PROFESSOR ERKLÄRBÄR

Serie Informatik · Folge 15 · Advanced

INFORMATIK

LEHRERINFO ZUM VIDEO

Ein Rechner, der Punkt vor Strich kann

FACH	STUFE	DAUER	VIDEO
Informatik	Gymnasium · Sek II	15 min	youtube.com/@Erklaerbaer

01 Lernziele

Nach dem Video können die Schülerinnen und Schüler …

- einen Rechner für ganze Ausdrücke mit Klammern und Punkt vor Strich in Stufen (Lexer, Parser, Auswerter) entwerfen,
- einen Lexer als endlichen Automaten beschreiben und einen Ausdruck in Tokens zerlegen,
- Ausdrücke mit dem Shunting-Yard-Algorithmus in Postfix umwandeln und mit einem Stack auswerten,
- aus einer Grammatik einen rekursiven Abstiegsparser ableiten,
- Fehler mit eigenen Ausnahmen behandeln und begründen, warum `eval()` tabu ist,
- ein Programm mit Unit-Tests absichern, auch mit Fehlerfällen.

02 Voraussetzungen

Taschenrechner, Funktionen, `try`/`except` (Folge 15 · Grundlagen); Tokens und Grammatik (Folge 7 · Advanced); endliche Automaten (Folge 8 · Advanced); Stack (Folge 9 · Advanced); Klassen (Folge 10 · Advanced); Rekursion (Folge 6 · Advanced); `eval` als Sicherheitslücke (Folge 13 · Advanced).

03 Ablauf des Videos

ZEIT	KAPITEL	INHALT UND BEISPIEL
0:00	Einstieg	3 + 4 · 2: einfacher Rechner 14, wissenschaftlicher 11 · Ident · Fahrplan
0:35	Was du brauchst	Rückblick 15G, 07A, 08A, 09A, 10A
1:01	Architektur	Text hinein, Zahl heraus · Lexer → Parser → Auswerter · Klassendiagramm · Merksatz
2:02	Der Lexer	Automat für Zahlen (3,5 und 3.5) · Zeichen-Tabelle · 9 Tokens · Vorzeichen
3:01	Postfix und Stack	3 4 2 1 − · + · Stack-Auswertung · UPN-Rechner PAUSE-FRAGE 5 1 2 + 4 · + 3 −
4:24	Shunting-Yard	Dijkstra 1961 · Regeln und Prioritäten · Tabelle für 3 + 4 · (2 − 1) · 8 − 2 − 3 PAUSE-FRAGE (1 + 2) · (3 + 4) · Merksatz
6:26	Rekursiver Abstieg	Grammatik → expression, term, factor · Aufrufbaum für (3 + 4) · 2
7:32	Fehler und Grenzen	CalcError · drei Fehlerfälle · `decimal` · kein `eval`
8:40	Testen	`unittest` · 10 Testfälle, davon 3 Fehlerfälle · testgetriebene Entwicklung · Merksatz
9:37	Erweitern und Rückblick	2 ^ 3 ^ 2 = 512 · Vorzeichen, Funktionen, Tabellenkalkulation · Bausteine aller Advanced-Folgen
10:41	Prüfungsfallen	fünf Fallen (siehe 04)
11:34	Zusammenfassung	Kernaussagen · zurück zum Einstieg

12:15	Quiz	4 Fragen · Endkarte Folge 16

04 Prüfungsfallen

- Postfix braucht Klammern – gerade nicht; die Reihenfolge ist eindeutig.
- Bei gleicher Priorität kommt der neue Operator einfach auf den Stack – bei linksassoziativen Operatoren wird zuerst der alte ausgegeben (8 − 2 − 3 = 3).
- Potenzen sind linksassoziativ – 2 ^ 3 ^ 2 = 2 ^ (3 ^ 2) = 512, also rechtsassoziativ.
- Bestandene Tests beweisen Korrektheit – Tests zeigen nur, dass die getesteten Fehler fehlen.
- `eval()` ist ein guter Parser – `eval` lässt beliebigen Code laufen, eine Sicherheitslücke.

05 Einsatz im Unterricht

Stack zum Anfassen: Postfix mit Karteikarten auswerten; ein Stapel auf dem Tisch ist der Stack.

Rangierbahnhof spielen: Lernende sind Tokens und laufen nach den Shunting-Yard-Regeln in «Ausgabe» oder «Nebengleis».

Testen: Gruppen schreiben je drei Testfälle, die den Code einer anderen Gruppe zum Scheitern bringen sollen.

Vergleich: Shunting-Yard und rekursiver Abstieg an denselben Testfällen laufen lassen; Unterschiede bei fehlerhaften Eingaben besprechen.

Hinweis: Der Code im Anhang ist mit Python 3.12 getestet; alle 10 Unit-Tests bestehen. Er nutzt `^` für Potenzen (in Python selbst wäre das `**`). Der Shunting-Yard-Teil prüft die Syntax nicht vollständig (z. B. zwei Zahlen hintereinander erkennt erst die Auswertung oder gar nicht); der rekursive Abstieg meldet solche Eingaben zuverlässig («Zeichen nach dem Ende») – ein gutes Diskussionsthema. Der Java-Parser ist eine direkte Übersetzung des rekursiven Abstiegs ohne Potenz und Vorzeichen.

06 Quellen

- E. W. Dijkstra (1961): ALGOL-60 Translation, Mathematisch Centrum, Amsterdam – Ursprung des Shunting-Yard-Algorithmus
- A. Aho, M. Lam, R. Sethi, J. Ullman: Compilers – Principles, Techniques, and Tools, 2. Auflage, Pearson 2006
- N. Wirth: Grundlagen und Techniken des Compilerbaus, Oldenbourg
- Python-Dokumentation: `unittest`, `decimal`, Built-in `eval` – docs.python.org/3/library

NAME _____ KLASSE _____ DATUM _____

Ein Rechner, der Punkt vor Strich kann

1 Postfix auswerten ●○○

Werte mit einem Stack aus und **notiere** den Stack nach jedem Operator: a) 2 3 4 · + b) 9 3 / 2 − c) 5 1 2 + 4 · + 3 −

2 Infix → Postfix ●○○

a) 2 + 3 · 4 b) (2 + 3) · 4 c) (1 + 2) · (3 + 4)

3 Shunting-Yard Zeichen für Zeichen ●●○

Fülle die Tabelle für 7 − 2 · (1 + 3) aus und **berechne** das Ergebnis.

TOKEN	AUSGABE	OPERATOR-STACK
7		
−		
2		
·		
(		
1		
+		
3		
)		
Ende		

4 Assoziativität ●●○

Berechne und gib das Postfix an: a) 10 − 4 − 3 b) 2 ^ 2 ^ 3. **Begründe** den Unterschied.

5 Rekursiver Abstieg ●●●

Zeichne den Aufrufbaum von `expression`, `term` und `factor` für 2 · (3 + 1) und trage an jedem Knoten den zurückgegebenen Wert ein.

6 Unit-Tests ●●○

Schreibe vier weitere Testmethoden für den Rechner, davon mindestens eine mit `assertRaises(CalcError)`.

Lösungen

1 Postfix auswerten

a) Stack 2 3 4 → · → 2 12 → + → 14 · b) 9 3 → / → 3 · 3 2 → − → 1 · c) 5 1 2 → + → 5 3 · 4 → · → 5 12 → + → 17 · 3 → − → 14

2 Infix → Postfix

a) 2 3 4 · + (= 14) · b) 2 3 + 4 · (= 20) · c) 1 2 + 3 4 + · (= 21)

3 Shunting-Yard

Ausgabe | Stack: 7 | − · 7 | − · 7 2 | − · 7 2 | − · · 7 2 | − · (· 7 2 1 | − · (· 7 2 1 | − · (+ · 7 2 1 3 | − · (+ · 7 2 1 3 + | − · · Ende: 7 2 1 3 + · − · Ergebnis: 1 + 3 = 4, 2 · 4 = 8, 7 − 8 = −1.

4 Assoziativität

a) 10 4 − 3 − = 3 (Minus linksassoziativ: (10 − 4) − 3) · b) 2 2 3 ^ ^ = 2 ^ 8 = 256 (Potenz rechtsassoziativ: 2 ^ (2 ^ 3)); linksherum wären es (2 ^ 2) ^ 3 = 64.

5 Rekursiver Abstieg

expression → term; term → factor (2), sieht «·», → factor: sieht «(» → expression → term → factor (3), «+», term → factor (1) → expression liefert 4 → factor liefert 4 → term: 2 · 4 = 8 → expression liefert 8.

6 Unit-Tests

z. B. `assertEqual(c.calc("2 * (3 + 4)"), 14)`, `assertEqual(c.calc("10 / 4"), 2.5)`, `assertEqual(c.calc("2 + 3 * 4"), 14)`, `with self.assertRaises(CalcError): c.calc("1 +")` (Meldung «Operator ohne Zahl»).

Q Quiz im Video

1 B (Stack) · 2 A (3) · 3 C (Tokens) · 4 B (lässt beliebigen Code laufen)

Anhang A: Ausdrucksrechner in Python

Datei `ausdruck.py` (Python 3, getestet mit Python 3.12). Start mit `python ausdruck.py`.

```
# Ausdrucksrechner - Serie Informatik, Folge 15 (Advanced)
# Text → Lexer → Tokens → Shunting-Yard → Postfix → Stack → Zahl


class CalcError(Exception):
    """Fehler im Ausdruck: Syntaxfehler oder Division durch null."""


class Token:
    def __init__(self, kind, value):
        self.kind = kind      # "NUM", "OP", "LPAR" oder "RPAR"
        self.value = value

    def __repr__(self):
        return f"{self.kind}({self.value})"


class Lexer:
    """Endlicher Automat: liest Zeichen für Zeichen und bildet Tokens."""

    def __init__(self, text):
        self.text = text.replace(",", ".")

    def tokens(self):
        out, i, t = [], 0, self.text
        while i < len(t):
            ch = t[i]
            if ch == " ":
                i += 1
            elif ch.isdigit() or ch == "." or (ch == "-" and self.sign_allowed(out)
                                               and i + 1 < len(t) and t[i + 1].isdigit()):
                j = i + 1
                while j < len(t) and (t[j].isdigit() or t[j] == "."):
                    j += 1
                try:
                    out.append(Token("NUM", float(t[i:j])))
                except ValueError:
                    raise CalcError(f"Ungültige Zahl: {t[i:j]}")
                i = j
            elif ch in "+-*/^":
                out.append(Token("OP", ch))
                i += 1
            elif ch == "(":
                out.append(Token("LPAR", ch))
                i += 1
            elif ch == ")":
                out.append(Token("RPAR", ch))
                i += 1
            else:
                raise CalcError(f"Unbekanntes Zeichen: {ch}")
        return out

    @staticmethod
    def sign_allowed(out):
        return not out or out[-1].kind in ("OP", "LPAR")


PREC = {"+": 1, "-": 1, "*": 2, "/": 2, "^": 3}
RIGHT = {"^"}


def apply(op, a, b):
    if op == "+":
        return a + b
    if op == "-":
        return a - b
    if op == "*":
        return a * b
    if op == "/":
        if b == 0:
            raise CalcError("Division durch null")
        return a / b
```

```
    return a ** b


class Parser:
    """Alternative: rekursiver Abstieg nach der Grammatik aus Folge 7 Advanced."""

    def __init__(self, tokens):
        self.toks, self.pos = tokens, 0

    def to_postfix(self):
        """Infix → Postfix (Edsger Dijkstra, 1961) mit einem Operator-Stack."""
        output, stack = [], []
        for tok in self.toks:
            if tok.kind == "NUM":
                output.append(tok)
            elif tok.kind == "OP":
                while (stack and stack[-1].kind == "OP" and
                       (PREC[stack[-1].value] > PREC[tok.value] or
                        (PREC[stack[-1].value] == PREC[tok.value] and tok.value not in RIGHT))):
                    output.append(stack.pop())
                stack.append(tok)
            elif tok.kind == "LPAR":
                stack.append(tok)
            else:  # RPAR
                while stack and stack[-1].kind != "LPAR":
                    output.append(stack.pop())
                if not stack:
                    raise CalcError("Klammer zu ohne Klammer auf")
                stack.pop()
        while stack:
            if stack[-1].kind == "LPAR":
                raise CalcError("Klammer auf ohne Klammer zu")
            output.append(stack.pop())
        return output

    def peek(self):
        return self.toks[self.pos] if self.pos < len(self.toks) else None

    def take(self):
        tok = self.peek()
        if tok is None:
            raise CalcError("Ausdruck endet zu früh")
        self.pos += 1
        return tok

    def parse(self):
        value = self.expression()
        if self.peek() is not None:
            raise CalcError("Zeichen nach dem Ende")
        return value

    def expression(self):          # Ausdruck = Term { ("+" | "-") Term }
        value = self.term()
        while self.peek() and self.peek().value in ("+", "-"):
            op = self.take().value
            value = apply(op, value, self.term())
        return value

    def term(self):                # Term = Faktor { ("*" | "/") Faktor }
        value = self.factor()
        while self.peek() and self.peek().value in ("*", "/"):
            op = self.take().value
            value = apply(op, value, self.factor())
        return value

    def factor(self):              # Faktor = Zahl | "(" Ausdruck ")"
        tok = self.take()
        if tok.kind == "NUM":
            return tok.value
        if tok.kind == "LPAR":
            value = self.expression()
            if self.take().kind != "RPAR":
                raise CalcError("Klammer zu fehlt")
            return value
        raise CalcError(f"Zahl erwartet, nicht {tok.value}")


class Calculator:
    def evaluate(self, postfix):
        """Postfix mit einem Zahlen-Stack auswerten."""
        stack = []
        for tok in postfix:
            if tok.kind == "NUM":
                stack.append(tok.value)
            else:
                if len(stack) < 2:
                    raise CalcError("Operator ohne Zahl")
                b = stack.pop()
                a = stack.pop()
                stack.append(apply(tok.value, a, b))
```

```
        if len(stack) != 1:
            raise CalcError("Zahl ohne Operator")
        return stack[0]

    def calc(self, text):
        return self.evaluate(Parser(Lexer(text).tokens()).to_postfix())


if __name__ == "__main__":
    calc = Calculator()
    while True:
        text = input("Ausdruck (leer = Ende): ")
        if not text:
            break
        try:
            print("=", round(calc.calc(text), 10))
        except CalcError as e:
            print("Fehler:", e)
```

Anhang B: Unit-Tests

Datei `test_ausdruck.py` im selben Ordner. Start mit `python -m unittest -v test_ausdruck` – Ausgabe: «Ran 10 tests … OK».

```
import unittest
from ausdruck import Calculator, CalcError


class TestCalculator(unittest.TestCase):
    def setUp(self):
        self.c = Calculator()

    def test_punkt_vor_strich(self):
        self.assertEqual(self.c.calc("3 + 4 * 2"), 11)

    def test_klammern(self):
        self.assertEqual(self.c.calc("3 + 4 * (2 - 1)"), 7)

    def test_linksassoziativ(self):
        self.assertEqual(self.c.calc("8 - 2 - 3"), 3)

    def test_klammer_zuerst(self):
        self.assertEqual(self.c.calc("(3 + 4) * 2"), 14)

    def test_potenz_rechtsassoziativ(self):
        self.assertEqual(self.c.calc("2 ^ 3 ^ 2"), 512)

    def test_komma_und_punkt(self):
        self.assertEqual(self.c.calc("3,5 + 1.5"), 5)

    def test_vorzeichen(self):
        self.assertEqual(self.c.calc("-3 + 5"), 2)

    def test_division_durch_null(self):
        with self.assertRaises(CalcError):
            self.c.calc("7 / 0")

    def test_operator_doppelt(self):
        with self.assertRaises(CalcError):
            self.c.calc("3 + * 4")

    def test_klammer_fehlt(self):
        with self.assertRaises(CalcError):
            self.c.calc("(1 + 2")


if __name__ == "__main__":
    unittest.main()
```

Anhang C: Der Parser in Java

Datei `Parser.java`: rekursiver Abstieg mit + − · / und Klammern. Start mit `javac Parser.java` und `java Parser`.

```
// Rekursiver Abstieg in Java – Serie Informatik, Folge 15 (Advanced)
// Grammatik: Ausdruck = Term { ("+" | "-") Term } · Term = Faktor { ("*" | "/") Faktor } · Faktor = Zahl | "(" Ausdruck ")"
public class Parser {
    private final String s;
    private int pos = 0;

    public Parser(String text) {
        s = text.replace(" ", "").replace(",", ".");
    }

    public double parse() {
        double v = expression();
        if (pos < s.length()) throw new IllegalArgumentException("Zeichen nach dem Ende");
        return v;
    }

    private double expression() {
        double v = term();
```

```
        while (pos < s.length() && (s.charAt(pos) == '+' || s.charAt(pos) == '-')) {
            char op = s.charAt(pos++);
            double r = term();
            v = (op == '+') ? v + r : v - r;
        }
        return v;
    }

    private double term() {
        double v = factor();
        while (pos < s.length() && (s.charAt(pos) == '*' || s.charAt(pos) == '/')) {
            char op = s.charAt(pos++);
            double r = factor();
            if (op == '/' && r == 0) throw new ArithmeticException("Division durch null");
            v = (op == '*') ? v * r : v / r;
        }
        return v;
    }

    private double factor() {
        if (pos < s.length() && s.charAt(pos) == '(') {
            pos++;
            double v = expression();
            if (pos >= s.length() || s.charAt(pos) != ')') throw new IllegalArgumentException("Klammer zu fehlt");
            pos++;
            return v;
        }
        int start = pos;
        while (pos < s.length() && (Character.isDigit(s.charAt(pos)) || s.charAt(pos) == '.')) pos++;
        if (start == pos) throw new IllegalArgumentException("Zahl erwartet");
        return Double.parseDouble(s.substring(start, pos));
    }

    public static void main(String[] args) {
        System.out.println(new Parser("(3 + 4) * 2").parse());      // 14.0
        System.out.println(new Parser("3 + 4 * (2 - 1)").parse());  // 7.0
        System.out.println(new Parser("8 - 2 - 3").parse());        // 3.0
    }
}
```