

第 7 讲: 程序设计范型

姓名: 林凡琪 学号: 211240042

评分: _____ 评阅: _____

2021 年 11 月 18 日

请独立完成作业, 不得抄袭。
若得到他人帮助, 请致谢。
若参考了其它资料, 请给出引用。
鼓励讨论, 但需独立书写解题过程。

- 函数式程序设计, 你值得拥有



1 作业 (必做部分)

题目 1 (Haskell 与函数式)

学习 Haskell 语言和函数式程序设计^①, 完成下列题目^②。

解答:

③

```
1 — 定义测试函数。你可以跳过阅读这一段程序。
2 test :: String -> Bool -> IO()
3 test description assertion = do
4   putStr description
5   case assertion of
6     False -> putStrLn " failed"
7     True  -> putStrLn " passed"
8
9 — 现在, 我们给出produce的定义。该函数返回 1 到 n 的列表。
10 — 示例: 当produce 10时, 返回[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
11
12 produce :: Int -> [Int]
13 produce n = [i | i <- [1..n]]
14
15 — 练习: 完成produceOdd函数, 其返回1~n中的奇数组成的 list
16 — 要求: 在produceOdd的定义中, 调用produce函数
17 — 示例: 当produceOdd 10时, 返回[1, 3, 5, 7, 9]
18
19 produceOdd :: Int -> [Int]
20 produceOdd n = [i | i <- [1..n], odd i]
21
22 testOdd = do
23   test "Odd 1" $ produceOdd 10 == [1,3,5,7,9]
```

① 推荐学习 [在线教程](#) 的第 2 章和第 5 章。

② 推荐使用 [在线平台](#) 运行程序。(免注册, 但请记得保存!)

③ 作业压缩包中含有上述程序的源代码 haskell1-sol.hs, 你可以将修改后的程序放入该文件中并与作业一起上传。

```

24 test "Odd 2" $ all odd $ produceOdd 100
25 test "Odd 3" $ (length $ produceOdd 100) == 50
26 test "Odd 4" $ (length $ produceOdd 99) == 50
27
28 — 练习: 完成produceList函数, 其接受一个参数n, 返回[[1],
    [1,2], ..., [1,...,n]]
29 — 要求: 在produceList的定义中, 调用produce函数
30 — 示例: 当produceList 3时, 返回[[1], [1, 2], [1, 2, 3]]
31
32 produceList :: Int -> [[Int]]
33 produceList n = [produce i | i <- produce n]
34
35 testList = do
36   test "List 1" $ produceList 1 == [[1]]
37   test "List 2" $ produceList 2 == [[1],[1,2]]
38   test "List 3" $ produceList 3 == [[1],[1,2],[1,2,3]]
39   test "List 4" $ (length $ produceList 99) == 99 && (length
    $ last $ produceList 99) == 99
40
41 — 练习: 完成produceStrangeList函数, 其接受一个参数n, 由其
    “后i个元素”和“前n-i个元素”组成的list(1 <= i <= n)
42 — 提示: ++ 可连接两个list
43 — 示例: 当produceStrangeList 5时, 返回
    [[2,3,4,5,1],[3,4,5,1,2],[4,5,1,2,3],[5,1,2,3,4],[1,2,3,4,5]]
44
45 produceStrangeList :: Int -> [[Int]]
46 produceStrangeList n = [drop i (produce n) ++ take i (produce
    n) | i <- produce n]
47
48 testStrangeList = do
49   test "StrangeList 1" $ produceStrangeList 1 == [[1]]
50   test "StrangeList 2" $ produceStrangeList 2 ==
    [[2,1],[1,2]]
51   test "StrangeList 3" $ produceStrangeList 3 ==
    [[2,3,1],[3,1,2],[1,2,3]]
52   test "StrangeList 4" $ produceStrangeList 5 ==
    [[2,3,4,5,1],[3,4,5,1,2],[4,5,1,2,3],[5,1,2,3,4],[1,2,3,4,5]]
53
54 — 现在, 我们给出produce的另一种定义方式。
55 — 需要注意的是, produce 是一个函数, 接受参数 n 指示生成序列
    的长度。
56 — produce2 是一个无限长度列表, 不需要参数。因此如果直接输出
    该列表将永无终止。
57 — 在需要时可以使用 take 函数获取该无限列表的前几项。
58 — 示例, 当take 5 (produce2)时, 返回构造出的produce序列的前5
    项, 为[1, 2, 3, 4, 5]
59
60 produce2 :: [Int]
61 produce2 = [1] ++ [(produce2!!i) + 1 | i <- [0..]]
62

```

```

63 — 练习: 请在理解 produce2 的基础上, 给出 factorial 的定义
64 — 示例输出: take 10 (factorial) 输出阶乘序列前10项
65 — [1,1,2,6,24,120,720,5040,40320,362880]
66
67 factorial :: [Int]
68 factorial = [1] ++ [(factorial!!i) * (i + 1) | i <- [0..]]
69
70 testFactorial = do
71   test "factorial 1" $ take 10 factorial ==
72     [1,1,2,6,24,120,720,5040,40320,362880]
73   test "factorial 2" $ factorial!!20 == 2432902008176640000
74
75 — 练习: 请在理解 produce2 的基础上, 给出 fibonacci 的定义
76 — 示例输出: take 10 (fibonacci) 输出斐波那契序列前10项
77 — [1,1,2,6,24,120,720,5040,40320,362880]
78
79 fibonacci :: [Integer]
80 fibonacci = [0, 1] ++ [(fibonacci !! i) + (fibonacci !! (i +
81   1)) | i <- [0..]]
82
83 testFibonacci = do
84   test "fibonacci 1" $ take 10 fibonacci ==
85     [0,1,1,2,3,5,8,13,21,34]
86   test "fibonacci 2" $ fibonacci!!60 == 1548008755920
87
88 — 计算机中常用的随机数是“伪随机数”, 其中的“伪”体现在给出的
89   “随机数”其实是通过一定的算法确定性的计算出来的。
90 — 换句话说, 如果仅靠随机函数自己, 每次返回的数字都会是同一个。
91 — 因此随机函数需要一个参数, 称为“随机数种子”, 为函数的输出
92   提供随机性。输入不同的“种子”, 生成不同的“随机数”。
93 — 以 LCG 随机算法为例, 其一种常见的定义如下。
94
95 random :: Integer -> Integer
96 random seed = (25214903917 * seed) `rem` 562949953421312
97
98 — 为了生成多个不同的“伪随机数”, 显然不能使用同一个随机数种子
99   作为输入。
100 — 在实践中, 常常使用上一个生成的随机数作为下一个随机数的种子。
101 — 即, 当随机数种子为 seed 时, 第一个随机数是 random seed, 第
    二个随机数是 random (random seed)。
102 — 练习: 请完成 randomList 的定义, 使得 randomList seed 能够
    返回一个无限长度的随机数列表
103
104 randomList :: Integer -> [Integer]
105 randomList seed = [random seed] ++ [random (randomList seed
106   !! i) | i <- [0..]]
107
108 testRandomList = do
109   test "randomList 1" $ (randomList 10)!!10 ==
110     357758112612178
111   test "randomList 2" $ (randomList 10)!!20 ==
112     218930212691682

```

```

102 test "randomList 3" $ (randomList 20)!!10 ==
    152566271803044
103
104
105 main = do
106     testOdd
107     testList
108     testStrangeList
109     testFactorial
110     testFibonacci
111     testRandomList

```

2 作业 (选做部分)

题目 1 (24 点)

下面的程序试图使用 Haskell 解决 24 点问题。得益于 Haskell 和函数式的强大表达能力，多数函数均只有一行。推荐在开始前学习 [教程](#) 第 4 章、第 6 章的内容。

解答：

④

```

1 import qualified Data.Set as Set
2 import Data.List (permutations)
3
4 — 下面的程序尝试使用 Haskell 求解 24 点。我们暂时不考虑程序
   效率。
5 — 核心思路非常简单：使用函数生成每一种可能的运算组合，直到得
   出结果。
6
7 — 为了能够输出最终的计算表达式，我们需要一种方式定义“运
   算”。
8 — 以下代码为框架代码，如果你暂时无法理解，可以先跳过这段程
   序。
9 data Exp = Value Double
10         | Add Exp Exp
11         | Sub Exp Exp
12         | Mul Exp Exp
13         | Div Exp Exp
14         deriving (Show, Eq, Ord)
15
16 — eval 函数接受一个 Exp 作为输入，递归计算该表达式并返回结
   果。
17 — 你可以试试看执行下面几个函数调用，想想返回值是怎么计算的：
18 — eval $ Value 10
19 — eval $ Add (Value 10) (Value 1)
20 — eval $ Mul (Value 10) (Sub (Value 3) (Value 2))
21 — eval $ Mul (Sub 10 6) (Sub (Value 3) (Value 2))
22 eval :: Exp -> Double
23 eval (Value v) = v
24 eval (Add a b) = eval a + eval b

```

④ 作业压缩包中含有上述程序的源代码 haskell2-sol.hs，你可以将修改后的程序放入该文件中并与作业一起上传。

```

25 eval (Sub a b) = eval a - eval b
26 eval (Mul a b) = eval a * eval b
27 eval (Div a b) = eval a / eval b
28
29 — 和 eval 类似, showExp 函数接受一个 Exp 作为输入, 返回该表
   达式的数学表示。
30 — 你可以试试看执行下面几个函数调用, 想想返回值是怎么计算的:
31 — showExp $ Value 10
32 — showExp $ Add (Value 10) (Value 1)
33 — showExp $ Mul (Value 10) (Sub (Value 3) (Value 2))
34 — showExp $ Mul (Sub 10 6) (Sub (Value 3) (Value 2))
35 showExp :: Exp -> String
36 showExp (Value v) = show $ round v
37 showExp (Add a b) = "(" ++ showExp a ++ "+" ++ showExp b ++ "
   )"
38 showExp (Sub a b) = "(" ++ showExp a ++ "-" ++ showExp b ++ "
   )"
39 showExp (Mul a b) = "(" ++ showExp a ++ "*" ++ showExp b ++ "
   )"
40 showExp (Div a b) = "(" ++ showExp a ++ "/" ++ showExp b ++ "
   )"
41
42 test :: String -> Bool -> IO()
43 test description assertion = do
44   putStr description
45   case assertion of
46     False -> putStrLn " failed"
47     True  -> putStrLn " passed"
48
49 unique xs = Set.toList $ Set.fromList xs
50
51 — 练习: 完成 buildOperations 的定义。
52 — 对于输入的表达式 a b, 应该返回所有形如 Op a b 的表达式。
53 — 其中, a b 分别为 xs ys 中给出的表达式, “Op” 为 Add/Sub/
   Mul/Div 中的任意一个。
54 — 如果不是很明白函数的作用, 你可以参考下面的样例测试。
55 — 另外, 这个函数的参考实现只有一行。
56 — 如果你的实现过于复杂, 推荐重新看看教程中的 List
   Comprehension 小节, 获取一些灵感。
57 buildOperations :: [Exp] -> [Exp] -> [Exp]
58 buildOperations xs ys = []
59
60 testBuildOperations = do
61   test "buildOperations 1" $ (Set.fromList $ unique $ map
     eval $ buildOperations [Value 1] [Value 2]) ==
62     Set.fromList [1+2, 1-2, 1*2, 1/2]
63   test "buildOperations 2" $ (Set.fromList $ unique $ map
     eval $ buildOperations [Value 1, Value 2] [Value 3]) ==
64     Set.fromList [1+3, 1-3, 1*3, 1/3, 2+3, 2-3, 2*3, 2/3]
65   test "buildOperations 3" $ (Set.fromList $ unique $ map
     eval $ buildOperations [Value 1, Value 2] [Value 3,
     Value 4]) ==

```

```

66   Set.fromList [1+3, 1-3, 1*3, 1/3, 2+3, 2-3, 2*3, 2/3,
67               1+4, 1-4, 1*4, 1/4, 2+4, 2-4, 2*4, 2/4]
68 — 练习: 完成 buildExpressions 的定义。
69 — 对于输入的表达式 a b c d, 利用 buildOperations 返回所有可
   能的表达式。
70 — 如 (a + b) * (c + d)。不需要交换 a b c d 的顺序。
71 — 提示: 你可以通过递归完成该函数。
72 — 例如, buildExpressions [1,2,3,4] =
73 —     buildOperations (buildExpressions [1]) (
74   buildExpressions [2,3,4])
75 — ++ buildOperations (buildExpressions [1,2]) (
76   buildExpressions [3,4])
77 — ++ buildOperations (buildExpressions [1,2,3]) (
78   buildExpressions [4])
79 — 其中, 多个 list 的合并可以使用 concat 函数。
80 — 推荐的定义形如
81 — buildExpression xs = concat [buildOperations ??? ??? | n
   <- [1..(length xs)-1] ]
82 — 你可以执行下面一行来查看你的 buildExpressions 函数的返回
   值:
83 — putStr $ unlines $ map showExp $ unique $ buildExpressions
   $ map Value [1,2,3]
84
85 buildExpressions :: [Exp] -> [Exp]
86 buildExpressions xs = []
87
88 testBuildExpressions = do
89   — buildExpress 对于 [1, 2] 应该能够返回如下的 4 种结果 (顺
   序不必相同)
90   test "buildExpressions 1" $ Set.fromList (map eval $
   buildExpressions [Value 1, Value 2]) ==
91   Set.fromList [1+2, 1-2, 1*2, 1/2]
92 — 按照规律, 对于 [1, 2, 3] 应该能够返回20种不同的结果。
93 test "buildExpressions 2" $ (length $ unique $ map eval $
   buildExpressions [Value 1, Value 2, Value 3]) == 20
94 test "buildExpressions 3" $ (length $ unique $ map eval $
   buildExpressions [Value 1, Value 2, Value 3, Value 4])
   == 104
95
96 — 练习: 完成 calc24 的定义。
97 — 利用上面定义的 buildExpressions 生成所有可能的表达式, 然后
   使用 eval 求解, 仅保留。
98 — 在生成表达式的过程中, 不必交换 a b c d 的顺序。
99 — 也即生成的表达式仅相当于在数字间插入 +-* / 和括号。
100 — 你可以使用下面这行代码查看你的 calc24 的返回值。
101 — putStr $ unlines $ map showExp $ unique $ calc24 $ map
   Value [1,2,3,4]
102
103 calc24 :: [Exp] -> [Exp]
104 calc24 xs = []

```

```

103 testCalc24 = do
104   test "calc24 1" $ sols [1,5,5,5] == 0
105   test "calc24 2" $ sols [5,5,1,5] > 0
106   test "calc24 3" $ sols [1,1,1,8] > 0
107   test "calc24 4" $ sols [1,8,1,1] == 0
108   test "calc24 5" $ sols [4,7,4,7] == 0
109   test "calc24 6" $ sols [4,4,7,7] > 0
110   test "calc24 7" $ sols [3,7,9,13] == 0
111   test "calc24 8" $ sols [2,2,13,13] > 0
112   test "calc24 9" $ sols [10,11,13,13] == 0
113   test "calc24 10" $ sols [1,3,10,13] == 0
114   test "calc24 11" $ sols [3,4,11,11] == 0
115
116   — 特殊测试: 5个数的24点, 你的程序理论上可以不做修改通过
117   test "calc24 12" $ sols [2,3,4,5,6] > 0
118   test "calc24 13" $ sols [4,5,6,7,8] > 0
119   — 计算解的个数
120   where sols xs = length $ calc24 $ map Value xs
121
122   — 如果我们想任意顺序的使用数字得到24点?
123   — 简单, 只需要对输入数字的每一种排列都调用一次写好的 calc24
      即可。
124   — 下面已给出参考实现。
125
126   calc24' :: [Exp] -> [Exp]
127   calc24' xs = concat [calc24 xs' | xs' <- permutations xs]
128
129   testCalc24' = do
130     test "calc24' 1" $ sols [1,5,5,5] > 0
131     test "calc24' 2" $ sols [5,5,1,5] > 0
132     test "calc24' 3" $ sols [1,1,1,8] > 0
133     test "calc24' 4" $ sols [1,8,1,1] > 0
134     test "calc24' 5" $ sols [4,7,4,7] > 0
135     test "calc24' 6" $ sols [1,6,11,13] > 0
136     test "calc24' 7" $ sols [3,7,9,13] > 0
137     test "calc24' 8" $ sols [2,2,13,13] > 0
138     test "calc24' 9" $ sols [10,11,13,13] == 0
139     test "calc24' 10" $ sols [1,3,10,13] == 0
140     test "calc24' 11" $ sols [3,4,11,11] == 0
141
142     — 特殊测试: 5个数的24点, 你的程序理论上可以不做修改通过
143     test "calc24' 12" $ sols [2,3,4,5,6] > 0
144     test "calc24' 13" $ sols [4,5,6,7,8] > 0
145     — 计算解的个数
146     where sols xs = length $ calc24' $ map Value xs
147
148   main = do
149     testBuildOperations
150     testBuildExpressions
151     testCalc24
152     testCalc24'
153     putStr $ unlines $ map showExp $ unique $ calc24' $ map

```

Value [1,5,5,5]

3 Open Topics

Open Topics 1 (Lambda Calculus)

请介绍 lambda 演算的历史和主要概念。

参考资料:

- [Lambda Calculus @ wiki](#)
- [Stanford Encyclopedia of Philosophy](#)

Open Topics 2 (函数式编程)

以 Haskell 为例, 请介绍函数式编程语言如何体现 Lambda Calculus 的主要概念。

参考资料:

- [Functional Programming @ wiki](#)

4 订正

5 反馈

你可以写 ^⑤:

- 对课程及教师的建议与意见
- 教材中不理解的内容
- 希望深入了解的内容
- ...

^⑤ 优先推荐 [ProblemOverflow](#)