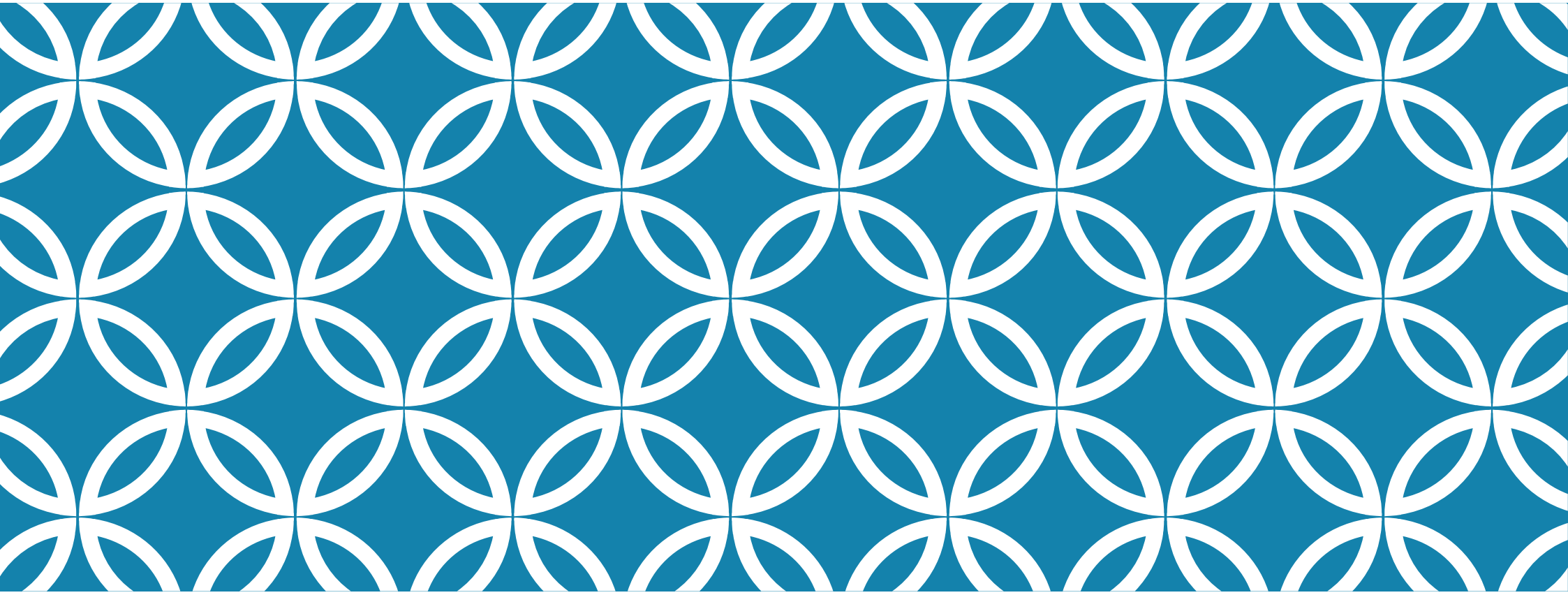


HASKELL函数式编程简介

语言的分类

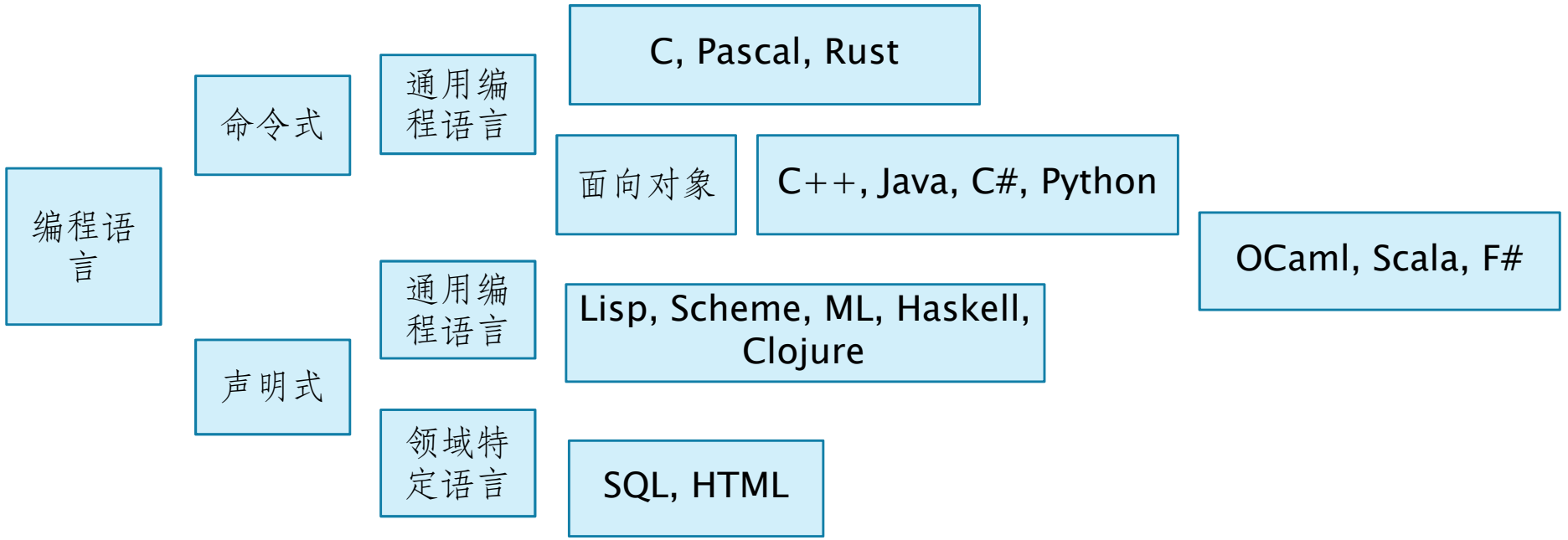
命令式（imperative）：用详细的指令描述如何完成计算。例如C、C++、Java、Fotran：

```
User[] users
for(var i=0; i<users.length; i++)
{
    if(users[i].user_name=="Alice")
    {
       print(user[i]);
    }
}
```

声明式（declarative）：描述要什么样的计算，语言并不直接描述如何修改变量、内存、CPU寄存器存储状态。把计算的描述从描述计算过程中解放出来，例如SQL语言等DSL。函数式编程语言往往被认为是声明式的。

```
SELECT * from user WHERE user_name = "Alice"
```

语言的分类


编程语言
命令式
通用编程语言
C, Pascal, Rust
面向对象
C++, Java, C#, Python
OCaml, Scala, F#
声明式
通用编程语言
Lisp, Scheme, ML, Haskell, Clojure
领域特定语言
SQL, HTML

什么是HASKELL

一个有着强大的类型推断系统、静态强类型、惰性的、纯的函数式编程语言。

类型推断系统：在没有开语言扩展前提下所有表达式的类型可以推断，用户可以省略所有的类型标注

静态强类型：所有表达式必须有明确的类型，不支持隐式转型。例如：

惰性：在求值过程中，只求自己需要的部分

纯函数：输入确定，返回结果确定的函数，对于有IO操作的函数需要在类型上加以限制

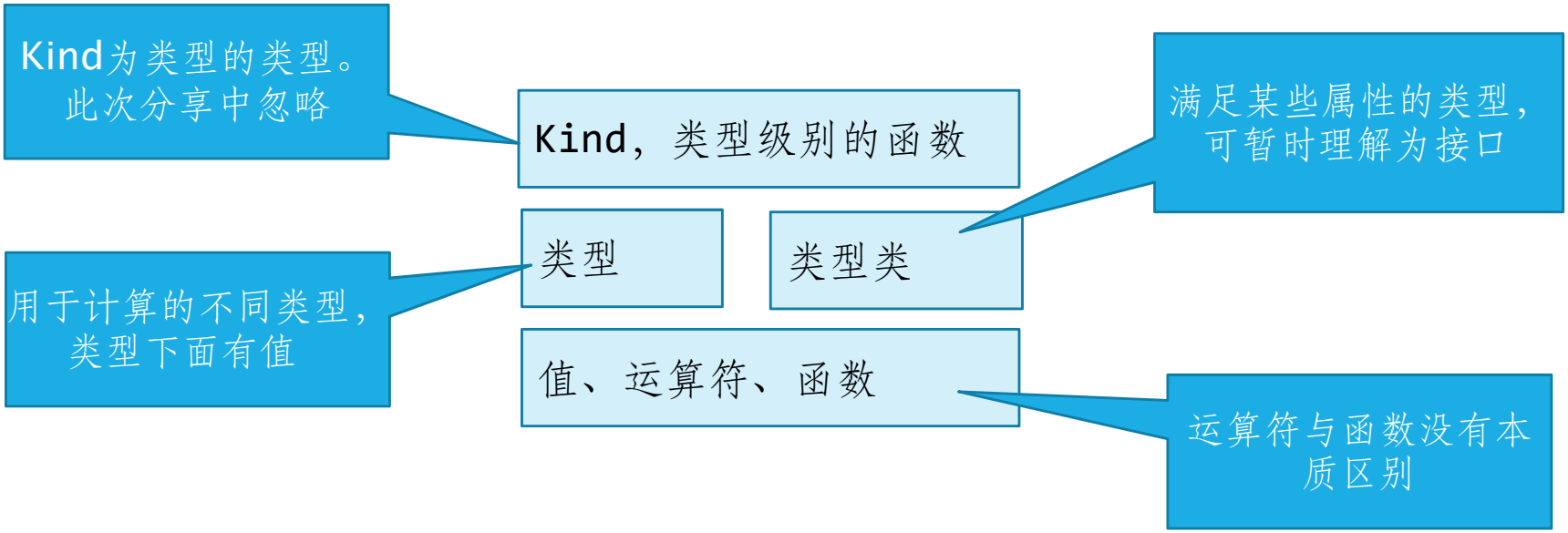
函数式：以函数为第一成员，支持以函数为参数或作为返回值的语言。

```
a = "haha"
b = True
c = 'c'
d = 1
e = \x -> x
```

```
-- Haskell
c :: Char
c = 65 -- type error

// C
char c = 65 // seems ok
```

```
> [1..]
[1,2,3.....]

> take 10 [1..]
??
```

主要概念


Kind为类型的类型。
此次分享中忽略
Kind，类型级别的函数
满足某些属性的类型，
可暂时理解为接口
类型
类型类
用于计算的不同类型，
类型下面有值
值、运算符、函数
运算符与函数没有本
质区别

值、运算符、函数

```
> 1 :: Int
1
> :type 'c'
'c' :: Char
> 1 == 1
True
> :type (==)
(==) :: Eq a => a -> a -> Bool
> (==) 1 1
True
> :t [1,2,3]
[1,2,3] :: Num a => [a]
> :t (True, "abc")
(Bool, String)
```

Eq为类型类

运算符与函数无本质区别

定义函数

```
name :: <type>
name [args] = function body
```

```
id :: a -> a
id x = x

const :: a -> (b -> a)
const x y = x
```

小写字母开头的类型为多态类型，即可以被替换为任何其他类型的类型

->为右结合的
a -> (b -> a)

函数应用为左结合的
id id 5 = (id id) 5

```
const id 5 False
```

定义函数

模式匹配

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n - 1)
```

```
fact :: Int -> Int
fact n = case n of
           0 -> 1
           n -> n * fact (n - 1)
```

条件表达式:

```
fact n = if n == 0 then 1 else n * fact (n - 1)
```

```
  fact 3
= 3 * fact 2
= 3 * (2 * fact 1)
= 3 * (2 * (1 * fact 0))
= 3 * (2 * (1 * 1))
= 6
```

守卫表达式:

otherwise为布尔值

```
fact n | n == 0 = 1
       | otherwise = n * fact (n - 1)
```

高阶函数

通常以函数为参数的函数被称为高阶函数

```
map :: (a -> b) -> ([a] -> [b])
map f [] = []
map f (x:xs) = f x : map f xs

> map (+1) [1,2,3]
[2,3,4]

> map even [1,2,3]
[False, True, False]

> map (const "abc") [1,2,3]
```

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

> iterate (+1) 1
[1,2,3,4,5,6,7,8,9,10,11...

返回
[x, f x, f (f x), f (f (f x)) ...
```

类型

```
data Bool = False | True

not False = True
not True = False

(||) :: Bool -> Bool -> Bool
(||) False a = a
(||) a False = a
(||) _ _ = True
```

```
data List a = Nil | Cons a (List a)

head Nil = error "head empty list"
head (Cons a _) = a

map' :: (a -> b) -> List a -> List b
map' f Nil = Nil
map' f (Cons x xs) = Cons (f x) (map' f
xs)
```

```
data Maybe a = Nothing | Just a

safeDiv :: Int -> Int -> Maybe Int
safeDiv a 0 = Nothing
safeDiv a b = Just (div a b)
```

```
data Tree a = Leaf | Node a (Tree a) (Tree a)

tree2List Leaf = []
tree2List (Node a l r) = [a] ++ tree2List l ++
tree2List r
```

类型类

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y               = not (x == y)
  x == y               = not (x /= y)

elem :: Eq a => a -> [a] -> Bool
elem x [] = False
elem x (e:es) | x == e = True
              | otherwise = elem x es

data Color = Red | Green | Blue

instance Eq Color where
  Red == Red = True
  Green == Green = True
  Blue == Blue = True
  _ == _ = False

> elem Blue [Red, Blue, Blue]
```

元素是否在列表中

实现了Eq就可以用elem

不需要再实现/=，==与/=只需要定义其中之一

类型类

```
mapList ::  (a -> b) -> List a -> List b
mapList f Nil = Nil
mapList f (Cons a l) = Cons (f a) (mapList f l)

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Leaf = Leaf
mapTree f (Node n l r) = Node (f n) (mapTree f l) (mapTree f r)
```

类型类

```
mapList ::  (a -> b) -> List a -> List b
mapList f Nil = Nil
mapList f (Cons a l) = Cons (f a) (mapList f l)

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Leaf = Leaf
mapTree f (Node n l r) = Node (f n) (mapTree f l) (mapTree f r)
```

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

instance Functor List where
  fmap f Nil = Nil
  fmap f (Cons a l) = Cons (f a) (fmap f l)

instance Functor Tree where
  fmap f Leaf = Leaf
  fmap f (Node n l r) = Node (f n) (fmap f l) (fmap f r)
```

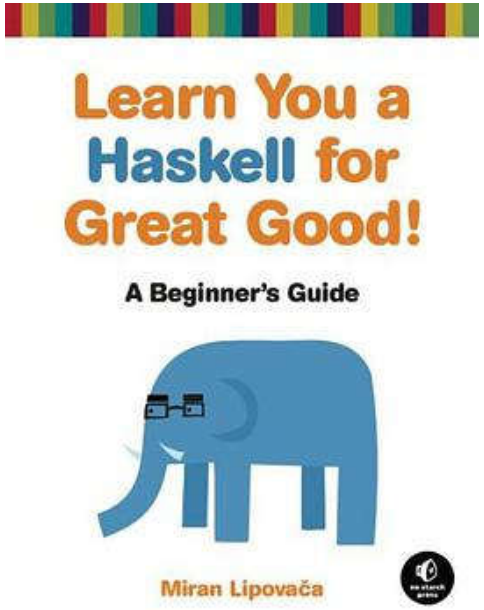
类型类

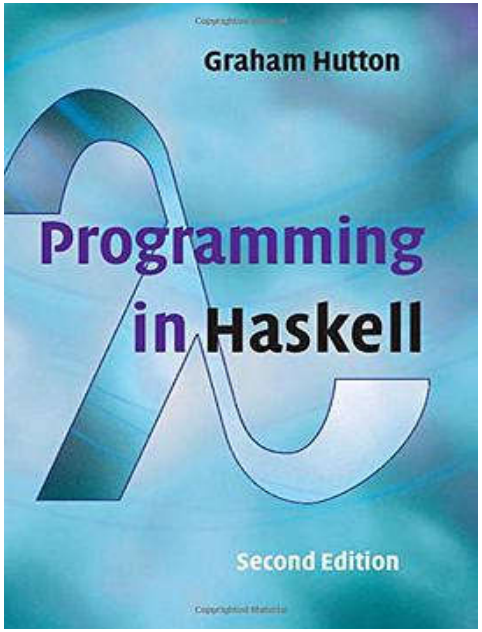
```
mapList ::  (a -> b) -> List a -> List b
mapList f Nil = Nil
mapList f (Cons a l) = Cons (f a) (mapList f l)

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Leaf = Leaf
mapTree f (Node n l r) = Node (f n) (mapTree f l) (mapTree f r)
```

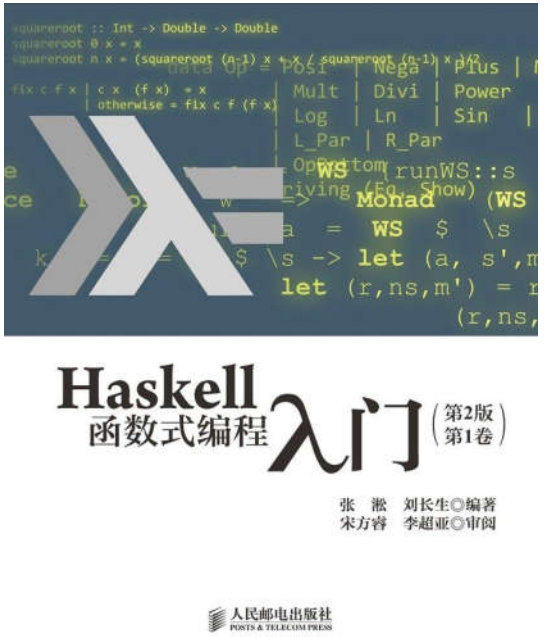
```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor List where
  fmap f Nil = Nil
  fmap f (Cons a l) = Cons (f a) (mapList f l)

instance Functor Tree where
  fmap f Leaf = Leaf
  fmap f (Node n l r) = Node (f n) (mapTree f l) (mapTree f r)
```

f 是一高阶类型构造器!

书籍


Learn You a Haskell for Great Good!
A Beginner's Guide
Miran Lipovača


Graham Hutton
Programming in Haskell
Second Edition


Haskell
函数式编程入门
(第2版 第1卷)
张 湝 刘长生◎编著
宋方睿 李超亚◎审阅
人民邮电出版社
POSTS & TELECOM PRESS

deeplang西南交大交流群

该二维码7天内(5月3日前)有效，重新进入将更新

Deeplang

Deeplang语言是一种自制编程语言，由来自浙大、中科大、的同学共同完成。

我们致力于将Deeplang语言打造为具有鲜明内存安全特性的面向IoT场景的语言，设计过程中参考Rust的安全机制，但又根据IoT场景的特性选择了更合适的解释执行的模式。这是一种静态类型、强类型语言，按照C-style设计语法，同时支持面向对象、过程式和逻辑式的混合范式。

Deeplang的独特安全特性帮助其具有一定的竞争力。作者正在持续的开发和迭代中。

https://github.com/deeplang-org/deeplang