ANSWERS

A6.6 Reverse Polish notation

Theory of computation · A level · OCR H446 2.3.1, AQA 7517 4.3.3.1 · about 25 min

BugBotLab

What this lesson is about

Infix and postfix, converting both ways, evaluating RPN with a stack, and the shunting-yard algorithm.

- Infix and postfix
- Why and where RPN is used
- Converting infix to RPN by hand
- Evaluating RPN with a stack
- Converting infix to RPN with a program

Questions

6 marks in all

1. Convert (4 + 6) × 3 − 2 to Reverse Polish notation. Use * for ×, and separate tokens with spaces. [1 mark]

> **Answer:** 4 6 + 3 * 2 -. Bracket it as (((4 + 6) × 3) − 2) and move each operator after its operands.

2. Evaluate the RPN expression 7 2 3 * - 4 + [1 mark]

> **Answer:** 5. Push 7, 2, 3; * gives 6; - gives 7 - 6 = 1; push 4; + gives 5.

3. When evaluating RPN with a stack, an operator pops two values. For 9 3 -, which is the right-hand operand? [1 mark]

- ☐ **A** 9, the first popped
- ☐ **B** 3, the first popped
- ☐ **C** 9, the second popped
- ☐ **D** 3, the second popped

> **Answer:** B. 3 is on top, so it is popped first and becomes the right-hand operand: 9 - 3 = 6.

4. Why is Reverse Polish notation used? [1 mark]

Tick every answer that is true.

- ☐ **A** It needs no brackets
- ☐ **B** It needs no operator precedence rules
- ☐ **C** It can be evaluated left to right with a stack
- ☐ **D** It uses fewer operands than infix

> **Answer:** A, B, C. RPN has the same operands as infix. Its advantage is that a stack machine can evaluate it simply, which is why interpreters and bytecode use it.

5. Convert the RPN expression 5 1 2 + 4 * + to infix, with only the brackets that are needed. Use * for ×. [1 mark]

> **Answer:** 5 + (1 + 2) * 4. 1 2 + is (1 + 2); times 4 gives (1 + 2) * 4; then 5 + that.

6. What does this program print? [1 mark]

```
stack = []
for token in "3 4 2 * +".split():
    if token in "+*":
        right = stack.pop()
        left = stack.pop()
        stack.append(left + right if token == "+" else left * right)
    else:
        stack.append(int(token))
    print(stack)
```

Answer:

```
[3]
[3, 4]
[3, 4, 2]
[3, 8]
[11]
```

The stack grows to [3, 4, 2], * replaces 4 and 2 with 8, and + replaces 3 and 8 with 11.

The task: an RPN calculator

Write both halves of a calculator for expressions with whole numbers, `+`, `-`, `*`, `/` and brackets. In every expression, tokens are separated by single spaces, including the brackets. - `to_rpn(infix)` takes an infix string such as `"( 3 + 4 ) * 2"` and returns the RPN string with single spaces, such as `"3 4 + 2 *"`. Use the shunting-yard algorithm with a stack. All four operators work left to right. - `evaluate(rpn)` takes an RPN string and returns its value, using a list as a stack. `/` is ordinary division; every test divides exactly. - Do not use Python's `eval`. - For each expression in `tests`, in order, print the infix, `->`, the RPN, `=`, then the value as a whole number. For example `3 + 4 * 2 -> 3 4 2 * + = 11`. Four lines in all.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

tests = ["3 + 4 * 2", "( 3 + 4 ) * 2", "( 5 - 1 ) * ( 2 + 6 ) / 4", "8 - 2 - 3"]

def to_rpn(infix):
    output = []
    stack = []
    return ""
```

The hint students can ask for: For to_rpn, follow the five steps of the shunting-yard algorithm, giving * and / a bigger precedence number than + and -. Remember an opening bracket on the stack must stop the popping in step 2. For evaluate, the first item popped is the right-hand operand. Build each output line from the two function results.

A solution

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

tests = ["3 + 4 * 2", "( 3 + 4 ) * 2", "( 5 - 1 ) * ( 2 + 6 ) / 4", "8 - 2 - 3"]
PRECEDENCE = {"+": 1, "-": 1, "*": 2, "/": 2}

def to_rpn(infix):
    output = []
    stack = []
    for token in infix.split():
        if token in PRECEDENCE:
            while stack and stack[-1] in PRECEDENCE and PRECEDENCE[stack[-1]] >=
PRECEDENCE[token]:
                output.append(stack.pop())
            stack.append(token)
        elif token == "(":
            stack.append(token)
        elif token == ")":
            while stack[-1] != "(":
                output.append(stack.pop())
            stack.pop()
        else:
            output.append(token)
    while stack:
        output.append(stack.pop())
    return " ".join(output)

def evaluate(rpn):
```

```
    stack = []
    for token in rpn.split():
        if token in PRECEDENCE:
            right = stack.pop()
            left = stack.pop()
            if token == "+":
                stack.append(left + right)
            elif token == "-":
                stack.append(left - right)
            elif token == "*":
                stack.append(left * right)
            else:
                stack.append(left / right)
        else:
            stack.append(int(token))
    return stack.pop()

for infix in tests:
    postfix = to_rpn(infix)
    print(infix, "->", postfix, "=", int(evaluate(postfix)))
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.