第 7 讲: 程序设计范型

姓名: 马骏　　**学号:** 99149102

评分: _____　　**评阅:** _____

2021 年 11 月 9 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 。

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!)

解答:
③

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。

```haskell
-- 定义测试函数。你可以跳过阅读这一段程序。
test :: String -> Bool -> IO()
test description assertion = do
  putStr description
  case assertion of
    False -> putStrLn " failed"
    True  -> putStrLn " passed"

-- 现在，我们给出produce的定义。该函数返回 1 到 n 的列表。
-- 示例: 当produce 10时，返回[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

produce :: Int -> [Int]
produce n = [i | i <- [1..n]]

-- 练习: 完成produceOdd函数，其返回1–n中的奇数组成的list
-- 要求: 在produceOdd的定义中，调用produce函数
-- 示例: 当produceOdd 10时，返回[1, 3, 5, 7, 9]

produceOdd :: Int -> [Int]
produceOdd n = []

testOdd = do
```

```haskell
  test "Odd 1" $ produceOdd 10 == [1,3,5,7,9]
  test "Odd 2" $ all odd $ produceOdd 100
  test "Odd 3" $ (length $ produceOdd 100) == 50
  test "Odd 4" $ (length $ produceOdd 99) == 50

-- 练习: 完成produceList函数, 其接受一个参数n, 返回[[1], [1,2], ..., [1,...,n]]
-- 要求: 在produceList的定义中, 调用produce函数
-- 示例: 当produceList 3时, 返回[[1], [1, 2], [1, 2, 3]]

produceList :: Int -> [[Int]]
produceList n = []

testList = do
  test "List 1" $ produceList 1 == [[1]]
  test "List 2" $ produceList 2 == [[1],[1,2]]
  test "List 3" $ produceList 3 == [[1],[1,2],[1,2,3]]
  test "List 4" $ (length $ produceList 99) == 99 && (length $ last $ produceList 99) == 99

-- 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)
-- 提示: ++ 可连接两个list
-- 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]]


produceStrangeList :: Int -> [[Int]]
produceStrangeList n = []

testStrangeList = do
  test "StrangeList 1" $ produceStrangeList 1 == [[1]]
  test "StrangeList 2" $ produceStrangeList 2 == [[2,1],[1,2]]
  test "StrangeList 3" $ produceStrangeList 3 == [[2,3,1],[3,1,2],[1,2,3]]
  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]]


-- 现在, 我们给出produce的另一种定义方式。
-- 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。
-- 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。
-- 在需要时可以使用 take 函数获取该无限列表的前几项。
-- 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]

produce2 :: [Int]
produce2 = [1] ++ [(produce2 !! i) + 1  | i <- [0..]]

```

```haskell
-- 练习: 请在理解 produce2 的基础上, 给出 factorial 的定义
-- 示例输出: take 10 (factorial)输出阶乘序列前10项
-- [1,1,2,6,24,120,720,5040,40320,362880]

factorial :: [Int]
factorial = [0..]

testFactorial = do
  test "factorial 1" $ take 10 factorial ==
    [1,1,2,6,24,120,720,5040,40320,362880]
  test "factorial 2" $ factorial!!20 == 2432902008176640000

-- 练习: 请在理解 produce2 的基础上, 给出 fibonacci 的定义
-- 示例输出: take 10 (fibonacci)输出斐波那契序列前10项
-- [1,1,2,6,24,120,720,5040,40320,362880]

fibonacci :: [Integer]
fibonacci = [0..]

testFibonacci = do
  test "fibonacci 1" $ take 10 fibonacci ==
    [0,1,1,2,3,5,8,13,21,34]
  test "fibonacci 2" $ fibonacci!!60 == 1548008755920

-- 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“随机数”其实是通过一定的算法确定性的计算出来的。
-- 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。
-- 因此随机函数需要一个参数, 称为“随机数种子”, 为函数的输出提供随机性。输入不同的“种子”, 生成不同的“随机数”。
-- 以 LCG 随机算法为例, 其一种常见的定义如下。
random :: Integer -> Integer
random seed = (25214903917 * seed) `rem` 562949953421312

-- 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作为输入。
-- 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。
-- 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。
-- 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
randomList :: Integer -> [Integer]
randomList seed = [0..]

testRandomList = do
  test "randomList 1" $ (randomList 10)!!10 ==
    357758112612178
  test "randomList 2" $ (randomList 10)!!20 ==
    218930212691682
  test "randomList 3" $ (randomList 20)!!10 ==
    152566271803044
```

```haskell

main = do
  testOdd
  testList
  testStrangeList
  testFactorial
  testFibonacci
  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

```haskell
import qualified Data.Set as Set
import Data.List (permutations)

-- 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。
-- 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。

-- 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”。
-- 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。
data Exp = Value Double
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | Div Exp Exp
         deriving (Show, Eq, Ord)

-- 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。
-- 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:
-- eval $ Value 10
-- eval $ Add (Value 10) (Value 1)
-- eval $ Mul (Value 10) (Sub (Value 3) (Value 2))
-- eval $ Mul (Sub 10 6) (Sub (Value 3) (Value 2))
eval :: Exp -> Double
eval (Value v) = v
eval (Add a b) = eval a + eval b
eval (Sub a b) = eval a - eval b
eval (Mul 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。

```haskell
eval (Div a b) = eval a / eval b

-- 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。
-- 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:
-- showExp $ Value 10
-- showExp $ Add (Value 10) (Value 1)
-- showExp $ Mul (Value 10) (Sub (Value 3) (Value 2))
-- showExp $ Mul (Sub 10 6) (Sub (Value 3) (Value 2))
showExp :: Exp -> String
showExp (Value v) = show $ round v
showExp (Add a b) = "(" ++ showExp a ++ "+" ++ showExp b ++ ")"
showExp (Sub a b) = "(" ++ showExp a ++ "-" ++ showExp b ++ ")"
showExp (Mul a b) = "(" ++ showExp a ++ "*" ++ showExp b ++ ")"
showExp (Div a b) = "(" ++ showExp a ++ "/" ++ showExp b ++ ")"

test :: String -> Bool -> IO()
test description assertion = do
  putStr description
  case assertion of
    False -> putStrLn " failed"
    True  -> putStrLn " passed"

unique xs = Set.toList $ Set.fromList xs

-- 练习: 完成 buildOperations 的定义。
-- 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。
-- 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。
-- 如果不是很明白函数的作用, 你可以参考下面的样例测试。
-- 另外, 这个函数的参考实现只有一行。
-- 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。
buildOperations :: [Exp] -> [Exp] -> [Exp]
buildOperations xs ys = []

testBuildOperations = do
  test "buildOperations 1" $ (Set.fromList $ unique $ map
      eval $ buildOperations [Value 1] [Value 2]) ==
    Set.fromList [1+2, 1-2, 1*2, 1/2]
  test "buildOperations 2" $ (Set.fromList $ unique $ map
      eval $ buildOperations [Value 1, Value 2] [Value 3]) ==
    Set.fromList [1+3, 1-3, 1*3, 1/3, 2+3, 2-3, 2*3, 2/3]
  test "buildOperations 3" $ (Set.fromList $ unique $ map
      eval $ buildOperations [Value 1, Value 2] [Value 3,
      Value 4]) ==
    Set.fromList [1+3, 1-3, 1*3, 1/3, 2+3, 2-3, 2*3, 2/3,
       1+4, 1-4, 1*4, 1/4, 2+4, 2-4, 2*4, 2/4]
```

```haskell

-- 练习: 完成 buildExpressions 的定义。
-- 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。
-- 如 (a + b) * (c + d)。不需要交换 a b c d 的顺序。
-- 提示: 你可以通过递归完成该函数。
-- 例如, buildExpressions [1,2,3,4] =
--         buildOperations (buildExpressions [1]) (buildExpressions [2,3,4])
--      ++ buildOperations (buildExpressions [1,2]) (buildExpressions [3,4])
--      ++ buildOperations (buildExpressions [1,2,3]) (buildExpressions [4])
-- 其中, 多个 list 的合并可以使用 concat 函数。
-- 推荐的定义形如
-- buildExpression xs = concat [buildOperations ??? ??? | n <- [1..(length xs)-1] ]
-- 你可以执行下面一行来查看你的 buildExpressions 函数的返回值:
-- putStr $ unlines $ map showExp $ unique $ buildExpressions $ map Value [1,2,3]

buildExpressions :: [Exp] -> [Exp]
buildExpressions xs = []

testBuildExpressions = do
  -- buildExpress 对于 [1, 2] 应该能够返回如下的 4 种结果 (顺序不必相同)
  test "buildExpressions 1" $ Set.fromList (map eval $ buildExpressions [Value 1, Value 2]) ==
    Set.fromList [1+2, 1-2, 1*2, 1/2]
  -- 按照规律, 对于 [1, 2, 3] 应该能够返回20种不同的结果。
  test "buildExpressions 2" $ (length $ unique $ map eval $ buildExpressions [Value 1, Value 2, Value 3]) == 20
  test "buildExpressions 3" $ (length $ unique $ map eval $ buildExpressions [Value 1, Value 2, Value 3, Value 4])
    == 104

-- 练习: 完成 calc24 的定义。
-- 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。
-- 在生成表达式的过程中, 不必交换 a b c d 的顺序。
-- 也即生成的表达式仅相当于在数字间插入 +-*/ 和括号。
-- 你可以使用下面这行代码查看你的 calc24 的返回值。
-- putStr $ unlines $ map showExp $ unique $ calc24 $ map Value [1,2,3,4]

calc24 :: [Exp] -> [Exp]
calc24 xs = []

testCalc24 = do
  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 :

⑤ 优先推荐 ProblemOverflow

- 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
- ...