

Haskell - P18

Map.null :: Map.Map k v → Bool, 用来判断一个 Map 是否为 empty

Map.size :: Map.Map k v → Int, 用来计算一个 Map 中的 key 个数 (元素个数或 mapping 个数)

Map.singleton :: k → v → Map.Map k v, 用传入的 k, v 构造一个只有一个 mapping 的 Map, 即 fromList [(k,v)]

(Eq k) =>

lookup :: k → [k, v] → Maybe v, 用于在一个字典中查找与 k 绑定的第一个 v 值, 返回 Just v, 否则返回 Nothing

Map.lookup :: (Ord k) => k → Map.Map k v → Maybe v, 与 lookup 相似, 但是在 Map 中查找

Map.member :: (Ord k) => k → Map.Map k v → Bool, 用于判断一个 key 是否在 Map 中

Map.map :: (a → b) → Map.Map k a → Map.Map k b] 与 filter 的 map, filter 类似, 注意 map

Map.filter :: (a → Bool) → Map.Map k a → Map.Map k a] 与 filter 的函数都仅作用于 value, 与 key 无关

Map.

toList :: Map.Map k v → [(k, v)], 与 fromList 相反, 将一个 Map 转化为 [pair].

但应注意由于 fromList 过程中可能丢失部分信息, 所以不是严格的 inverse, toList 是一对多的.

Map.keys :: Map.Map k v → [k]] 分别返回一个 Map 中的 key 列表和 value 列表.

Map.elems :: Map.Map k v → [v]] 分别与 (map fst). (Map.toList). go (map snd). (Map.toList) 等价

Map.fromListWith :: (v → v → v) → [(k, v)] → Map.Map k v] 与 fromList 和 insert 的区别在于遇到

Map.insertWith :: (v → v → v) → k → v → Map.Map k v → Map.Map k v 同样 key 值时从直接丢旧值

传入参数函数类型, 用在相同的 key 值时解决冲突 改为将旧值与新值传入函数得出结果

如 Map.fromListWith (+) [(1,1), (2,3), (5,8), (1,3), (2,1)] → [(1,4), (2,4), (5,8)]

而 Map.fromList [(1,1), (2,3), (5,8), (1,3), (2,1)] → [(1,3), (2,1), (5,8)]

Data.Set 与 Data.Map 类似, 但主要是作用在定义的一个类型 Set 上, 操作与数学上的集合类似.

另外由于同样与 Prelude 和 Data.List 有 name clash, 所以加载应用 import qualified Data.Set as

注意 Set.Set 的基本型态为 fromList [a], 当 a 为 (key, value) 时, 即为 Map, 但仍有不同

Set.fromList :: (Ord a) => [a] → Set.Set a, 用于将一个 a 的列表转变为一个 a 的集合 Set. Set a

注意 Set.Set 与 Map.Map 的区别在于, 如果作用于 [pair], Set 并不要求 key 值 (fst) 唯一,

如 Set.fromList [(1,1), (2,3), (5,8), (1,3), (2,1)] → fromList [(1,1), (1,3), (2,1), (2,3), (5,8)]

而 Map.fromList → fromList [(1,3), (2,1), (5,8)]

Flaskell - P19

Typeclass

Functor 是一种特别的 Typeclass, 支持函数 fmap :: (Functor f) \Rightarrow (a \rightarrow b) \rightarrow (f a) \rightarrow (f b)
fmap applies a function to a value 可以实现为 fmap (+3) Just 9
which is inside a context 有 class Functor f where 可视为 constraint
a Typeclass - Container(Context) fmap :: (a \rightarrow b) \rightarrow (f a) \rightarrow (f b)
Functor 的实例可用于多种类型, 只要在 fmap 和使用时满足 pattern matching

instance Functor Maybe where 注意: 在以下代码中, 相当于给出了这些类型
fmap f (Just a) = Just (f a) so Maybe, List, Function, 作为
fmap f Nothing = Nothing Functor 的成员时, fmap 如何作用于
这些类型.

instance Functor [] where 特别是如 Maybe, [], (\rightarrow a) 表示
fmap = map 是这些类型的 constructor (构造函数)
即在定义类时所描述的所有情况.
所以此处不可以写上参数, 或需要通过
pattern matching 的东西.

instance Functor (a \rightarrow a) where
fmap f g = f.g (复合函数)

<*> Applicative Functor,

- Both values and functions inside contexts
- Apply a function (inside context) to a value (inside context)
 $\text{to } [(*)^2, (+3)] <*> [1, 2, 3] \rightarrow [2, 4, 6, 4, 5, 6]$

to a value inside a context

Monad - 一个 Typeclass, Apply a function (which returns a value inside a context)
class Monad m where

$(\gg=)$ $(\gg=) :: (\underline{m a}) \rightarrow (\underline{a \rightarrow (m b)}) \rightarrow (m b)$

(bind) 传入一个 Monad 传入参数函数类型

instance Monad Maybe where

Nothing $\gg= f = \text{Nothing}$

Just a $\gg= f = fa$

$\text{to } \text{Just } 20 \gg= \text{half} \gg= \text{half} \rightarrow \text{Just } 5$

half :: (Integral a) $\Rightarrow a \rightarrow \text{Maybe a}$

half x = if (even x) then Just (x `div` 2) else Nothing

Haskell — P20

$\text{U}, \cap, - :: (\text{Ord } a) \Rightarrow \text{Set}.\text{Set } a \rightarrow \text{Set}.\text{Set } a \rightarrow \text{Set}.\text{Set } a$, 用于集合间的运算, 与数学上~~对应~~对~~应~~并集(\cup): $\text{Set}.\text{union}$, 交集(\cap): $\text{Set}.\text{intersection}$, 差集($-$): $\text{Set}.\text{difference}$

$\text{Set}.\text{empty} :: \text{Set}.\text{Set } a$, 用于生成一个空的 Set, 其值为 $\text{fromList} []$

$\text{Set}.\text{null} :: \text{Set}.\text{Set } a \rightarrow \text{Bool}$, 用于判断一个 Set 是否为空集(\emptyset)

$\text{Set}.\text{size} :: \text{Set}.\text{Set } a \rightarrow \text{Int}$, 用于计算集合中的元素个数, 等同于 $(\text{length } s)$

$\text{Set}.\text{member} :: (\text{Ord } a) \Rightarrow a \rightarrow \text{Set}.\text{Set } a \rightarrow \text{Bool}$, 用于判断一个元素是否在集合中, 即 $(\text{命题 } a \in s)$

$\text{Set}.\text{singleton} :: a \rightarrow \text{Set}.\text{Set } a$, 用于生成一个单元素集合, 即 $(s = \{a\})$

$\text{Set}.\text{insert} :: (\text{Ord } a) \Rightarrow a \rightarrow \text{Set}.\text{Set } a \rightarrow \text{Set}.\text{Set } a$, 用于向集合中添加/从集合中删除元素

$\text{Set}.\text{delete} :: a \rightarrow \text{Set}.\text{Set } a \rightarrow \text{Set}.\text{Set } a$, 等同于 $(s \cup \{a\}) / (s - \{a\})$

子集 $:: (\text{Ord } a) \Rightarrow \text{Set}.\text{Set } a \rightarrow \text{Set}.\text{Set } a \rightarrow \text{Bool}$, 用于判断集合是否为另一个的子集(真子集)

(命题 $A \subseteq B$): $\text{Set}.\text{isSubsetOf}$, (命题 $A \subset B$): $\text{Set}.\text{isProperSubsetOf}$

$\text{Set}.\text{map} :: (\text{Ord } b) \Rightarrow (a \rightarrow b) \rightarrow \text{Set}.\text{Set } a \rightarrow \text{Set}.\text{Set } b$ 与 $\text{List}.\text{map}$ 、 $\text{map}.\text{filter}$ 类似, 但是注意

$\text{Set}.\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow \text{Set}.\text{Set } a \rightarrow \text{Set}.\text{Set } a$ $\text{Set}.\text{map}$ 要求 $(\text{Ord } b)$ 是因为需对 $\text{Set}.\text{Set } b$ 排序

$\text{Set}.\text{toList} :: \text{Set}.\text{Set } a \rightarrow [a]$, 用于将集合 Set 转变为普通的 List.

注意由于在形成集合时已经排序, 所以 $\text{Set}.\text{toList} . \text{Set}.\text{fromList}$ 与 $\text{List}.\text{map}$ 相似 (尤其是长列表)

与 nub 相比 $\text{Set}.\text{toList} . \text{Set}.\text{fromList}$ 可以实现去掉重复元素的功能, 但与 nub 相比速度更快

原因是其过程类似 $\text{nub}.\text{sort}$, 所以只需要去掉连续重复元素即可, 但会失去原始排序

但需要注意 nub 只要求排序 List 中为 (Eq) 的成员, 而 $\text{Set}.\text{toList} . \text{Set}.\text{fromList}$ 需求 (Ord)

Type Inference given a function def \rightarrow type of the function?

Milner, Hindley, Damas

1969 \rightarrow 1978 \rightarrow 1982

$f 0 = 1$

pattern matching

(Exponential Time (Complete))

$f :: (\text{Num } a) \Rightarrow a \rightarrow a$

无法通过

Linear Time (in case)

注意这个类型依旧是 $a \rightarrow a$, 只不过非参数

Haskell - P21

III - Type Inference

Static Type Inference

① Build a pass tree

② Introduce Type variables for each node in the pass tree

③ Generate a list of constraints on the Type variable

④ Solve the system of constraints

(1)(2) : 根据 (+) 的类型与 T_3 中 8 的类型

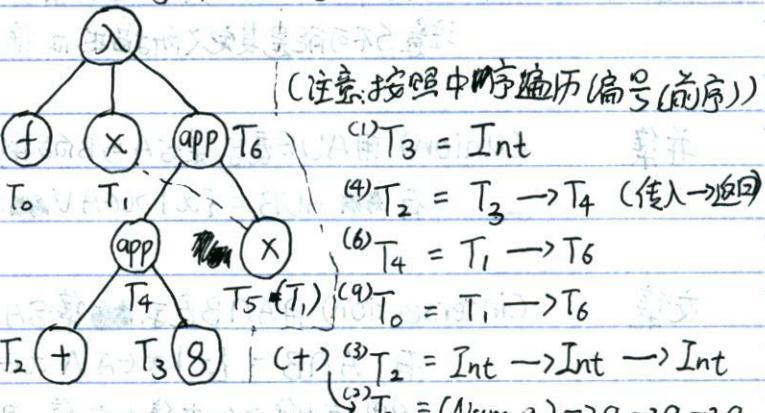
(3) 得到 $T_2 = \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

(1)(3)(4) : $T_3 \rightarrow T_4 = \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

则有: $T_4 = \text{Int} \rightarrow \text{Int}$ (5)

(5)(6) : $T_1 \rightarrow T_6 = \text{Int} \rightarrow \text{Int}$ 则 (7)(8)(9) 得到 $T_0 = \text{Int} \rightarrow \text{Int}$

则有 $T_1 = \text{Int}$ (7), $T_6 = \text{Int}$ (8), 于是有 $f :: \text{Int} \rightarrow \text{Int}$



Unification Given two Terms T_1 and $T_2 \rightarrow$ find Most General Unifier (MGU) of T_1 and T_2

Mapping from Type Variables to typed Terms $\text{so: MGU}(T_1 \rightarrow T_2, \text{Int} \rightarrow T_3)$

$L T_1 \{m\} \equiv T_2 \{m\}$

$T_1 \mapsto \text{Int} \vdash T_2 \mapsto T_3$

any other unifier is more specific than MGU $T_1 \rightarrow T_2 \equiv \text{Int} \rightarrow T_3$

Application Node

Apply a function to an actual parameter

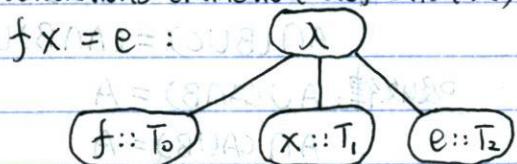
$\text{so: } f x : \text{app} :: T_2$

T_1 must be the type of the domain of f

T_2 must be the type of the range of f

$T_0 \equiv T_1 \rightarrow T_2$

Abstractions (Function definition λ)



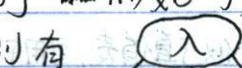
Type of f must be $d \rightarrow r$

d same type as x , r same type as body e

Constraint: $T_0 = T_1 \rightarrow T_2$

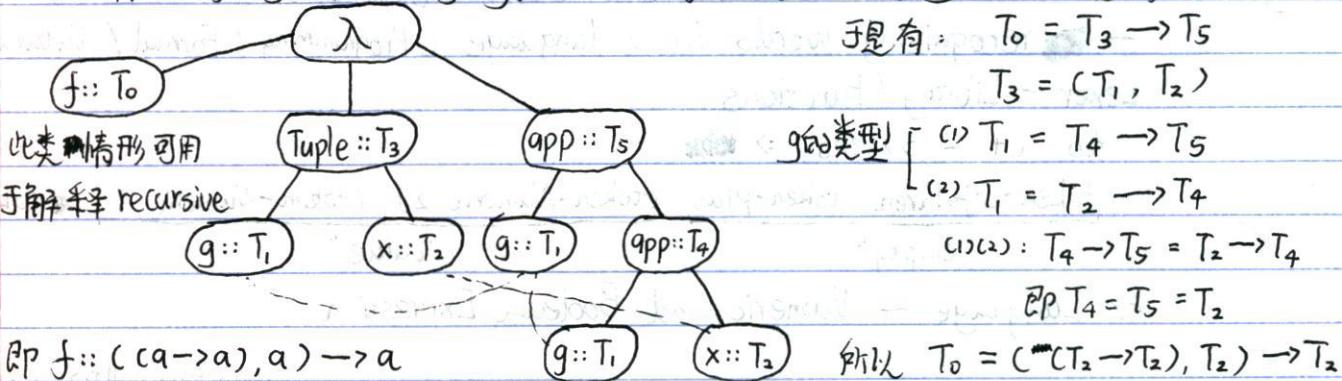
Polyomorphic 对于 形如 $g = g \circ g$, 即 对于 传入一个参数函数 g , 返回 $g \circ g$

Types 则有 $T_0 = T_1 \rightarrow T_4$ 于是有 $T_0 = T_1 \rightarrow T_4$



Haskell - P22

Polyomeric 对于形如 $f(g, x) \rightarrow g(g, x)$, 即 f 对传入的函数 g 对 x 执行两次
Types 提有: $T_0 = T_3 \rightarrow T_5$



注意: 形如 $\text{rev} [] = []$ 会得到形如 $\text{rev} :: [a] \rightarrow [b]$ 的类型
 $\text{rev} (x:xs) = \text{rev} xs$ 因为无法将函数类型与 x 的类型关联起来。

module 自定义模块在 .hs 文件的开头以 module name (func1, ...) where 的形式定义
其中 name 必须以大写字母开头, 函数可列出所有模块中定义的函数,
但是列出的函数必须在模块中有定义, 否则会在加载模块时报错,
另外即使未在开头声明, 模块中的函数也必须有效定义, 也可以直接调用
name 可以写成 main:sub 的形式, 但要注意 main 和 sub 都要写
在加载不同模块时, 要特别注意 name-clash 的情况

自定义类 形式
Type 自定义类的声明以 data name = value constructors deriving (Typeclass) 的形式
类的名字 name 必须以大写字母开头, value constructors 列举了此类型所有可能的值
如 data Bool = True | False, data Int = -2147483648 | ... | -11011 | ... | 2147483647
(注意此处仅为示例, 实际中无效)

data shape = Circle Float Float Float | Rectangle Float Float Float Float

注意 Circle 和 Rectangle 是 constructor, 即构造函数 $T_1 = \text{Float} \rightarrow T_2$

$\text{Circle} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \rightarrow \text{Shape}$

$\text{Rectangle} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \rightarrow \text{Shape}$

deriving 表示将类型声明为某个类型类 (Typeclass) 的成员

如声明成 (Show) 的成员, 类的值才可像字符串一样输出到显示

在加载模块时, 可以声明如 module Shapes (Shape(..)) where

其中 Shape(..) 相当于 Shape(Circle, Rectangle) 即 Shape 类型下

包含所有 value constructor, 无法直接构造 Map. Map 类是因为没有 export 构造函数

$T_1 = \text{Float} \rightarrow T_2$

Haskell - P23

P29 - 2023.5.25

record 在自定义类型(尤其是自定义结构 struct 时)可以使用 record syntax
Syntax record syntax 可以视为在定义 constructor(构造函数)时, 同时定义属性及调用函数
使用 record syntax 的定义形式如 data name = constructor { field1 :: Type1, field2 :: Type2, ... }
如 data Car = Car { company :: String, model :: String, year :: Int } deriving (Show)
则有两种方式调用: 1. Car "Ford" "Mustang" 1967, 这两种调用的结果是相同的
2. Car {company = "Ford", model = "Mustang", year = 1967}
在 record syntax 中, field 同时也是一个函数, 调用输出结构中对应 field 的值
 $t \text{ field} \rightarrow \text{field} : \text{name} \rightarrow \text{Type}$, 所以 $t \text{ company} \rightarrow \text{company} : \text{Car} \rightarrow \text{String}$
注意在第二种调用 constructor 函数中, field 的排序是任意的, 只要 field 的名称、数量、类型正确
另外 Show 对 record syntax 的处理方式也不相同, 在显示以 record syntax 定义的类型时
会表示成形如 constructor {field1 = value1, field2 = value2, ...} 不论调用是否使用

Type parameter 在定义 type constructor 时, 对于某些值的类型可以依据实际情况决定, 可使用 type parameter
如 data Maybe a = Nothing | Just a 中的 a, 可根据情况拥有不同类型
但是注意 a 不可能是 [], 尽管可以使用 Just [] , 但是 $t \text{ Just} \rightarrow \text{Just} : \text{Maybe} [a]$,
即 [] 必须属于某一个 [a] 类型, 与此相比 () (空 tuple) 是一个独立类型, 即 $t \text{ Just} () \rightarrow \text{Just} () : \text{Maybe} ()$
注意在 record syntax 中也可以使用, 如 data name = constructor { field1 = type parameter1, ... }
在使用 type parameter 时应注意, 大多数是用在一个类型的结构中, 并且 type parameter 并不
影响类型本身的使用, 如 [a] 中的 a 并不影响 List 本身的运作, 而对 a 的限制可以放在函数中
注意不要在定义类时使用 constraint (never add typeclass constraints in data declarations)
因为如果在类定义时有 typeclass constraint, 那么在所有使用该类的函数定义中都必须有相同的
typeclass constraint, 而不管函数本身是否需要这些限制

Derived instance 将一个 type 作为一个 Type-class 的 instance, 则这个 type 将支持 typeclass 的行为 (behavior)
如 deriving (Eq) 使 Type 支持函数 (=) 和 (/=), 且当 (Eq) 应用于 record syntax 时, 首先 Haskell 会
比较 constructor 是否一致, 然后对于其中的每个 field 的 value 是否相等 (=)
另外 (Eq) 支持 elem 函数, 用于查找某个元素 a 是否存在于列表 [a] 中 (Eq a)
如 Show 和 Read 支持函数 show 和 read, Show 用于将 value 转化为 String, read 用于将 String 转化为
value
在应用于 record syntax 时, show 将得到形如 constructor { field1 = show(value), ... } 的 String
Read 是 Show 的 inverse typeclass, read 可以读取并生成 String, 读取 "Just 't'" ::
Eq Ord 用于支持函数 compare 及其他比较, 如 Just a > Nothing, Just a > Just b = a > b,
Eq Enum 使 Typeclass 中元素拥有 predecessors 和 successors, 即通过 pred 和 succ 取前一个或后一个元素
Eq Bounded 使 Typeclass 可以读取上限与下限, 即通过 minBound :: Type, maxBound :: Type

Haskell - P24

Type (同义类型) 即在 Type checking 和 Type inference 中被视为同一类型的不同类型名字。
Synonyms 如 String 和 [Char] 在编译中是相同(equivalent)和可相互转换(interchangeable)的
Type Synonyms 可由形如 type synonym = Type 定义, 定义后 synonym 的使用等同于 Type
如: type Phone Number = String
inPhoneBook :: Name → Phone Number → PhoneBook
type Name = String
type Phone Book = [(Name, PhoneNumber)] 且在使用时, 显示同样结果,
另外 Type Synonym 可以 parameterized (参数化), 即引入 type variable.
如: 定义 type AssocList k v = [(k, v)], 此处 AssocList 是一个 type constructor
即 type (Eq k) => k → AssocList k v → Maybe v, 此处的 Maybe 也是一个 type constructor
将等同于 type (Eq k) => k → [(k, v)] → Maybe v. 即 Maybe 不是 Type, Maybe a 才是
(具体类型) value 可以具有的 Type 必须是 concrete type, concrete type 可以是定义好的
也可以由 type constructor 调用 type variable 实现 (extra type)
如: [a] apply Int → [Int], Maybe apply String → Maybe String
type constructor extra type concrete type
另外 Type Synonym 定义的 type constructor 也可以是 partially applied function
如 type AssocList' v = [(Int, v)], 等同于 type AssocList! v = (AssocList Int) v
注意: 与一般的 partially applied function 不同, type constructor 中的 type parameter 必须是 concrete type
即 type AssocList' = AssocList Int 是无效的, 要求等式右侧必须是一个 concrete type
另外注意: 在使用中不能使用如 AssocList [(1,1), (2,3)], 因为 AssocList 传入必须是 concrete type
且也不能是 [(1,1), (2,3)] :: AssocList' a, 因为 value 具有的必须是 concrete type
可以嵌套地使用 type constructor 如 [(1, Just 1)] :: AssocList' (Maybe Int)
type inference: 可以如此理解 type inference 中的 introduce type variables, 如 T₀ = T₁ → T₂
即一系列的 type synonym 如 type a₀ = a₁ → a₂, a₁ = Int, a₂ = String
然后在不冲突的前提下, 尽可能地将更多的 type synonym 转换为 concrete type
而不冲突也无法即时确定的 type synonym 则保留在 type declaration 和 type annotation 中
parameters: type synonym 中可以将多个参数 type 分别用于不同的 type constructor,
即 定义 type Either a b = Left a | Right b, 注意同一类型下的不同 type constructor
deriving (Eq, Ord, Read, Show) 必须具有不同的名字
即有 Left '1' :: Either Char b 和 Right "abc" :: Either a [Char]
特别注意: 不同类型下相同名字的 constructor, 尤其是因其他分支而形成的, 是不同的
如: 在某函数中返回的结果 a = Nothing :: Maybe Int 尽管形式上同为 Nothing
和另一函数中返回的结果 b = Nothing :: Maybe Float 但分别具有不同的 concrete type
当尝试 a == b 时将报错 couldn't match type 'Float' with 'Int'

Haskell - P25

Recursive Data Structure
当 Type 的多个 constructor 拥有相同的类型, 那么有可能创造出递归结构 Recursive Data Structure
如 list 类型, [5] 是 (5:[]) 的简写 (syntactic sugar), [4,5] 是 (4:(5:[])) 的简写 任何
(:) 左边是一个 value, 右边是一个 list, 左边具有 type a, 右边具有 type [a] ([]) 可以拥有 type [a]
data List a = Empty | Cons a (List a) 基础步骤: [] 是一个 type [a] 的列表 (list)
deviring (Show, Read, Eq, Ord)
注意 (Cons) 是一个 constructor, 且有 $:t \text{ Cons} \rightarrow \text{Cons} :: a \rightarrow \text{List } a \rightarrow \text{List } a$
则 Cons 可以作为 infix function 的形式调用, 如 5 `Cons` Empty $\rightarrow \text{Cons } 5 \text{ Empty}$
可以看出 Cons 与 (:) 是等价的, 因为 $:t (:) \rightarrow (:) :: a \rightarrow [a] \rightarrow [a]$

i 在 Haskell 中, 可以用 :t 命令查看运算符的详细定义, 包括 type constraint, 定义文件以及符号与结合顺序
如 :t (+) $\rightarrow \text{class Num where }$:t (^) $\rightarrow (\wedge) :: (\text{Num } a, \text{Integral } b) \Rightarrow a \rightarrow b \rightarrow a$
 $(+) :: a \rightarrow a \rightarrow a$ -- Defined in 'GHC.Num'
infixl 6 + -- Defined in "GHC.Num"
:t (++) $\rightarrow (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$:t (\$) $\rightarrow \text{forall } c r :: \text{GHC.Types.RuntimeRep}$
infixr 9 . -- Defined in 'GHC.Base'
:t (==) $\rightarrow \text{class Eq } a \text{ where }$ infixr 0 \$ -- Defined in 'GHC.Base'
infix 4 == -- Defined in 'GHC.Classes'

infix 可以通过 infix 关键字来定义运算符, 形式为 infix rate identifier, 如 infixl 6 +
infix 表示中缀运算, 只支持左右两个操作数, 且不支持连续使用, 如 (==) 不支持 $a == b == c$ 的形式
infixl 表示运算符为左结合 (left-associative), 如 (+) 有 $a + b + c$ 将被视为 $((a+b)+c)$
infixr 表示运算符为右结合 (right-associative), 如 复合函数 (.) 有 $f.g.h.x$ 被视为 $(f(g(h(x)))$

运算优先级 (rate), Haskell 中运算优先级标记为 0 到 9, 0 优先级最低, 9 优先级最高

如 0 1 4 (==) 5 (:) 6 7 8 9
infixl 4 (==) -- bind (>=) 比较运算符
infixl 5 (:) -- list 运算符
infixr 6 (++) -- bind (++) 列表运算符
infixr 7 (++) -- bind (++) 逻辑运算符
infixr 8 (^) -- bind (^) 数学运算符
infixr 9 (\$) -- bind (\$) 复合函数 (.)

可以参考 (++) 运算符的定义来自定义运算符 data List a = Empty | a :-: (List a) deriving (Show, Read, Ord)

infixr 5 (++) -- Defined in 'GHC.Base'
infixr 5 (++) -- bind (++)
(++) :: [a] $\rightarrow [a] \rightarrow [a]$ (++) :: List a \rightarrow List a \rightarrow List a

[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

Haskell - P26

Pattern matching pattern matching 实际上是在匹配 constructor, 即给出的 pattern 是否符合 type 定义中的某个 constructor, 如果有即通过 pattern matching, so $(x:-:xs)$ 可用是因为 $(:-:)$ 是 type 中的 constructor

typeclass $\text{class Eq } a \text{ where }$ 定义一个 typeclass 名为 Eq, a 为一个 type variable, 作为 Eq 的 instance
 $(==) :: a \rightarrow a \rightarrow \text{Bool}$] 定义了这个 typeclass 支持的 function, 实际上只是适用于 typeclass 的
 $(/=) :: a \rightarrow a \rightarrow \text{Bool}$] function 的声明 (declaration), 并未定义函数的实际实现 (根据 a 不同)

$x == y = \text{not } (x /= y)$] 定义了这两个函数之间的关系, 原则上是可以省略的, 但是缺点是 Haskell
 $x /= y = \text{not } (x == y)$] 不知道函数间关系, 另外在 instance Eq 时必须分别定义 $(==)$ 和 $(/=)$

`data TrafficLight = Red | Yellow | Green`

instance Eq TrafficLight where 定义一个 typeclass 为 instance, 使得 type 拥有 typeclass's behavior

$\text{Red} == \text{Red} = \text{True}$] 定义了这个 type 如何支持 typeclass 支持的 function

$\text{Yellow} == \text{Yellow} = \text{True}$] 即 overwrite function in instance declaration

$\text{Green} == \text{Green} = \text{True}$] fulfill the minimal complete definition, 所以仅需定义 $(==)$

$= = = = = \text{False}$] pattern matching 为兜底条件, 即不满足前面条件的均为 False

注意 minimal complete definition 指的是为了使 type 拥有 typeclass 的 behavior 所需实现的 function 最少集合

对于 Eq 来说即 $(==)$ 和 $(/=)$ 二选一即可, 如果 Eq 定义中未说明 $(==)$ 与 $(/=)$ 关系, 则两个都必须定义

$\text{instance Show TrafficLight where }$:
 $\text{show Red} = \text{"Red Light"}$] 定义 TrafficLight 如何被显示成字符串形式
 $\text{show Yellow} = \text{"Yellow Light"}$] 即在 show 函数调用中, TrafficLight 将如何生成 String
 $\text{show Green} = \text{"Green Light"}$]

特别注意在 class Eq a where so instance Eq a where 中, a 必须是一个 concrete type

即无法使用 so instance Eq Maybe where, 因为 Maybe 是 constructor 而非 concrete type

所以需要定义 so instance (Eq a) => Eq (Maybe a) where 的形式定义 instance

且必须加入 class constraints so (Eq a) =>, 否则无法保证 (Maybe a) 可以支持函数 $(==)$ 和 $(/=)$

通常来说, class declaration 为 class constraints 是为了将一个 typeclass 用作另一个的 subclass, so \Rightarrow Num \Rightarrow Eq a

如果用在 instance declaration, 则是为了表示 requirements about the contents of some type.

$a \rightarrow \text{Bool}$ 在 weakly typed language 中, 存在用非 Bool 类型的 value 或 expression 进行 if condition 判断的情况
so C++ 中 bool 为 int / char / double, 以及 Python 中非零数值, 非空字符串, 非空 list 视为 True
在 Haskell 中, 可以通过 typeclass 实现

`class YesNo a where` ; instance [a] where

`yesno :: a → Bool` ; instance [a] where

`yesno [] = False` ; instance [a] where

`yesno _ = True` ; instance [a] where

`id :: a → a, return the same thing as the parameter`