第六章作业：目录

- **第七次Python上机作业**

  自行安排，无需提交

第七次上机作业

- **整理BinaryTree类定义的代码，包括课件中提到的遍历、__str__等常用方法。**
- **利用BinaryTree类实现逻辑表达式的处理，包括如下处理函数：**

  buildParseTree: 建立逻辑表达式解析树;

  evaluate: 逻辑表达式求值;

  printExp: 输出中缀表达式（去除不必要的括号）。

  注意：需要处理的逻辑表达式不是全括号形式，操作数为True/False，操作符为'and', 'or', 'not'，优先级为：括号>not>and>or，单词仍然是用空格隔开。

  如：not (True or False) and (False or True and True)

第七次作业参考

- **BinaryTree的__str__**

  递归调用！

```
def __str__(self):
    return '[%s,%s,%s]' % (self.getRootVal(), \
                           self.getLeftChild(), self.getRightChild())
__repr__= __str__
```

- **buildParseTree的思路**

  参考栈中的中缀表达式直接求值的作业

  或者参考栈课件中的中缀表达式转后缀表达式，以及后缀表达式求值的结合

  设置两个栈

  - 一个栈用于保存操作符，处理不完全括号中的优先级情况
  - 另一个栈用于保存操作数，参考求值的过程同步建立解析树

结果

```
>>>
EXPR: True or False and ( not False or True )
TREE: [or,[True,None,None],[and,[False,None,None],[or,[not,None,[False,None,None
]],[True,None,None]]]]

EXPR: True or False and False or True
TREE: [or,[or,[True,None,None],[and,[False,None,None],[False,None,None]]],[True,
None,None]]

EXPR: not True or False
TREE: [or,[not,None,[True,None,None]],[False,None,None]]

EXPR: ( not True )
TREE: [not,None,[True,None,None]]
```

代码


```
def buildParseTree(infixexpr):
    def makeTree(op): #根据操作符来构造解析树
        eTree= BinaryTree(op)
        if op in op2List: #双目运算符，从操作数栈中弹出两个子树，先出为右
            eTreeRight= operandStack.pop()
            eTreeLeft= operandStack.pop()
            eTree.setLeft(eTreeLeft)
            eTree.setRight(eTreeRight)
        elif op in op1List: #单目运算符，从操作数栈中弹出一个子树，作为右子树
            eTreeRight= operandStack.pop()
            eTree.setLeft(None)
            eTree.setRight(eTreeRight)

        operandStack.push(eTree) #将生成的子树压回操作数栈

    prec= {'not':4, 'and':3, 'or':2, '(':1}
    constList= ['True', 'False']
    op1List= ['not']
    op2List= ['and', 'or']

    opStack= Stack() #操作符栈，栈中元素为操作符in op1List, op2List
    operandStack= Stack() #操作数栈，栈中元素为BinaryTree树根
```

代码


```
tokenList= infixexpr.split()
for token in tokenList:
    if token== '(':
        opStack.push(token)

    elif token== ')': #上升，直到左括号的节点
        topToken= opStack.pop()
        while topToken!= '(':
            makeTree(topToken)
            topToken= opStack.pop()

    elif token in constList:
        eTree= BinaryTree(eval(token))
        operandStack.push(eTree)

    elif token in (op1List+ op2List):
        while (not opStack.isEmpty()) and \
              (prec[opStack.peek()]>= prec[token]):
            topToken= opStack.pop()
            makeTree(topToken)
        opStack.push(token)

while not opStack.isEmpty():
    topToken= opStack.pop()
    makeTree(topToken)

return operandStack.pop()
```