

Haskell - P10

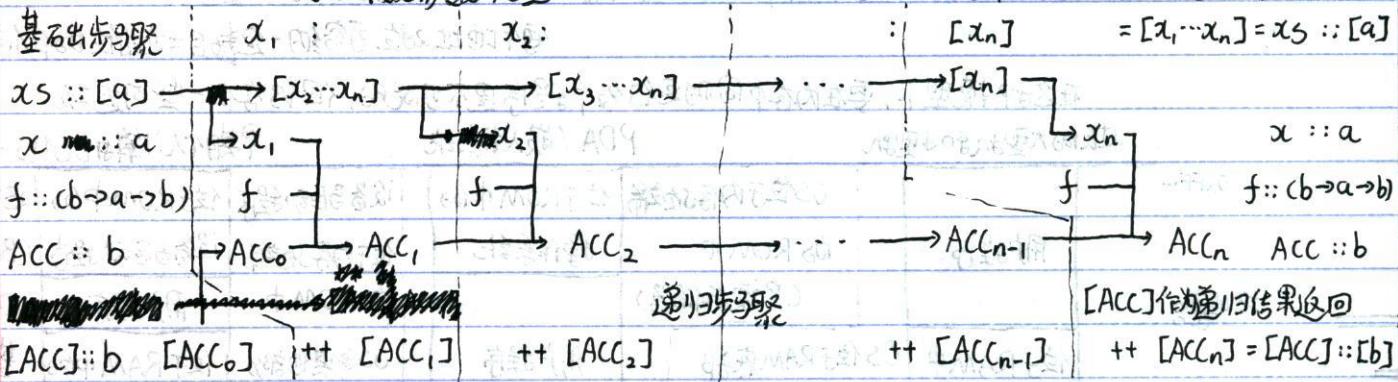
scanl

Scanl 与 foldl 类似，同样是对列表中的元素依次迭代计算，但 scanl 会保留每次的值并生成列表

scanr

即 $:t \text{ scanl} \rightarrow \text{scanl} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow [b]$ ，注意由于结果为列表，此外 $+a$ 也变为 $[a]$

基础步骤骤



通过 foldl 实现 scanl 的方法为： $\text{scanl}' f acc xs = \text{foldl} (\lambda x \rightarrow \text{last} (f (last (acc)) x)) [acc] xs$

scanr 与 foldr 类似，也是从最后一个元素向左迭代，但与 foldr 不同，scanr 的列表是自右向左生成的

即 $\text{scanl}' (+) 0 [1..10] \rightarrow [0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55]$ ，自左向右排列的迭代结果

~~而~~ $\text{scanr}' (+) 0 [1..10] \rightarrow [55, 54, 52, 49, 46, 43, 40, 34, 27, 19, 10, 0]$ ，迭代结果自右向左排列

scanl 可用于求序列 $\{a_n\}$ 的前 n 项和 S_n 的值，方法是生成 $n \rightarrow \{a_n\} \rightarrow \{S_n\}$ 的序列，即可进一步处理

\$

函数调用运算符 (function application)，表示将一个参数传递给函数并返回结果或 partially applied func

函数调用

$(\$) :: (a \rightarrow b) \rightarrow a \rightarrow b$ 但其与空格的区别在于 $\$ x$ 的运算优先级最高，而 $\$ \$ x$ 优先级最低

$f \$ x = f x$ 如 $\text{sqrt} 3 + 4 + 9$ 等价于 $(\text{sqrt} 3) + 4 + 9$ ，而 $\text{sqrt} \$ 3 + 4$ 等价于 $\text{sqrt} (3 + 4)$

另外通常调用函数是 left-associative (左结合) 的，即 $f a b c$ 等价于 $((f a) b) c$

而 \$ 运算符是 right-associative (右结合) 的，即 $f \$ g \$ h \$ x$ 等价于 $(f (g (h (x))))$

所以 \$ 可用于嵌套调用函数，以及调用 partially applied func 列表，如 $\text{map} (\$ x) [(1 *), \dots, (10 *)]$

复合函数运算符 (function composition)，与 \$ 有类似性的性质，但是针对的函数的复合

复合函数

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$ 用数学描述即 f 为定义在 $b \rightarrow c$ 上的函数， g 为定义在 $a \rightarrow b$ 上的函数

$f . g = \lambda x \rightarrow f (g x)$ (f . g)，或 (f \\$ g)，或 (f . g) 为定义在 $a \rightarrow c$ 上的函数

注意与数学中对复合函数的定义域-值域要求类似， $f . g$ 也要求满足两个函数的类型 检查

即 f 为从 $X \rightarrow Y$ 的函数，要求 g 为从 $X \rightarrow Y_0$ 的函数， f 为从 $Y \rightarrow Z$ 的函数且 $Y_0 \subseteq Y$ ，

而 $(f . g)$ 类型为 $a \rightarrow c$ 成立，要求 $g :: a \rightarrow b$ ， $f :: b \rightarrow c$ ，且 g 的输出 b 与 f 的输入 b 须一致

于是有 $(f (g (h (x))))$ 、 $(f . g . h) (x)$ 、 $f . g . h \ x$ 、 $f \$ g \$ h \$ x$ 都是相互等价的

另外注意 $f . g . h \ x$ 是右结合的，即 $f (g (h)) \ x$ 是等价的。

\$ 与 . 的组合使用可以将对单个参数的连续操作用 (partially applied func 1). (partially applied func 2). ... (partially applied func n) \$ x 的形式

Collatz

如 $\text{last} (\text{quickSort} (\text{map} (\text{chainStep} (\text{filter odd} [1..1000]))))$ 与 $\text{last}. \text{quickSort}. (\text{map} (\text{chainStep}). (\text{filter odd}) [1..1000])$ 是等价的

最大状态数

Haskell - P11

Prelude Module 默认载入的模块，包括基本的 functions, Type, Type class

import 模块可以通过 import < module name > 载入，如 import Data.List，其中包括了用于 List 的函数
在 ghci 中以 :m + name1 name2... 的形式载入

nub nub :: (Eq, a) => [a] -> [a]，用于清除 List 中重复的元素，如 nub [1, 1, 2, 3] -> [1, 2, 3]

import <module name> (name1, name2...) 用于从模块中载入子包函数

import <module name> hiding (name1, name2...) 用于载入模块时忽略其中的指定函数

name clash 当载入模块中的函数名与已有名称相同时，载入失败并报错，即已有同名函数

import qualified <module name> 用于解决 name clash

如 import qualified Data.Map，则其中的 filter 函数须通过 Data.Map.filter 进行调用

另外可以对模块指定名字，形如 import qualified <module name> as name，则可用 name 代替模块名
如 import qualified Data.Map as M，则可调用 M.filter 函数

intersperse :: a -> [a] -> [a] 在列表每两个元素之间，插入指定元素，如 intersperse 0 [1, 2, 3] -> [1, 0, 2, 0, 3]

intercalate :: [a] -> [[a]] -> [a] 将指定列表插入每两个元素列表之间并连接，如 intercalate " ." ["a", "b", "c"] -> "a.b.c"

transpose :: [[a]] -> [[a]] 将一个列表的列表视为矩阵进行转置，如 transpose [[1, 2, 3], [4, 5, 6], [7, 8, 9]] -> [[1, 4, 7], [2, 5, 8], [3, 6, 9]]

注意 transpose 可用于多项式的系数求和，升维（rank up）

foldl'/foldl' foldl 函数的 stricter version，用于在 foldl 处理 big list 产生 stack overflow 时代替 foldl

Concat :: (Foldable t) -> (t[a]) -> [a]，用于将在一个可折叠 t 中的 [a] 连接成一个 [a]，如 concat [[1, 2, 3], [2, 3, 4]] -> [1, 2, 3, 2, 3, 4]
注意 concat 的嵌套调用对应于多级 List，如 concat (concat [[[a]]]) -> [a]

concatMap :: (Foldable t) -> (a -> [b]) -> (t a) -> [b]，相当于一个 Map 函数和一个 concat 函数的组合
注意 其实际效果是先通过将 map 函数 (a -> [b]) 和列表 [a] 传入 Map 生成 [[b]]，再传入 concat 生成 [b]

splitAt :: Int -> [a] -> ([a], [a])，用于将一个列表在指定位置处切成两段，并返回成一个 tuple。
(pair)

注意 传入的 Int 不会引发类似 "Out of Range Error"，而是视为在位置 0 或位置 length - 1 处切开
此处 "为空" 如 splitAt 1 "hello" -> ("h", "ello") splitAt 0 "hello" -> ("", "hello") splitAt 5 "hello" -> ("", "hello")

Haskell - P12

- and :: (Foldable t) => (t Bool) -> Bool, 相当于 $\bigwedge_{i \in t} t_i$, 所以 and \$ map (>4) [4, 5, 6, 7] -> False
OR :: (Foldable t) => (t Bool) -> Bool, 相当于 $\bigvee_{i \in t} t_i$, 所以 OR \$ map (>4) [2, 3, 4, 5] -> True
- any :: (Foldable t) => (a -> Bool) -> (t a) -> Bool, 类似一个 ~~map f xs~~ 的结构
注意由于参数传递和递归的问题，不要使用如 (OR, map) 的结构，其实实际结果为 [Bool] 而非 Bool
- iterate :: (a -> a) -> a -> [a], 类似于递归定义序列 基础: a_0 , 递归: $a_{n+1} = f(a_n)$, f 为传入的参数函数 ($a -> a$)
特别注意 iterate 生成的是无限序列，使用时注意使用相关函数如 take 3 \$ iterate (*2) 1 -> [1, 2, 4]
而生成过程与绑定变量名不受影响是因为 Haskell 懒惰求值的特性，即使用时再给出表达式的求值
- dropWhile :: (a -> Bool) -> [a] -> [a], 与 takeWhile 类似，可描述为 while (~~px~~) { $x_s = x_s - \{x_i\}$, $i++$ }
Span :: (a -> Bool) -> [a] -> ([a], [a]), 与 takeWhile 有关，可看作将 takeWhile 的部分与剩余部分输出为一个 tuple
break :: (a -> Bool) -> [a] -> ([a], [a]), 在第一个使 px 为真的位置切断，与 span (not, p) 等价
- Sort :: (Ord a) => [a] -> [a], 对 [a] 按从大到小的顺序排序，如果是字符串则按 ASCII 字典顺序
group :: (Eq a) => [a] -> [[a]], 对 [a] 中的元素，将相邻且相等的元素合并成一个列表，从而形成 [[a]]
将 sort 和 group 可以计算列表中相同元素的个数，实现统计列表的功能。
- countList :: (Ord a) => [a] -> [(a, Int)], (countList [1, 1, 2, 3, 5, 8, 1, 3, 2, 1]) -> [(1, 4), (2, 2), (3, 2), (5, 1), (8, 1)]
countList xs = map (\l @ (x:xs) -> (x, length l)) group, sort, xs
表示 l 作为 (x:xs) 整体 表示复合函数 如果没有 \$ 则会破坏复合函数
- inits :: [a] -> [[[a]]], 输出所有可能的 take 结果，可描述为 $f(i=0, i < \text{length}[a], ++i)$ { [[a]] } = [[a]] ++ take i [a]
tails :: [a] -> [[[a]]], 输出所有可能的 drop 结果，可描述为 $f(i=0, i < \text{length}[a], ++i)$ { [[a]] } = [[a]] ++ drop i [a]
- isInfixOf :: (Eq a) => [a] -> [a] -> Bool, 用于检查某个列表是不是另一个列表的连续子列表，“bc” `isInfixOf` “abcd” -> True
isPrefixOf :: (Eq a) => [a] -> [a] -> Bool, 用于检查一个列表是否另一个列表的开头 注意对这三个函数检查空列表
isSuffixOf :: (Eq a) => [a] -> [a] -> Bool, 用于检查一个列表是否另一个列表的结尾 [] 会始终得到 True
- elem/notElem :: (Foldable t, Eq a) => a -> (t a) -> Bool, 用于检查一个元素是否在一个列表中。如 notElem 'a' "abc" -> False
- partition :: (a -> Bool) -> [a] -> ([a], [a]), 依据 px 的值 将 [a] 分为两个 [a] 并生成一个 tuple ([a], [a])
如 partition (>4) [1, 1, 2, 3, 5, 8, 1, 3, 2, 1] -> ([5, 8], [1, 1, 2, 3, 1, 3, 2, 1]) 可描述为 foreach a in [a]
- 注意其与 span 和 break 的区别在于，partition 类似于 map 需要对每元素进行检查， $p a \leftrightarrow a \in [a]_{\text{fst}}$
 $(p a) \leftrightarrow a \in [a]_{\text{snd}}$

Haskell - P13

Maybe value 表示一个特殊的类型，可以是 Just sth 或者 Nothing。例如 $\text{Maybe Int} = \text{Int} / \text{Nothing}$ 参考 Tree 的定义，可以将 Maybe 定义为 $\text{data Maybe a} = \text{Nothing} | \text{Just a}$ 。

Maybe 可以用于返回一个搜索或查找结果，使得将是否找到(T/F)和返回结果统一成一个返回结果

find

$:: (\text{Foldable t}) \Rightarrow (a \rightarrow \text{Bool}) \rightarrow (t a) \rightarrow \text{Maybe a}$ ，用于在一个可折叠的 ta 中查找，使 pa 为 True 的值
如 $\text{find } (\lambda x \rightarrow x > 4) [1, 1, 2, 3, 5] \rightarrow \text{Just } 5$ 返回 Nothing (未找到) 或 Just a (找到的首个值)

findIndex

$:: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Maybe Int}$ ，用于在一个列表 [a] 中查找值，并返回第一个下标或 Nothing

findIndices

$:: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [\text{Int}]$ ，用于在一个列表 [a] 中查找所有使 pa 为 True 的下标，并返回 [Int]
注意在以上函数中都需要传入一个参数函数类型为 $(a \rightarrow [\text{Bool}])$

elemIndex $:: (\text{Eq a}) \Rightarrow a \rightarrow [a] \rightarrow \text{Maybe Int}$ ，在列表 [a] 中查找一个值，并返回找到的首个下标或 Nothing

elemIndices $:: (\text{Eq a}) \Rightarrow a \rightarrow [a] \rightarrow [\text{Int}]$ ，在列表 [a] 中查找一个值，并返回所有下标的列表 [Int]

zip3 $:: [a] \rightarrow [b] \rightarrow [c] \rightarrow [(a, b, c)]$ ，用于将 3 个列表组装成 tuple (a, b, c)，以最少元素列表长度为准

zip4 $:: [a] \rightarrow [b] \rightarrow [c] \rightarrow [d] \rightarrow [(a, b, c, d)]$ ，与 zip3 相似，只是组装 4 个列表

zipWith3 (4) $:: ((a \rightarrow b \rightarrow c \rightarrow d) \rightarrow [a] \rightarrow [b] \rightarrow [c] \rightarrow [d])$ ，用传入的函数处理 3 个列表，生成一个结果列表
传入的参数函数类型：传入参数列表 [a] <-> 以最少元素列表长度为准

lines

$:: \text{String} \rightarrow [\text{String}]$ ，将传入的字符串中的 "\n" 删掉并切断，结果传入一个字符串列表 [String]

unlines

$:: [\text{String}] \rightarrow \text{String}$ ，将传入的字符串列表 [String] 用 "\n" 链接，返回一个字符串 String

unlines 与 intercalate "\n" 类似，但区别在于 unline 会在字符串最后多加一个 "\n"，"\n" 为换行符

words

$:: \text{String} \rightarrow [\text{String}]$ ，将传入的字符串在空格处切断，结果传入一个字符串列表 [String]

unwords

$:: [\text{String}] \rightarrow \text{String}$ ，将传入的字符串列表用空格连接为一个字符串，与 intercalate " " 等价

delete

$:: (\text{Eq a}) \Rightarrow a \rightarrow [a] \rightarrow [a]$ ，在列表 [a] 查找值，将第一个找到的元素删除，如 $\text{delete } 'a' "aabc" \rightarrow "abc"$

(\ \)

$:: (\text{Eq a}) \Rightarrow [a] \rightarrow [a] \rightarrow [a]$ ，对于传入第二个列表中的每个元素，从第一个列表中删除一个相同的元素

xs \\ ys

与 foldl (\ l x \rightarrow delete x l) ys xs 等价，如 "aabc" \\ "ba" \rightarrow "ac"

union

$:: (\text{Eq a}) \Rightarrow [a] \rightarrow [a] \rightarrow [a]$ ，取两个列表的并集，按照 $A \cup (B - A)$ 的顺序排列

intersect

$:: (\text{Eq a}) \Rightarrow [a] \rightarrow [a] \rightarrow [a]$ ，取两个列表的交集，按照在 A 中的顺序排列

union 与 xs ++ filter (\ notElem x) ys 等价，intersect 与 filter (\ elem y) xs ++ ys 等价

Haskell - P14

insert :: (Ord a) => a -> [a] -> [a], 将元素 x 插入列表中第一个大于等于 x 的元素之前.

genericXX :: length, take, drop, splitAt, !!, replicate 只能使用或返回 Int, genericXX 使其可以操作 Num
如 50 / clength [1..10] 会报错, 由于 (/) 使用 Fractional, 所以 50 / genericLength [1..10] -> 5.0
genericLength :: (Num i) => [a] -> i, genericIndex :: (Integral i) => [a] -> i -> a
genericTake :: (Integral i) => i -> [a] -> [a], genericDrop :: (Integral i) => i -> [a] -> [a]
genericSplitAt :: (Integral i) => i -> [a] -> ([a], [a]), genericReplicate :: (Integral i) => i -> a -> [a]

XX By
如 nub, delete, union, intersect, group 在执行判断时都是针对 (Eq), 即判断 $a == b$,
而 XX By 允许传入一个类型为 $(a -> a -> \text{Bool})$ 的函数作为参数, 并依据参数函数的值进行操作
如 groupBy ($\lambda x y \rightarrow (x < 4) == (y < 4)$) [1, 1, 2, 3, 5, 8, 1, 3, 2, 1] -> [[1, 1, 2, 3], [5, 8], [1, 3, 2, 1]]
group 与 groupBy ($==$) 是等价的, 同理可知其他 XX 与 XX By ($==$) 是等价的
nubBy :: $(a -> a -> \text{Bool}) -> [a] -> [a]$, groupBy :: $(a -> a -> \text{Bool}) -> [a] -> [[a]]$
unionBy :: $(a -> a -> \text{Bool}) -> [a] -> [a] -> [a]$, deleteBy :: $(a -> a -> \text{Bool}) -> a -> [a] -> [a]$
intersectBy :: $(a -> a -> \text{Bool}) -> [a] -> [a] -> [a]$
sortBy :: $(a -> a -> \text{Ordering}) -> [a] -> [a]$, insertBy :: $(a -> a -> \text{Ordering}) -> a -> [a] -> [a]$
maximumBy, minimumBy :: $(\text{Foldable } t) -> (a -> a -> \text{Ordering}) -> (t a) -> a$
类似的也可以传入一个 $(a -> a -> \text{Ordering})$ 的函数用于如 sortBy 的 XX By 函数, 主要是比较和排序
如 sort 与 sortBy compare 是等价的, 同理这部分 XX 与 XX By compare 是等价的

on :: $(b -> b -> c) -> (a -> b) -> a -> a -> c$, 用于将两个函数组合成一个函数, 位于 Data.Function
f(x,y) -> g(x) 传入的参数数, 实际作用如 $f(x_1, x_2) = f(g(x_1), g(x_2))$
传入的两个函数的类型 即有 $f \circ g = \lambda x y \rightarrow f(gx, gy)$
如 sortBy (compare `on` length) [1, 2, [3, 4, 5, 6], [], [7, 8, 9]] -> [[], [1, 2], [7, 8, 9], [3, 4, 5, 6]]

Function body of function is evaluated in the env. where the function is found (Statically Scope)
Evaluation not the env. where the function is called (Lexical Scoping)

Environment mapping of names to values (expressions, functions), maintain this map
Create Empty env. Create :: ? -> [] 这是程序如何实现将值绑定
Extend env. Extend :: (n, v) -> [(n, v)] -> E(n, v) 到变量
Apply env. Apply :: n -> v
Data Structure: a list of bindings | tuple: (name, value)

Haskell - P15

closure function definition with current environment ~~variable~~

lexical function a technique for implementing lexically scoped name binding in a language with first-class functions

operationally, a record storing a function together with an environment

Environment a mapping associating each free variable of the function with the value or reference to which the name was bound when the closure was created (variables that are used locally, but defined in an enclosing scope)

闭包 closure 包含自由(未绑定到特定对象)变量,这些变量不是在这个代码块或者全局上下文中定义,而是在定义代码块的环境中定义(局部变量) (作用域 scope)

closure 定义代码块的环境中定义(局部变量) (作用域, scope)
源于要执行的代码块(自由变量以及引用的对象未被释放)和为自由变量提供绑定的计算环境的结合

Data.Char 模块中为各类处理 character 的函数，同时与 map 与 filter 结合以处理 String ([char])

`xx::Char` 这类函数对传入的 `Char` 进行条件判断，并返回 `Bool`

\rightarrow Bool isControl : 是否为 control character (控制符), isSpace : 是否为空格 .

`isLower`: 是否小写字母, `isUpper`: 是否大写字母, `isAlpha`: 是否字母, `isAlphaNum`: 是否字母或数字

isPrint: 是否可打印的字符, 如 control character (控制符) 属于非打印符, 即打印(显示)时不出现
isDigit: 是否十进制数字; isAlphaDigit: 是否以大写字母开头; isHexDigit: 是否十六进制数字 (以16进制表示)

`isDigit`:是否十进制数字, `isOctDigit`:是否八进制数字, `isHexDigit`:是否十六进制数字。(以上数字皆一位)

`isLetter`: 是否字母，`isNumber`: 是否数字（注意无法辨认十六进制的A-F）

isMark: 是否 Unicode mark character, & check mark (✓)

isPunctuation : 是否 Punctuation, ASCII 中有 'p', '!', '#', '%', '&', '(', ')', '*', ',', '-','.', ':', ';' ?
'@', '[', '\', '=', '{'}'

`isSymbol`: 是否 fancy mathematical or currency symbol, ASCII中有 '\$', '+', '<', '=', '>', '^', ' π ', ' ∞ '

isSeparator: 是否 Unicode 中的 space 或 separator, isLatin1: 是否 Unicode 前 256 个字符

`isAscii`: 是否 Unicode 前 128 个字符, `isAsciiUpper`: 是否 Ascii 且是大写, `isAsciiLower`: 是否 Ascii 且是小写
构造与 words 等价的函数如 `filter (not . any isSpace)`, `groupBy (c ==)`on`isSpace)`, `"Hello world"`

筛选含有空格的串 将字符串拆分为空格串与无空格串 $\rightarrow ["Hello", "world"]$

General Categories

一种属于 Typeclass Eq 和 Enum 的类型，支持函数 generalCategory :: Char → GeneralCategory

General Category 类型源于 Unicode General Category，用于对字符进行通常分类。

`top`: UppercaseLetter, LowercaseLetter, MathSymbol, Space, Control, DecimalNumber

Haskell - P16

Tail Recursion (Runtime Stack) (continuation passing style)

Activation Records contains info necessary to execute a function call

① Static Allocation of A.R.

L ② Return Address

b) memory for each parameter

c) memory for each local variable

d) memory for return value

② Dynamic Allocation

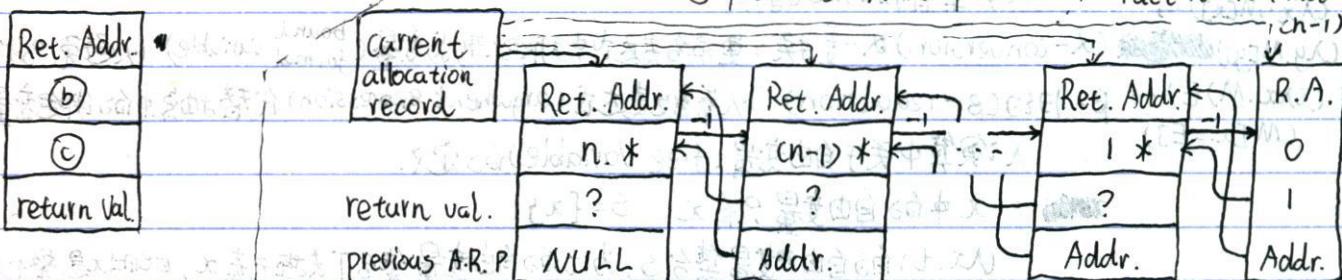
L ② Return Address

b) memory for parameters

c) memory for local vars

d) memory for return value, Fact 0 = 1

e) previous A.R. pointer, Fact n = n * Fact



考虑 $\text{Fact}' \bullet acc = acc$

$\text{Fact}' n acc = \text{Fact}' (n-1) (n * acc)$

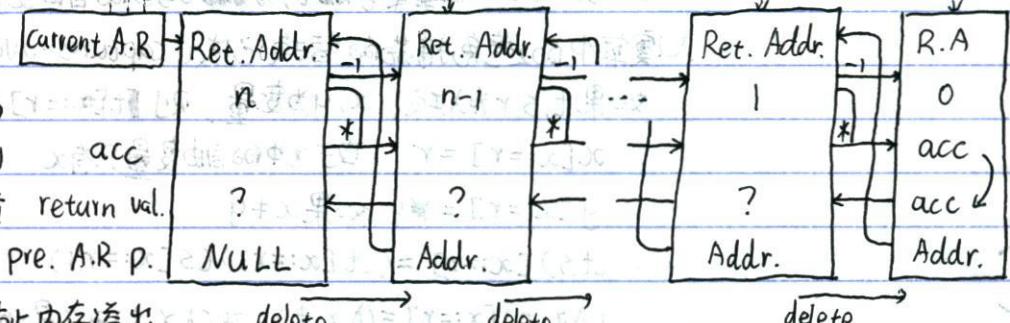
可以发现，在创建下一个调用时， acc

上一个调用已经完成了所有运算 $return val.$

而只需要等待结果返回，

所以可以释放上一个空间以防止内存溢出

($delete$)



Tail Position

① $fx = e$, then e is Tail Position

当一个表达式 e 的计算或调用一个函数位于一个

② when an expression is NOT Tail Position

函数返回结果之前，即计算或调用结果将直接用于

then ~~None~~ NONE of its subexp. are

当前函数返回，则称其位于尾位置 (Tail Position)

③ if e_1 then e_2 else $e_3 \Rightarrow$

如果是调用函数，则为尾调用 (Tail Call)

e_2, e_3 are in Tail Position

如果是调用自身，则为尾递归 (Tail Recursion)

④ let $band_1, band_2, \dots$ in e

也就是说，在尾递归中，电脑可以不用记忆多个调用的

e is in Tail Position

也就是说，在尾递归中，电脑可以不用记忆多个调用的

⑤ functions call parameters

返回位置，而是直接将参数传入当前函数堆栈结构

are NOT in Tail Position

(return position, stack frame) 的参数位置并返回当前函数的起始位置

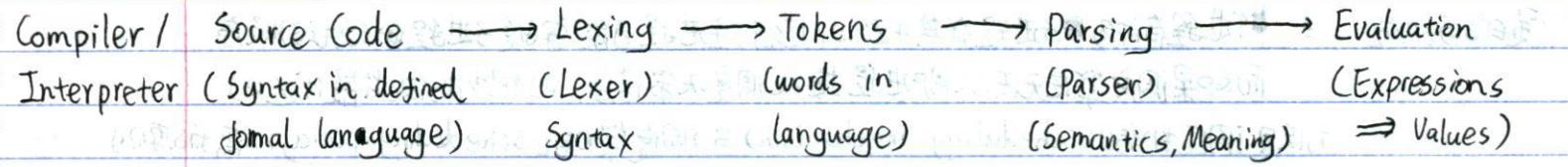
此为 Tail Call Elimination 或 Optimization, TCO

将堆栈空间需求从 $O(n)$ 降低到 $O(1)$

例如 $f(x) = x + s * x : x, s \text{ defined}, (x + s)^n \rightarrow x^n + s^n, ((x + s)^n - 1) / s$

或 $f(x) = x * s * x : x, s \text{ defined}, (x * s)^n \rightarrow x^n * s^n, ((x * s)^n - 1) / (s - 1)$

Haskell - P17



`toXX :: Char` `toUpper` 用于将字母变为大写字母, `toLower` 用于将字母变为小写字母
`→ Char` `toTitle` 用于将字符串变为 title case, 对于大多数来说等价于 uppercase

`digitToInt :: Char → Int` 和 `intToDigit :: Int → Char` 用于在字符串与数字字符之间转换,
`intToDigit` 如 `digitToInt 'A' → 10`, `intToDigit 15 → 'F'`, 注意转换按十六进制, 另外 `digitToInt` 接受大写

`ord :: Char → Int`] 这两个函数用于字符串与 corresponding number, 按照 Unicode 转换, 前 128 个与 ASCII 相同
`chr :: Int → Char`] 可以用于移位密码, 如 (Caesar cipher) 可写为 `map (chr . (+ shift)) . ord` message

Data.Map Data.Map 中的函数主要作用于字典, 即 `(key, value)` 为 pair (2 元素 Tuple), 注意与 Prelude 和 Data.List 的 name clash, 加载时应用形如 `import qualified Data.Map as Map` 的方式

`findKey :: (Eq k) => k → [C(k, v)] → Maybe v`, 用于在字典中与 `k` 绑定的 `v`, 如有则返回第一个 Just v, 否则返回 Nothing
`findKey key = foldr (\(k, v) acc → if key == k then Just v else acc)`, Nothing
接收传入的第一个参数, 如果 `key` 与当前 pair 的 `k` 相同, `acc` 更新为 `Just v`, 否则不变
`foldr` 与 `findKey key [] = Nothing` 注意两种实现的效果一样, 因为 `foldr` 是自右向左
Explicit recursion `findKey key (c(k, v) : xs) = if key == k then Just v else findKey key xs` 所以始终保持着更新到最左的一个符号条件结果
如果使用 `foldl` 则保留所有结果, 最后一个结果

`Map.fromList :: [(Ord k) => [C(k, v)]] → Map.Map k v`, 用于将一个 [pair] 转换为一个 Map 中的特殊类型 Map.Map
注意转化时: 1. 结果会按照 `k` 值的非序, 以及如果存在两个 pair 有相同的 `k`, 则取在 [pair] 中出现最新的

`Map.Map` 类型具有如下定义 `data Map = fromList [(k, v)]`

`Map.empty :: Map.Map k v` 用于生成一个空的 Map, 即 `fromList []`

`Map.insert :: (Ord k) => k → v → Map.Map k v → Map.Map k v` 将传入的 `k` 和 `v`, 插入传入的 Map.Map 中
如 `fromList` 函数可写为 `fromList' = foldl (\acc (k, v) → Map.insert k v acc) Map.empty`
但不能写为 `fromList' = foldr (\(k, v) acc → Map.insert k v acc) Map.empty`, 相同的 `key` 会保留 [pair] 中最新的