e devem sempre **começar com uma letra maiúscula**.
- As keywords **`module`** e **`where`** são usadas para definir um módulo.

Importação

RaizesEq.hs

test > RaizesEq.hs

```haskell
{- nomes de modulos sempre começam com
   letra maíscula -}
module RaizesEq where

{- importa todas as funções do Módulo
   Data.List -}
import Data.List

{- importa apenas as funções toLower
   e toUpper do módulo Data.char-}
import Data.Char (toLower, toUpper)
```

- Os nomes dos módulos são alfanuméricos e devem sempre **começar com uma letra maiúscula**.
- As keywords **`module`** e **`where`** são usadas para definir um módulo.
- Módulos podem importar outros módulos.

Importação

RaizesEq.hs

test > RaizesEq.hs

```haskell
{- nomes de modulos sempre começam com
   letra maíscula -}
module RaizesEq where

{- importa todas as funções do Módulo
   Data.List -}
import Data.List

{- importa apenas as funções toLower
   e toUpper do módulo Data.char-}
import Data.Char (toLower, toUpper)
```

- Os nomes dos módulos são alfanuméricos e devem sempre **começar com uma letra maiúscula**.
- As keywords **`module`** e **`where`** são usadas para definir um módulo.
- Módulos podem importar outros módulos.
- Módulos podem importar funções de outros módulos;

Qualified Names

E se dois módulos têm diferentes entidades com o mesmo nome?

Main.hs

```haskell
module Tree (Tree(Leaf, Branch), fringe) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x) = [x]
fringe (Branch left right) = fringe left ++ fringe right

------------------------------------------------------------------
module Fringe(fringe) where
import Tree(Tree(..))

fringe :: Tree a -> [a]    -- definição diferente para a função fringe
fringe (Leaf x) = [x]
fringe (Branch x y) = fringe x
```

Qualified Names

E se dois módulos têm diferentes entidades com o mesmo nome?

- Uso de *qualified names*.
- Keyword **`qualified`**.

Main.hs

```haskell
module Tree (Tree(Leaf, Branch), fringe) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x) = [x]
fringe (Branch left right) = fringe left ++ fringe right

-----------------------------------------------------------------
module Fringe(fringe) where
import Tree(Tree(..))

fringe :: Tree a -> [a]    -- definição diferente para a função fringe
fringe (Leaf x) = [x]
fringe (Branch x y) = fringe x

-----------------------------------------------------------------
module Main (main) where
import Tree (Tree (Leaf, Branch), fringe)
import qualified Fringe (fringe)

main = do print(fringe (Branch (Leaf 1) (Leaf 2)))
          print (Fringe.fringe (Branch (Leaf 1) (Leaf 2)))
```

Qualified Names

E se dois módulos têm diferentes entidades com o mesmo nome?

- Uso de *qualified names*.
- Keyword **`qualified.`**
- Faz com que nomes importados possam ser prefixados com o nome do módulo.

Main.hs

```haskell
module Tree (Tree(Leaf, Branch), fringe) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x) = [x]
fringe (Branch left right) = fringe left ++ fringe right

-----------------------------------------------------------------
module Fringe(fringe) where
import Tree(Tree(..))

fringe :: Tree a -> [a]    -- definição diferente para a função fringe
fringe (Leaf x) = [x]
fringe (Branch x y) = fringe x

-----------------------------------------------------------------
module Main (main) where
import Tree (Tree (Leaf, Branch), fringe)
import qualified Fringe (fringe)

main = do print(fringe (Branch (Leaf 1) (Leaf 2)))
          print (Fringe.fringe (Branch (Leaf 1) (Leaf 2)))

```

Relação com Outras Linguagens

Foreign Function Interface: Interface de comunicação entre haskell e outras linguagens de programação.

Foreign types: Tipos de outras linguagens devem ser importados caso seja necessário ex: CLong, CShort

Ponteiros de funções: Haskell suporta ponteiros de funções de outras linguagem além de passar ponteiros da sua funções

```haskell
crypt' :: AESCtx -> B.ByteString -> Int -> IO B.ByteString
crypt' (ECBCtx (EncryptCtx ctx))    = call _aes_ecb_encrypt ctx
crypt' (ECBCtx (DecryptCtx ctx))    = call _aes_ecb_decrypt ctx
crypt' (CBCCtx iv (EncryptCtx ctx)) = calliv _aes_cbc_encrypt iv ctx
crypt' (CBCCtx iv (DecryptCtx ctx)) = calliv _aes_cbc_decrypt iv ctx
crypt' (CFBCtx iv Encrypt ctx)      = calliv _aes_cfb_encrypt iv ctx
crypt' (CFBCtx iv Decrypt ctx)      = calliv _aes_cfb_decrypt iv ctx
crypt' (OFBCtx iv ctx)              = calliv _aes_ofb_crypt iv ctx
crypt' (CTRCtx iv ctx)              = aes_ctr_crypt iv ctx

aes_ctr_crypt :: IV -> EncryptCtxP -> B.ByteString -> Int -> IO B.ByteString
aes_ctr_crypt (IV ctr) ctx (BI.toForeignPtr -> (bs,offset,len)) retLen =
  withForeignPtr ctx $ \ctxp ->
  withForeignPtr bs $ \bsp ->
  withForeignPtr ctr $ \ctrp ->
  BI.create retLen $ \obuf ->
  ensure $ _aes_ctr_crypt (bsp `plusPtr` offset) obuf len ctrp _ctr_inc ctxp

foreign import ccall unsafe "aes_ecb_encrypt" _aes_ecb_encrypt
  :: Ptr Word8 -> Ptr Word8 -> Int -> Ptr EncryptCtxStruct -> IO Int
foreign import ccall unsafe "aes_ecb_decrypt" _aes_ecb_decrypt
  :: Ptr Word8 -> Ptr Word8 -> Int -> Ptr DecryptCtxStruct -> IO Int
foreign import ccall unsafe "aes_cbc_encrypt" _aes_cbc_encrypt
  :: Ptr Word8 -> Ptr Word8 -> Int -> Ptr Word8 -> Ptr EncryptCtxStruct -> IO Int
foreign import ccall unsafe "aes_cbc_decrypt" _aes_cbc_decrypt
  :: Ptr Word8 -> Ptr Word8 -> Int -> Ptr Word8 -> Ptr DecryptCtxStruct -> IO Int
foreign import ccall unsafe "aes_cfb_encrypt" _aes_cfb_encrypt
  :: Ptr Word8 -> Ptr Word8 -> Int -> Ptr Word8 -> Ptr EncryptCtxStruct -> IO Int
foreign import ccall unsafe "aes_cfb_decrypt" _aes_cfb_decrypt
  :: Ptr Word8 -> Ptr Word8 -> Int -> Ptr Word8 -> Ptr EncryptCtxStruct -> IO Int
foreign import ccall unsafe "aes_ofb_crypt" _aes_ofb_crypt
  :: Ptr Word8 -> Ptr Word8 -> Int -> Ptr Word8 -> Ptr EncryptCtxStruct -> IO Int
foreign import ccall unsafe "aes_ctr_crypt" _aes_ctr_crypt
  :: Ptr Word8 -> Ptr Word8 -> Int -> Ptr Word8 -> FunPtr (Ptr Word8 -> IO ()) -> Ptr EncryptCtxStruct -> IO Int
foreign import ccall unsafe "&ctr_inc" _ctr_inc :: FunPtr (Ptr Word8 -> IO ())
```

Exemplos

Geometry

```haskell
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where

sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)

sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side

cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side

cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectangleArea a b * c

cuboidArea :: Float -> Float -> Float -> Float
cuboidArea a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

Módulo Geometry, que permite calcular o volume e a área de Esferas, Cubos e Prismas, ou seja, objetos tridimensionais

- Importação usando o comando

```haskell
import Geometry
```

- rectangleArea é usada dentro do módulo, mas não exportada

Exemplos

Geometry

`Sphere.hs`

```haskell
module Geometry.Sphere
( volume
, area
) where

volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)

area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)
```

`Cube.hs`

```haskell
module Geometry.Cube
( volume
, area
) where

import qualified Geometry.Cuboid as Cuboid

volume :: Float -> Float
volume side = Cuboid.volume side side side

area :: Float -> Float
area side = Cuboid.area side side side
```

`Cuboid.hs`

```haskell
module Geometry.Cuboid
( volume
, area
) where

volume :: Float -> Float -> Float -> Float
volume a b c = rectangleArea a b * c

area :: Float -> Float -> Float -> Float
area a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

Módulo Geometry separado em 3 módulos, todos dentro da mesma pasta Geometry

- Os 3 módulos exportam as mesmas funções, então deve-se realizar importações qualificadas:

```haskell
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
```

- Importação deve ser realizada em um arquivo no mesmo nível da pasta Geometry
- Por fim, basta chamar as funções como

`Sphere.area`, `Sphere.volume`, `Cuboid.area`,

Criptografia AES

```haskell
-- | A pure interface to AES
module Codec.Crypto.AES(
  Mode(..), Direction(..), crypt, crypt'
  ) where

import qualified Codec.Crypto.AES.Monad as AES
import Codec.Crypto.AES.Monad(Mode(..), Direction(..))
import qualified Data.ByteString as B
import qualified Data.ByteString.Lazy as BL

-- | Encryption/decryption for lazy bytestrings
crypt :: Mode
          -> B.ByteString -- ^ The AES key - 16, 24 or 32 bytes
          -> B.ByteString -- ^ The IV, 16 bytes
          -> Direction
          -> BL.ByteString -- ^ Bytestring to encrypt/decrypt
          -> BL.ByteString
crypt mode key iv dir bs = snd $ AES.runAES mode key iv dir (AES.crypt bs)

-- | Encryption/decryption for strict bytestrings
crypt' :: Mode
          -> B.ByteString -- ^ The AES key - 16, 24 or 32 bytes
          -> B.ByteString -- ^ The IV, 16 bytes
          -> Direction
          -> B.ByteString -- ^ Bytestring to encrypt/decrypt
          -> B.ByteString
crypt' mode key iv dir bs = B.concat $ BL.toChunks $ snd $ AES.runAES mode key iv dir (AES.crypt bs)
```

[Módulo AES no Hackage]

Exemplos

Criptografia AES

```haskell
-- | Primitive (in IO) AES operations
{-# CFILES cbits/aescrypt.c cbits/aeskey.c cbits/aestab.c cbits/aes_modes.c #-}
module Codec.Crypto.AES.IO(
  newCtx, newECBCtx, Direction(..), Mode(..), AESCtx, crypt
  ) where

import qualified Data.ByteString as B
import qualified Data.ByteString.Internal as BI

import Foreign
import Control.Applicative
import Control.Monad

#include "aesopt.h"
#include "aes.h"
#include "aestab.h"
#include "brg_endian.h"
#include "ctr_inc.h"

newtype AESKey = AESKey B.ByteString
               deriving(Show)


toKey :: B.ByteString -- ^ Must be 16, 24 or 32 bytes
      -> AESKey
toKey bs | B.length bs `elem` [16,24,32] = AESKey bs
         | otherwise = error $ "toKey: Key has wrong length: " ++ show (B.length bs)

newtype IV = IV (ForeignPtr Word8)

data Direction = Encrypt | Decrypt
```

```haskell
-- | An occasionally pure, monadic interface to AES
module Codec.Crypto.AES.Monad(
  AES, AEST, Mode(..), Direction(..), Cryptable(..), runAEST, runAES
  ) where

import qualified Codec.Crypto.AES.IO as AES
import Codec.Crypto.AES.IO(Mode(..), Direction(..), newCtx, AESCtx)
import Control.Applicative
import Control.Monad.ST.Lazy
import Control.Monad.Reader
import Control.Monad.Writer
import Control.Monad.UnsafeIO
import qualified Data.ByteString as B
import qualified Data.ByteString.Lazy as BL

type AEST m a = ReaderT AESCtx (WriterT BL.ByteString m) a

type AES s a = AEST (ST s) a

-- | Run an AES computation
runAEST :: MonadUnsafeIO m =>
           Mode
           -> B.ByteString -- ^ The AES key - 16, 24 or 32 bytes
           -> B.ByteString -- ^ The IV, 16 bytes
           -> Direction
           -> AEST m a
           -> m (a, BL.ByteString)
runAEST mode key iv dir aes = do
  ctx <- liftUnsafeIO $ newCtx mode key iv dir
  runWriterT $ runReaderT aes ctx
```

Conclusões

Uma recapitulação

- Facilitam a organização de códigos complexos
- Permitem a reutilização e compartilhamento entre diferentes projetos
- Reduzem conflitos de nomes
- Módulos clássicos (não first-class) não facilitam a escolha de implementação em runtime
- São puramente estáticos, estruturas apenas reconhecidas durante a compilação

Obrigado!

André Casagranda
André Zanghelini
Felipe Losso
Rafael Oliveira