

Haskell - P1

Boolen 值: True / False 注意大写 $/=$ (\neq) 表示判断“不等于” (C++: $!=$)

succ x : 加 1 (下一个值), $\text{succ} 9 * 10 = 100$, successor, 类似 (C++: $+x$)

可在 .hs 文件中定义函数 $\text{doubleme } x : x + x$, 可以调用 $\text{doubleus} = \text{doubleme } x$

判断句式 $x = \text{if } x > 100 \text{ then } x \text{ else } x * 2$ 如果是赋值, 所有结果必须是同一类型
变量名, 函数名中允许字符串, 如 i'm

let i=1 类似变量声明 declaration

(String) 字符串类型, 等价于 [char], 可进行与集合有关的操作, ++ 连接 "Hello" ++ "world" \rightarrow "Hello world"

list [] : [2, 3] \rightarrow [1, 2, 3] : 集合添加元素, 只能在开头, 只能是单个元素! 集合的形式, 注意类型

++ 同样适用于集合, [1, 2] ++ [3, 4] \rightarrow [1, 2, 3, 4]

| [1, 2, 3] !! | \rightarrow 2 . !! 集合中元素调用, 类似 (C++ vector[i])

集合可以比较大 小; 仅当元素间可比, 按字典顺序, [3, 2, 1] $>$ [3, 1, 100] \rightarrow True

head [1, 2, 3] \rightarrow 1 tail [1, 2, 3] \rightarrow [2, 3] head [1, 2, 3, 4, 5] \rightarrow 1 tail [1, 2, 3, 4, 5] \rightarrow [2, 3, 4, 5]

init [1, 2, 3] \rightarrow [1, 2] last [1, 2, 3] \rightarrow [3] init [1, 2, 3, 4, 5] \rightarrow [1, 2, 3, 4] last [1, 2, 3, 4, 5] \rightarrow 5

length [1, 2, 3] \rightarrow 3, 类似 (C++ vector.size())

null [] \rightarrow True, 类似 (C++ vector.empty())

take 3 [1, 2, 3, 4] \rightarrow [1, 2, 3], 截取前 n 个元素, n $\in \mathbb{N}$, n 大于集合元素个数时返回集合本身

drop 3 [1, 2, 3, 4] \rightarrow [4] 剔除前 n 个元素, n $\in \mathbb{N}$, n > 集合元素个数时返回空集 []

maximum [1, 2, 3] \rightarrow 3, minimum [1, 2, 3] \rightarrow 1

sum [1, 2, 3] \rightarrow 6, 求和 product [1, 2, 3, 4] \rightarrow 24, 求积 (阶乘)

!! `elem` [1, 2, 3] \rightarrow False 查找元素是否存在于集合中

[1..10] \rightarrow [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], ['a'..'e'] \rightarrow ['a', 'b', 'c', 'd', 'e'] 列表生成

[2, 4..11] \rightarrow [2, 4, 6, 8, 10] .. [1..] \rightarrow 生成器 (iterator) 类似 Python 中的生成器

take 5 (cycle [1, 2]) \rightarrow [1, 2, 1, 2, 1], 循环并截取用于生成周期序列

take 5 (repeat 1) \rightarrow [1, 1, 1, 1, 1], 重复并截取, 用于生成指定长度的全等序列

[表达式 (参数 1, ...) | 参数 1 来源, ..., 条件 1, ...] 列表生成器, 类似 Python

boom xs = [if x < 10 then "Boom" else "Bang" | x \leftarrow xs], boom [9, 10, 11] \rightarrow ["Boom", "Bang", "Bang"]

[x + y | x \leftarrow ["a", "b"], y \leftarrow ["c", "d"]] \rightarrow ["ac", "ad", "bc", "bd"]

[[x | x \leftarrow st, x `elem` ['A'..'Z']] | st \leftarrow XSS], XSS = ["Aa", "Bb", "Cc"] \rightarrow ["A", "B", "C"]

replicate 5 'a' \rightarrow "aaaaa", 给定一个 Int 和一个元素, 生成一个重复该值 Int 次的 List

Haskell - P2

字典

元组中元素不必是同一类型，如 ("name", 100)

tuple

fst ("name", 100) → "name", snd ("name", 100) → 100, 只能用于 pair(双元素元组)

zip [1,2,3,4,5] ["a","b","c"] → [(1,"a"), (2,"b"), (3,"c")], list 组装成 tuple

zip 仅支持两个 list 组装，以元素较少一方为准

快速找到

[ca,b,c] | c ← [1..10], b ← [1..c], a ← [1..b], a^2 + b^2 = c^2] → [(3,4,5), (6,8,10)]

符合条件的

元组生成器，向量生成器，允许条件生成。

元组

Static Environment

- Type Checking

names, \Leftrightarrow associated

(identifiers) \Leftrightarrow types

Dynamic Environment

- Value, evaluations

Haskell

强类型 (strong): 拒绝执行“类型不正确”(ill-typed) 的表达式，不会自动转换类型

静态的 (static): 编译时即明确每个值和表达式的类型

自动推导 (automatically inferred): 类型推导 (type inference) 自动推导出几乎所有表达式的类型

Syntax

Type checking

Evaluation (Semantics)

values (5,5.0,'a') Obvious? recognize associated type to themselves

Variables (foo) name = value record static env. that name = no value

has type the same as

the value or the expression

expression (+) e₁+e₂

type check e₁, e₂ in the current static env.

evaluate e₁+e₂ using the current dynamic env.

the result of expression

and the apply expression

is the same as the type to those value

'a' :: Char → 'a' 类型显示签名 / 自动推导, Char (Unicode 字符), Bool, Int, Integer, Double

[a] 为类型为 a 的列表, a 的类型可以自由代换, 且可以递归定义, 如 [[Int]], [[1,2],[3,4]]

() :: () 为仅有单个值的类型, 即 0-元组, 且不支持定义 1-元组, 即不存在 Python 中的 (1,)

:type :t 可查看类型, 包括函数类型, 函数类型的表示为类似 f:A→B, 如 :t head → [a] → a

另如 :t length → (Foldable t ⇒ t) a → Int, :t lines → String → [String].

可理解为从具有指定类型的参数空间到具有指定类型的函数空间的映射, f: A → B

⇒ :t (==) → (Eq a) ⇒ a → a → Bool, ⇒ 之前的部分为 class constraint (类型强制约束)

此例中意为两个操作数的类型必须一致 (Eq), Eq 属于 Typeclass, 除 IO (适用于 I/O 的类型) 外的其他类型都属于 Eq, 此例中即要求操作数的类型需为同一 Eq 成员

Haskell - P3

Typeclass

typeclass 可以 support to implement (实现) behavior (方法), 在 typeclass 定义中的如 Eq 支持 equality testing, 实现函数为 $= =$ 和 $/ =$, 当存在时可知 $= =$ 和 $/ =$ 有被使用。函数不属于 Eq 的成员, 还有 IO 也不是, 也不是 Ord, 也不是 Show。

相等的

Ord 实现所有标准 comparing functions, 如 $>$, $<$, \geq , \leq - 一个 comparing 函数。要求两个相同的在 Ord 成员中的类型, 并输出类 Ordering (GT, LT, Eq). 一个类型只有在成为 Eq 成员后也能成为 Ord 成员。如: $t (>) \rightarrow (\text{Ord } a) \Rightarrow a \rightarrow a \rightarrow \text{Bool}$ Show 的成员都可以被表示成字符串 (String). 最常用的函数为 show: $\text{show } 3.5 \rightarrow "3.5"$, $\text{show True} \rightarrow "True"$

可显示的

Read 的成员都可以以字符串的形式被读取 (与 Show 相反), 支持函数 read:
 $t \text{ read} \rightarrow (\text{Read } a) \Rightarrow \text{String} \rightarrow a$, 如: $\text{read "True"} \rightarrow \text{True}$
注意, 如果读入的字符串可以被转换成多种类型, 则在 read 之后必须在以下选一:
按照某种可用的类型使用, 如: $\text{read "5" + 3} \rightarrow 8$, $\text{read "5" + 3.0} \rightarrow 8.0$
或者采用 type annotation (类型注释), 如: $\text{read "5"} :: \text{Int}$
否则会因为 read 函数结果类型不明确而无法编译

可列举的

Enum 的成员是可以 enumerate 的, 支持将成员类型放入 list, 实现函数 succ, pred
succ 取 successor, 如 $\text{succ 'a'} \rightarrow 'b'$, pred 取 predecessor, 如 $\text{pred } 1.5 \rightarrow 0.5$
包括 (), Bool, Char, Ordering, Int, Integer, Float, Double (也包括 list 本身)

有界的

Bounded 的成员具有确定的上界与下界, 实现函数 minBound, maxBound
 $t : t \text{ minBound } (\text{Bound } a) \Rightarrow a$, 如: $\text{minBound} :: \text{Int} \rightarrow -2^{31}$,
元组也可以是 Bounded 仅当所有元素都是 Bounded $\text{maxBound} :: (\text{Bool}, \text{Int}) \rightarrow (\text{True}, 2^{31}-1)$

数字的

Num 的成员可作为数字进行计算, 如: $t (*) \rightarrow (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a$
整数是 polymorphic constant (多态常数), 所以可以表示为 Num 的所有成员
Num 中的成员必需已是 Show 和 Eq 的成员 (随后可作为 Ordering), 包括 Integral, Floating
Integral 包括 Int, Integer, Floating 包括 Float, Double

注意: Int 和 Integer 不可混用, 即 $(5 :: \text{Int}) * (6 :: \text{Integer})$ 无法通过编译
fromIntegral 可以将一个 Integral 类型转化为 Num 中的类型, $\text{fromIntegral } (5 :: \text{Int}) + 3.0 \rightarrow 8.0$
 $t : \text{fromIntegral} \rightarrow (\text{Num } b, \text{Integral } a) \Rightarrow a \rightarrow b$, 多个 class constraint 用逗号分隔

副作用

函数的副作用指, 函数的行为受系统的全局状态影响, 本质上是不可见的 (invisible) 的输入输出
如某个选取全局变量的函数, 如果有其他函数可以修改变量, 这个函数的输出取决于全局变量
当时的状态, 尽管并没有亲自修改。

Haskell 的函数在默认下都是无副作用的, 结果只取决于显式输入的参数

纯度

不带副作用的称为“纯 (pure) 函数”, 否则是“不纯 (impure) 函数”, Haskell 中签名以 IO 开头

Haskell - P4

下划线“_”在函数调用时，“_”表示无需在意的参数，目的是忽略函数不使用的部分以提高可读性。但是如果对被忽略的参数有 class constraint，则依旧必需遵守。
如：
 $(\text{Num } a) \Rightarrow (\text{a}, \text{a}, \text{a}) \rightarrow \text{a}$ $(x, _, _) = x$, $(1, 2.0, 'b') \rightarrow \text{error}$
 $(\text{Num } c) \Rightarrow (\text{c}, \text{a}, \text{b}) \rightarrow \text{a}$ $(x, _, _) = x$, $(1, 2.0, 'b') \rightarrow \text{error}$

Pattern Matching 形式匹配，调用函数时的参数赋值是基于形式匹配已完成的，形式和类型都必须准确。
如 $(\text{Num } c) \Rightarrow (\text{c}, \text{a}, \text{b}) \rightarrow \text{a}$ $(x, _, _) = x$, $(1, 'b', 2.0) \rightarrow (x : a, _ : b, _ : c) \rightarrow 1$
列表也可以进行形式匹配，且列表可写成形如 $x : xs$ 的形式，如 $[1, 2, 3]$ 等同于 $1 : 2 : 3 : []$
如 every Third 可以写成如下形式，注意其中应的分支的集合应该是全集。

$\text{every Third} :: [a] \rightarrow [a]$ 此处类似于以下结构：switch (zs)

$\text{every Third } [] = []$ case [] : return []

$\text{every Third } x : [] = []$ case [x] : return []

$\text{every Third } x : y : [] = []$ case [x, y] : return []

$\text{every Third } x : y : z : zs = z : (\text{every Third } zs)$ case [x, y, z] ++ zs : return [z] + everyThird(zs)

关于 list 的 pattern matching 还可使用形如 $xs @ (x : y : ys)$ ，‘@’符号前的参数 xs 表示整个列表，但不可使用 “++”，因为对形如 “ $xs ++ ys$ ” 的参数，无法区分 xs 和 ys ，除非明确定义如 $xs ++ [y]$

Guard Guard (守卫) 表现在函数名下有一组竖线 (pipes)，分别引导向每个分支，并执行对应部分的函数。

如 $\text{revZip2list} :: [a] \rightarrow [b] \rightarrow [(b, a)]$

$\text{revZip2list } xs _ Ys = \text{last } Ys, \text{last } xs : []$

| $\text{length } xs == 1 _ \& \text{length } Ys == 1$ = $(\text{last } Ys, \text{last } xs) : []$

| otherwise = $(\text{last } Ys, \text{last } xs) : \text{revZip2list } (\text{init } xs) (\text{init } Ys)$

(pipes) 注意最后应有一个兜底条件 (catch-all) (对应的函数体和返回值)

可视为 if-else 结构或 switch-case-default 结构。

与 pattern matching 的差异是其引导函数体的是条件 (boolean)，而 pattern matching 的是 pattern

判断的时候从第一行开始顺序向下，如果当前 Guard 是 True 则执行当前行，否则继续下一行

也可以写成 $\text{max} :: (\text{Ord } a) \Rightarrow a \rightarrow a \rightarrow a$ $\text{max } a b | a > b = a | \text{otherwise} = b$

where

在函数定义时可以用 where 将常量或表达式 (expression) 绑定 (bind) 到 name 上，之后在函数中使用这些 name

如 $\text{revZip2list } Xs _ Ys$

| $\text{length } Xs == 1 _ \& \text{length } Ys == 1$ = $t : []$

| otherwise = $t : \text{revZip2list } (\text{init } Xs) (\text{init } Ys)$

另外 where 中也支持 pattern matching

如 $(f : _) = \text{name} :: \text{String}$

where $t = \text{last } Ys, \text{last } Xs$

可以用于表示 name 的首字母

Haskell - P5

Syntax Type Checking Evaluate
if e₁, then e₂, else e₃. e₁ must be type checked in the current static environment and e₁ must be type Bool, e₂ and e₃ must be the same type. evaluate e₁ using dynamic environment if e₁ evaluates to True, then eval. e₂ in dyn. env. and the value is value of the conditional when False goes to e₃.

define function type checking each parameter Add the function definition name e₁, e₂, ..., e_n = body e₁::T₁, e₂::T₂, ..., e_n::T_n to the dyn. env. along with a name :: T₁→T₂→...→T_n→t type the body and get t copy in current static env. in the current static env. called "Closure"

call name e₁, e₂, ..., e_n find the type of name in the current static env. and check e_i's with the function define parameter, the result's type is the type of body evaluate e₁, e₂, ..., e_n in the dynamic env. add those bindings evaluate Body in the dynamic env.

let 与 where 是将常量或表达式绑定到 name, name 在整个函数中皆可用相比, let 是将常量或表达式绑定到另一个表达式中, 所以 let 具有形式 let <bindings> in <expression> 如 let countUp n = let count start to = if start == to then to : [] else start : count (start+1) to in count 1 n let 可用于 introduce function 将常量, 表达式, 函数 定义绑定到 name 在 local scope 中, 在表达式外 #name 均不可用

在表达式中使用 name in count 1 n let 可用于生成序列, 如 [let square x = x * x in [square 5, square 3]] → [(25,9)] [bmi | (w,h) ← xs, let bmi = w/h^2]

在 GHCi 中通过 let 定义的函数和常量, 在整个环境中可用

product tuple 中, (Int, Int) 表示 "and", "each of", 也有在类型可以如 "One of", unions.
type import Data.Maybe, Maybe a = Just a / Nothing, isJust 函数可判断是否 Just

Haskell - P6

Case

类似于 C, C++ 中的 switch-case 结构, 用于多条并行分支的判断

case <expression> of <pattern> → <result>
<pattern> → <result>
<pattern> → <result>

与 C++ 的 switch-case 相比, 考虑 possible cases of value 相比, 可以同时进行 pattern matching
即可以用如下形式 head' :: [a] → a, head xs = case xs of [] → error

需要注意的是 case 采用逐个匹配的方式, 当出现与 expression 匹配的 pattern 时, 即返回 result
而且不具有 C, C++ 中的 default 条件, 如果出现未匹配的情况, 会抛出 runtime error

需要特别注意情况判断的完备性

recursion

由于 Haskell 中没有循环结构, 所以所有的循环实际上都是通过递归 (recursion) 实现的
与离散数学中关于递归的定义的形式类似, 需要 recursion definition 和 edge condition

定义 fibonacci 数列, f_n 为第 n 个 fibonacci 数

$f_0 = 0$ 基础步骤 $f_{n \leq 0} = 0$
 $f_1 = 1$ edge condition $f_{n=1} = 1$
 $f_n = f_{n-1} + f_{n-2}$ 递归步骤 recursion definition
n ∈ N ∧ n ≥ 2

其他用例:

$\text{zip}' :: [a] \rightarrow [b] \rightarrow [(a, b)]$ $\text{take}' :: (\text{Num } i, \text{Ord } i) \Rightarrow i \rightarrow [a] \rightarrow [a]$
 $\text{zip}' [] = []$ $\text{take}' n_{} = []$
 $\text{zip}' [] = []$ $\text{take}' _{} = []$
 $\text{zip}' (x:xs) (y:ys) = (x,y) : \text{zip}' xs ys$ $\text{take}' n (x:xs) = x : \text{take}' (n-1) xs$

$\text{repeat}' :: a \rightarrow [a]$

$\text{repeat}' x = x : \text{repeat}' x$

$\text{quicksort} :: (\text{Ord } a) \Rightarrow [a] \rightarrow [a]$

$\text{quicksort} [] = []$

$\text{quicksort} (x:xs) =$

$\text{let smallerSorted} = \text{quicksort} [a | a < x, a \in xs]$ 快速排序算法: $Q(s) =$

$\text{biggerSorted} = \text{quicksort} [a | a > x, a \in xs]$ $\{\emptyset, |S|=0\}$

$\text{in smallerSorted} ++ [x] ++ biggerSorted$ $(Q(s_l) \cup \{a\} \cup Q(s_u), |S| > 0)$

Haskell - P7

在 Haskell 中，所有函数只输出一个参数 (officially only takes one parameter)
输出多个参数的函数称为 curried function

如: $\max :: (\text{Ord } a) \Rightarrow a \rightarrow a \rightarrow a$ 等价于 $\max :: (\text{Ord } a) \Rightarrow a \rightarrow (a \rightarrow a)$
即 $\max 4 \cdot 5$ 等价于 $(\max 4) 5$

在此例中 \max 函数先 take 参数 4，此时形成的 $(\max 4)$ 依旧可视为一个函数
 $:t (\max 4) \rightarrow (\text{Ord } a, \text{Num } a) \Rightarrow a \rightarrow a$
来源于第一个参数对 Type 的限制 来源于 $a \rightarrow (a \rightarrow a)$ 的括号内部分

partially applied function, 即上例中的 $(\max 4)$ ，形式上看即为未完成参数传递函数

又如: $\text{compare} :: (\text{Ord } a) \Rightarrow a \rightarrow a \rightarrow \text{Ordering}$,
而 $:t (\text{compare } 100) \rightarrow (\text{Num } a, \text{Ord } a) \Rightarrow a \rightarrow \text{Ordering}$

Infix function 中缀函数，包括中缀表示的运算符或者可中缀表示的函数，也可用作 partially applied function

如 $(/10)$ 即形如 (parameter missed, 运算符或中缀表示的函数, parameter taken)

~~而~~ $\text{divByTen} = (/10)$ ，则 $\text{divByTen } 200$ 与 $(/10) 200$ 与 $200/10$ 等价

特别注意与 $(*10)$ 、 $(\text{subtract } 10) :: (\text{Num } a) \Rightarrow a \rightarrow a$ 不同， $(/10) :: (\text{Fractional } a) \Rightarrow a \rightarrow a$

higher Orderism 函数本身可作为参数传递给函数，在类型声明特别注意函数本身的类型作为一个整体置于正确位置

函数类型 如 $\text{applyTwice } f x = f(f x)$ 其类型声明为 $\text{applyTwice} :: (a \rightarrow a) \rightarrow a \rightarrow a$
f 的类型 即应有 $f :: a \rightarrow a$ f 的类型 x result

又如 $\text{zipWith}' = [a \rightarrow b \rightarrow c] \rightarrow [a] \rightarrow [b] \rightarrow [c]$

$\text{zipWith}' [] = []$ 其余两个参数的类型

注意当 $x \times y = (x, y)$ 时有 $f :: a \rightarrow b \rightarrow a, b$

$\text{zipWith}' -- [] = []$

此时该函数即成为标准的 Zip 函数

$\text{zipWith}' f (x:xs) (y:ys) = f x y : \text{zipWith}' f xs ys$

嵌套调用 $\text{zipWith}' (\text{zipWith}' (*)) [[1,2],[3,5]] [[3,2],[3,4]] \rightarrow [[3,4],[9,20]]$

此为 partially applied function 且 $(\text{zipWith}' (*)) :: (\text{Num } a) \Rightarrow [a] \rightarrow [a] \rightarrow [a]$

flip flip 函数的作用是调换参数函数中两个参数传递的川序。即有 $g(x,y) = f(y,x)$

$\text{flip}' :: (a \rightarrow b \rightarrow c) \rightarrow (b \rightarrow a \rightarrow c)$ $\text{flip} :: (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c$

$\text{flip}' f = g$ $\text{flip } f x y = f y x$

where $g x y = f y x$

注意左右的差异，左边为三个参数 而右边是调用一个函数

两者的类型实质上是等价的，但应注意 $(\text{flip}' f)$ 实际上是 flip 的一个 partially applied function
即 $(\text{flip}' f) x y \Leftrightarrow g x y \Leftrightarrow f y x \Leftrightarrow \text{flip } f x y$

Haskell - P8

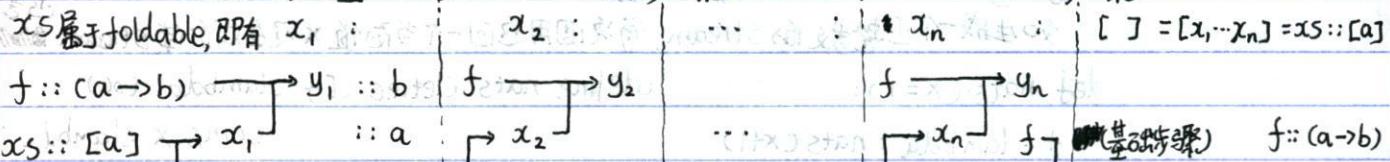
map: map 函数传入一个列表 [a] 和一个函数 (a → b), 将函数分别用于 [a] 的每个元素, 从而得到 [b]

即 $\text{map} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$, $\text{map} [] = []$, $\text{map} f (x : xs) = f x : \text{map} f xs$,

参数函数类型

基础步骤

递归步骤



得到的递归结果如 $y_1 : y_2 : \dots : y_n : [] = [y_1 \dots y_n] = ys :: [\beta]$

map 本质上与 list comprehension 是相同的, 即 $\text{map} f xs$ 与 $[f x | x \leftarrow xs]$ 是等价的

即 map 可描述为 $ys = \text{map} f xs \Leftrightarrow \forall i \in N . x_i \in xs \wedge y_i \in ys \wedge y_i = f(x_i)$

filter

与 map 函数类似, 但传入函数为 ($a \rightarrow \text{Bool}$), 以传入列表 [a] 中元素得到的 Bool 来区分是否存在于结果中

即 $\text{filter} :: (\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$, $\text{filter} [] = []$, $\text{filter} p (x : xs)$

参数函数类型

基础步骤, 递归步骤

$| px \Rightarrow x : \text{filter} p xs$

$\& qx \Rightarrow \text{filter} p xs$

$\text{quicksort} :: (\text{Ord } a) \rightarrow [a] \rightarrow [a]$

$\text{quicksort} [] = []$

$\&$

$\text{quicksort} (x : xs) = \dots$

与 map 类似, filter 本质上与 list comprehension 相同

$\text{let smallsort} = \text{quicksort} (\text{filter} (<= x) xs)$ 即 $\text{filter} p xs$ 与 $[x | x \leftarrow xs, p x]$ 是等价的

$\text{largesort} = \text{quicksort} (\text{filter} (> x) xs)$ 对 filter 进行嵌套调用等价于 list comprehension

$\text{in smallsort} ++ [x] ++ \text{largesort}$

条件的 &

takeWhile

takeWhile 与 filter 有相同的类型, 即 $\text{takeWhile} :: (\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$

但区别在于 filter 会对每个 list 中的元素进行判断, 但 takeWhile 会终止于第一个使函数为 False 的元素

即 filter 可描述为 $\text{for } x \text{ in } xs \text{ do } \{ \dots \}$, 而 takeWhile 则为 $\text{while}(p x_i) \{ \dots, i++ \}$

Collatz 序列

$\forall n \in \mathbb{Z}^+$ X_n 为奇数, 对于任何给定的奇数 X_1 , 有递归步聚 $X_{n+1} = \frac{3X_n + 1}{2^m}$, $\forall X_1$ 为奇数 $\exists n \in \mathbb{Z}^+ \frac{3X_n + 1}{2^m} = 1$

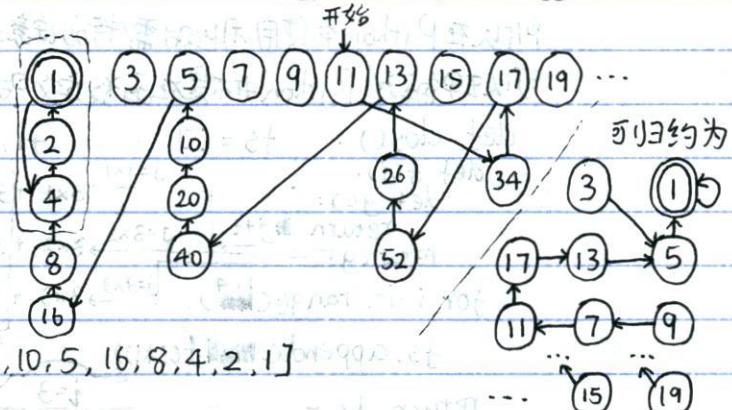
$\text{Chain} :: (\text{Integral } a) \Rightarrow a \rightarrow [\alpha]$

$\text{chain} n = [1]$

$| \text{even } n = n : \text{chain}(n / 2)$

$| \text{odd } n = n : \text{chain}(n * 3 + 1)$

$\text{chain} 11 \rightarrow [11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]$



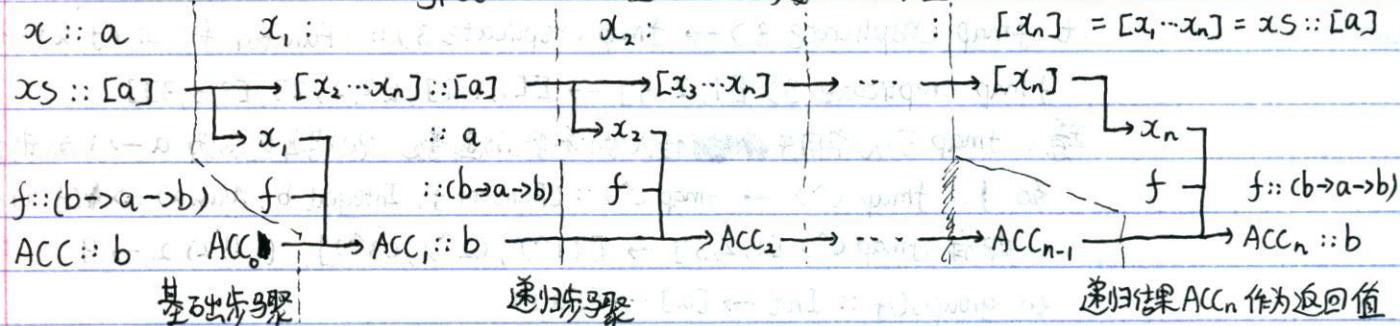
Haskell - P9

map 与 partially applied function
当向 map 传递了一个函数 $f :: a \rightarrow a \rightarrow a$ 和一个列表 $[a]$ 时, map 会返回一个函数的列表 $[(a \rightarrow a)]$
即当 map 向 $f :: a \rightarrow a \rightarrow a$ 依次传递 $[a]$ 中元素时, 生成的结果为 $[x :: a \rightarrow a]$.
如 $\text{map} (*) [1..5] \rightarrow [(1*) .. (5*)]$, 也即生成了一个 partially applied function 的列表
注意无法直接使用 partially applied function 列表与参数生成结果列表, 但可以单独使用某个元素
即可以用如下方法生成结果 $[m | m \leftarrow (\text{map} (*)) [1..3]], x \leftarrow [1..3] \rightarrow [1, 2, 3, 2, 4, 6, 3, 6, 9]$
且有 $:t \text{map} (*) [1..5] \rightarrow [(\text{Num } a, \text{Enum } a) \Rightarrow [a \rightarrow a]]$ partially applied function 列表
来源于 $[1..5]$ 的类型 要求了 a 必须是一个可列举的类型

lambda lambda 本质上是表达式(expression), 但返回的是一个函数所以 lambda 的使用既如表达式, 也如函数
lambda 具有基本形式 (parameters, \rightarrow function body) 是一个单独的 expression
可以一次性传递, 如 $\lambda x y \rightarrow$, 也可以分开传递 $\lambda x \rightarrow \lambda y \rightarrow$
注意 function body 部分可以使用 let-in 的结构来构造复杂的表达式结构.
但是 lambda 无法利用 pattern matching 来构造, 因为 lambda 在一个匹配失败后就会报错
另外对于 $\text{multiThree} :: (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a \rightarrow a$
 $\text{multiThree} = x \rightarrow y \rightarrow z \rightarrow (x * y * z)$] 两者的等价的, 只不过后一种反映了函数
 $\text{multiThree} = \lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow (x * y * z)$] 传递参数的过程, 与类型声明一致
lambda 的主要作用是生成即用即弃的函数并传递给 higher-order function, 如 map, filter

foldl foldl 通过传入一个函数 $f :: b \rightarrow a \rightarrow b$, 一个累加器 ACC_0 , 一个列表 $xs :: [a]$, 返回一个经过 n 次迭代的值
基础步骤: $\text{ACC}_0 = \text{ACC}_0 :: b$, 递归步骤: $\text{ACC}_n = f \text{ACC}_{n-1}, x_{n-1}$ 其中 $x_n \in xs$, ACC_n 为结果
 $:t \text{foldl} \rightarrow \text{foldl} :: (\text{Foldable } t) \Rightarrow (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow (t a) \rightarrow b$

Typeclasses 可折叠的 传入参数函数类型 包括但不限于 list



foldl 与 foldr 的区别仅在于 foldl 是从列表的最后一个元素开始迭代,

从应用上看, foldl 由于从头到尾的迭代顺序, 更易于理解, 适合用作数值计算与对顺序敏感的序迭代
而 foldr 更适用于由列表生成列表, 因为运算符 $(:) :: a \rightarrow [a] \rightarrow [a]$ 且添加至列表开头,
在这种用法中 foldr $(:)$ 与 foldl $(\text{flip}(:))$ 是等价的, 但应注意这种方法与 map 类似
但差别在于 map 生成的序列由单个元素的结果生成, 相互独立, 而 foldr 与每次迭代结果有关