# Infix-, Präfix und Postfix-Ausdrücke L+ Theorie

### Darstellung binärer Operationen

Eine *binäre Operation* ist eine Operation, die *zwei* Operanden zu einem Ausdruck verknüpft.

Wir sind uns seit Beginn der Schulzeit gewohnt, einen binären Operator *zwischen* seine beiden Operanden zu schreiben. Wir werden sehen, dass es auch noch andere Darstellungen gibt, welche sich für die Verarbeitung mit Computern besser eignen.

- *Infix*-Notation: Operator steht *zwischen* Operanden (`A + B`)
- *Präfix*-Notation: Operator steht *vor* Operanden (`+ A B`)
- *Postfix*-Notation: Operator steht *nach* Operanden (`A B +`)

### Rechenregeln

- Klammern haben höchste Priorität
- Multiplikation/Division vor Addition/Subtraktion
- Operationen gleicher Stufe: von links nach rechts auswerten

### Beispiel 1

Kleinere Ausdrücke lassen sich mit etwas Übung im Kopf konvertieren (umwandeln):

| | **Infix-Ausdruck** | **Präfix-Ausdruck** | **Postfix-Ausdruck** |
|---|---|---|---|
| (a) | `A - B + C` | `+ - A B C` | `A B - C +` |
| (b) | `A - (B + C)` | `- A + B C` | `A B C + -` |
| (c) | `A + B * C` | `+ A * B C` | `A B C * +` |
| (d) | `(A + B) * C` | `* + A B C` | `A B + C *` |

Zwei wichtige Beobachtungen

- Reihenfolge der Operanden: überall gleich
- Klammern: entfallen bei Präfix und Postfix

### Aufgabe 1

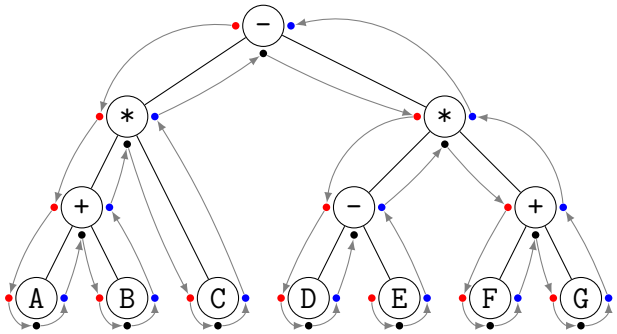
| | **Infix** | **Präfix** | **Postfix** |
|---|---|---|---|
| (a) | `A-(B+C)-D` | `- - A + B C D` | `A - B C + D -` |
| (b) | `A+B*C+D` | `+ + A * B C D` | `A B C * + D +` |
| (c) | `(A-B)*(C+D)` | `* A B - C D +` | `A B - C D + *` |
| (d) | `A-(B-(C-D))` | `- A - B - C D` | `A B C D - - -` |

### Binärbäume

Komplexere Infix-Ausdrücke können wir mit Hilfe gewurzelter Binärbäume in die Präoder Postfix-Darstellung konvertieren.

### Beispiel 2

`(A + B) * C - (D - E) * (F + G).`



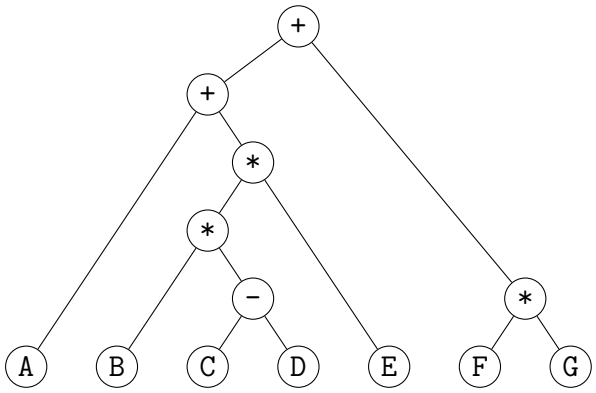

Starte in der Wurzel und „umfahre“ den Baum entlang der Pfeile.

Präfix (rote Punkte): `- * + A B C * - D E + F G`

Postfix (blaue Punkte): `A B + C * D E - F G + * -`

### Aufgabe 2

Wandle den Term `A + B * (C - D) * E + F * G` mit Hilfe eines Binärbaums in die Präfix- und die Postfix-Darstellung um.




Präfix: `+ + A * * B - C D E * F G`

Postfix: `A B C D - * E * + F G * +`

### Übersicht

Nachdem wir nun grundsätzlich wissen, wie man Infix-Ausdrücke in die Präfix- und Postfix-Darstellung verwandelt, konzentrieren wir uns von jetzt an auf die Postfix-Darstellung. Das Ziel ist es, jeweils ein Programm zu schreiben, das

(a) Infix-Strings in Postfix-Strings umwandelt und

(b) einen Postfix-String evaluiert (auswertet).

Da wir gerade Stacks (und keine Bäume) behandeln, werden wir den Algorithmus für (a) auf Grundlage eines Stacks formulieren. Auch für (b) verwenden wir einen Stack.

### Algorithmus Infix-to-Postfix

1. Erstelle einen leeren Stack für die Operatoren und eine leere Liste für die Ausgabe.
2. Tokenisiere (zerlege) die Infix-Zeichenkette in eine Liste von Operanden, Klammern und Operatoren.
3. Durchlaufe die Tokenliste von links nach rechts.
   (a) Ist das Token ein Operand, setze ihn ans Ende der Ausgabeliste.
   (b) Ist das Token eine linke Klammer, setze sie auf den Operatorstack.
   (c) Ist das Token eine rechte Klammer, entferne so lange Operanden vom Operandenstack und setze sie ans Ende der Ausgabeliste, bis die zugehörige linke Klammer erscheint und entferne auch sie vom Stack.
   (d) Ist das Token ein Operator, nimm so lange Operatoren vom Stack, wie sie höhere oder gleiche Priorität haben, setze sie ans Ende der Ausgabeliste und lege das Token auf dem Stack ab.
4. Sind noch Operatoren auf dem Stack, entferne sie der Reihe nach mit einer pop-Operationen vom Stack und setze sie ans Ende der Ausgabeliste.

### Beispiel 3

`Input:`

| ( | A | + | B | ) | * | C | - | ( | D | - | E | ) | * | ( | F | + | G | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

`Stack: (nimmt → zu)`

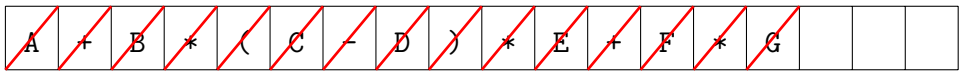
| ( | + | ) | * | - | ( | - | ) | * | ( | + | ) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

`Output:`

| A | B | + | C | * | D | E | - | F | G | + | * | - | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

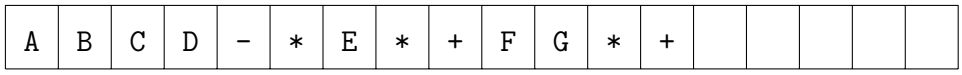
### Aufgabe 3

Input:




Stack: (nimmt → zu)


Output:




### Reguläre Ausdrücke (Auswahl)

Reguläre Ausdrücke sind Suchmuster, die in einem Text bestimmte Klassen von Teilstrings erkennen können.

### Beispiel 4

```
import re
pattern = re.compile('^\d+$')
txt = '1245'
print(pattern.match(txt))
```

```
import re
pattern = re.compile('^\d{3,4}$')
txt = '12'
print(pattern.match(txt))
```

### Aufgabe 4

```
import re
pattern = re.compile('^\d{2,4}$')
txt = '12345'
print(pattern.match(txt))
```

```
import re
pattern = re.compile('^\w?\d{2,4}$')
txt = 'A123'
print(pattern.match(txt))
```

### Tokenisierung von Zeichenketten

*Tokenisierung* bezeichnet die Zerlegung eines Textes (Zeichenkette) in seine kleinsten Einheiten – meist Wörter, um ihn in zerlegter Form weiterverarbeiten zu können.

In der einfachsten Form wird der Text an Leerzeichen, Zeilenschaltungen oder Tabulatoren aufgetrennt (White-Space-Tokenisierung).

In Python lassen sich Tokenisierungen mit

`L = text.split(separator)`

durchführen, wobei ***separator*** ein String ist, der angibt, an welchen Zeichen der ***text*** aufgetrennt werden soll. `L` enthält dann die Liste der Tokens.

### Beispiel 5

```
text = '5 + 3 * 23'
L = text.split()
print(L)
```

### Aufgabe 5

```
text = '5+3+23+9'
L = text.split('+')
print(L)
```

### Python-Programm infix_to_postfix.py

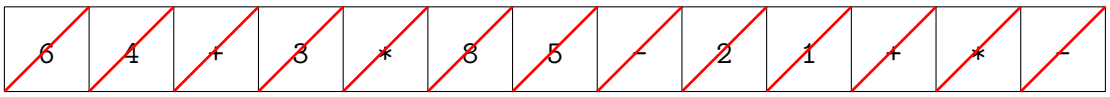
```
import re
from stack import Stack

def infix_to_postfix(infix_string):

    operand = re.compile('\d+|\w+')
    ops = {'*': 2, '/': 2, '+': 1, '-': 1}
    token_list = infix_string.split()
    output = []
    s = Stack()

    for token in token_list:
        if operand.match(token):
            output.append(token)
        elif token == '(':
            s.push(token)
        elif token == ')':
            top_item = s.pop()
            while top_item != '(':
                output.append(top_item)
                top_item = s.pop()
        elif token in ops:
            while not s.is_empty():
                top_item = s.peek()
                if top_item != '(' and ops[top_item] >= ops[token]:
                    output.append(s.pop())
                else:
                    break
            s.push(token)
        else:
            return None # ungültiges Token

    while not s.is_empty():
        output.append(s.pop())

    return ' '.join(output)
```

### Auswertung von Postfix-Ausdrücken

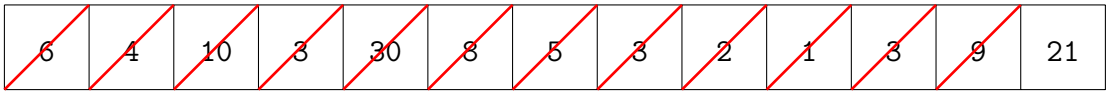
1. Erstelle einen leeren Stack für die Operanden.
2. Konvertiere die Zeichenkette in eine Liste von Token.
3. Durchlaufe die Tokenliste von links nach rechts.
   (a) Ist das Token ein Operand, konvertiere es in eine Ganz- oder Gleitkommazahl und lege die Zahl auf dem Operandenstack ab.
   (b) Ist das Token ein Operator (`*`, `/`, `+`, oder `-`), werden zwei Operanden vom Stack entfernt. Der zuerst entfernte Operand ist der zweite (rechte) Operand und der zuletzt entfernte Operand ist der erste (linke) Operand. Führe die arithmetische Operation aus und lege das Ergebnis auf dem Stack ab.
4. Wenn der Eingabeausdruck vollständig verarbeitet wurde, liegt das Ergebnis auf dem Stack und kann zurückgeben werden.

### Beispiel 6

`Input:`



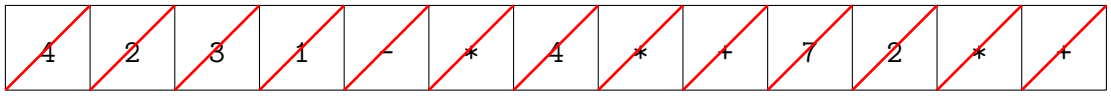

`Stack: (nimmt → zu)`



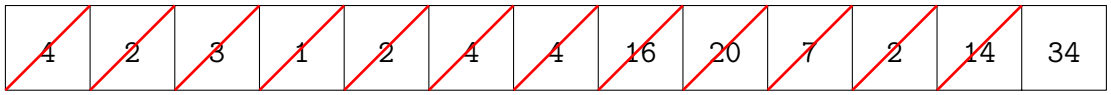

`Output:` 21

## Aufgabe 6

Input:




Stack: (nimmt → zu)




Output: 34

## Python-Programm postfix_eval.py

```
1  import re
2  from stack import Stack
3
4  def postfix_eval(input_string):
5      '''Wertet einen Postifix-String aus'''
6
7      int_str =  re.compile('\d+')
8
9      operand_stack = Stack() # leerer Stack
10     token_list = input_string.split() # String an Blanks auftrennen


11
12     for token in token_list:
13         if int_str.match(token):
14              operand_stack.push(int(token))
15         elif token in ['+', '-', '*', '/']:
16             op1 = operand_stack.pop()
17             op2 = operand_stack.pop()
18             if token == '+':
19                 operand_stack.push(op1 + op2)
20             if token == '-':
21                 operand_stack.push(op1 - op2)
22             if token == '*':
23                 operand_stack.push(op1 * op2)
24             if token == '/':
25                 operand_stack.push(op1 / op2)
26         else:
27             print('Unbekanntes Objekt auf Stack')
28             return None
29
30     return operand_stack.pop()
```