数据结构第 3 次上机实验报告

林子开

2023 年 9 月 19 日

目录

1 中序表达式到后序表达式的转换

1.1 python 代码

Listing 1: Postfix evaluator, python 源码

```python
from pythonds.basic import Stack  # 需要在终端输入 python -m pip install pythonds 进行安装

def parenthesis_checker(parenthesis):
    s = Stack()
    for par in parenthesis:
        if par == '(':
            s.push(par)
        else: # par == ')'
            if s.isEmpty():
                return False
            temp = s.pop()
            if temp == ')':
                return False
    return s.isEmpty() # 最终栈内应该无括号剩余

def infix_to_postfix(infix_expression):
    """本函数能够检查括号是否匹配，并提供 +加法、-减法、*乘法、/除法、mod 取余数、^求幂运算
    输入：一个中序表达式，要求所有的运算符和运算数之间都用空格间隔，
        例如A + B mod 5 * 8，并要求幂指数部分必须使用括号以示分界，例如A ^ ( 4 + b )
    输出：一个postfix表达式，其运算优先级为 level(+, -) < level(*, /, mod) < level(^)"""
    tokens = infix_expression.split()
    parenthesis = []
    for token in tokens:
```

```
        if token == '(' or token == ')':
            parenthesis.append(token)
    if not parenthesis_checker(parenthesis):  # 如果括号不匹配
        print('表达式的括号匹配有误！')
        return False
    level = {'(':0, ')':0, '+':1,'-':1,'*':2,'/':2,'mod':2,'^':3} # 使用字典存储各个运算符的优先顺序，其中左括号的优先级为0，以保证括号内的运算符能全都入栈
    ops =  set(level.keys()) # 所有的运算符
    op_stack = Stack()
    postfix_expression = []
    for token in tokens:
        if token not in ops: # 如果是数字
            postfix_expression.append(token)
        elif token == '(': # 左括号不进行优先级比较，直接入栈
            op_stack.push(token)
        elif token == ')':
            temp = op_stack.pop()
            while temp != '(': # 遇到右括号则不断弹出栈内运算符直到遇见左括号
                postfix_expression.append(temp)
                temp = op_stack.pop()
        else: # 除了左右括号之外的其他运算符
            while (not op_stack.isEmpty()) and (level[op_stack.peek()] >= level[token]): # 当栈还未空时，要不断检查栈顶的运算符优先级
                postfix_expression.append(op_stack.pop()) # 如果栈顶运算符大于等于token，则要把栈顶运算符弹出
            op_stack.push(token)
    while not op_stack.isEmpty(): # 表达式全部读完，把栈内剩余的运算符全部弹出
        postfix_expression.append(op_stack.pop())
    return ' '.join(postfix_expression)     # 返回的逆波兰式用空格连接


def main():
    while True:
        in_s =  input('请输入中序表达式: ')
        pos_s = infix_to_postfix(in_s)
        print(pos_s)

if __name__ == '__main__':
    main()
```

1.2 中序表达式到后序表达式的实验结果

1. Infix:(A + B) * C

   Postfix:A B + C *

2. Infix:A + (B - C)

   Postfix:A B C - +

3. Infix:A * (B + C) / D

Postfix:A B C + * D /

4. Infix:(A + B) * (C - D)

   Postfix:A B + C D - *

5. Infix: A + B * C - D / E

   Postfix:A B C * + D E / -

6. Infix:(A * B) + (C / D) - E

   Postfix:A B * C D / + E -

7. Infix:(A + B) / (C + D) * E

   Postfix:A B + C D + / E *

8. Infix: A * (B + C) - (D * E)

   Postfix:A B C + * D E * -

9. Infix:(A + B) * (C - D) / (E + F)

   Postfix:A B + C D - * E F + /

10. Infix: A * (B + (C * (D - (E / (F + (G * H)))))) / I

    Postfix:A B C D E F G H * + / - * + * I /

2 后序表达式运算器

2.1 python 代码

Listing 2: infix to postfix conversion, python 源码

```python
from pythonds.basic import Stack

def op_calculate(op1,op2,operator):
    if operator == '+':
        return op1 + op2
    if operator == '-':
        return op1 - op2
    if operator == '*':
        return op1 * op2
    if operator == '/':
        return op1 // op2 # 只进行整数除法


def postfix_calculator(postfix_expression):
    tokens = postfix_expression.split()
    numStack = Stack()
    operators = ['+','-','*','/']
    for token in tokens:
        if token not in operators: # token是数字
            numStack.push( int(token))
```

```
        else: # token是运算符
            op2 = numStack.pop()
            op1 = numStack.pop()
            numStack.push(op_calculate(op1,op2,token))
    return numStack.pop()

def main():
    while True:
        s =  input('请输入后序表达式: ')
         input(postfix_calculator(s))


if __name__ == '__main__':
    main()
```

2.2 后序表达式运算器的实验结果

1. Infix: 3 5 + 2 7 * /

   Result: 0

2. Infix: 25 5 + 3 * 21 7 / 1 + -

   Result: 86

3. Infix: 5 1 2 + 4 * + 3 - 7 4 5 - + +

   Result: 20

4. Infix: 36 3 / 5 + 2 * 14 - 3 * 24 2 / 1 - + 2 /

   Result: 35

5. Infix: 20 4 - 2 * 14 + 7 / 1 - 5 * 9 + 12 3 / 2 + -

   Result: 28

6. Infix: 10 5 + 2 * 8 - 4 / 3 + 6 * 12 2 * 4 + - 18 3 / 2 * + 5 -

   Result: 27

7. Infix: 24 3 / 6 + 2 * 14 - 2 / 5 + 4 * 16 2 * 3 + - 21 3 / 2 * +

   Result: 27

8. Infix: 20 6 + 2 * 14 - 7 / 1 + 4 * 10 2 * 3 + - 27 3 / 2 * + 4 -

   Result: 15

9. Infix: 8 4 + 3 * 18 - 2 / 7 + 5 * 10 - 2 + 4 * 12 - 6 + 2 * 3 - 2 /

   Result: 280

10. Infix: 36 4 / 7 + 2 * 14 - 2 / 6 + 3 * 12 - 5 + 4 * 16 - 8 + 2 / 5 + 2 3 * - 7 1 / +

    Result: 78