Symbolisches Rechnen am Computer

Was erwartet euch?

1. Einfache Taschenrechner-Anwendung
2. Vom Eingabestring zum Expression-Tree
3. Beliebig große Zahlen
4. Eine kleine Demo-Anwendung

Anmerkungen zur Präsentation werden in diesen Boxen angegeben :)

Einfacher Taschenrechner

Implementierung

```
1  Console.Write("Erste Zahl: ");
2  double a = Double.Parse(Console.ReadLine());
3  Console.Write("Operation: ");
4  string op = Console.ReadLine();
5  Console.Write("Zweite Zahl: ");
6  double b = Double.Parse(Console.ReadLine());
7
8  double result = op switch
9  {
10     "+" => a + b,
11     "-" => a - b,
12     "*" => a * b,
13     "/" => a / b,
14     _ => throw new NotImplementedException()
15 };
16
17 Console.WriteLine("Ergebnis: " + result);
```

Neuere C#-Versionen können Script-artig mit sog. Top-Level-Statements arbeiten

Probleme

- Hoher Aufwand für Implementierung von Funktionen
  - $\sin(x), \lfloor x \rfloor, \log_b(x), x^y, \ldots$
- Kein Support für längere Terme
  - $(5+2)^8 - 2 \cdot 9$
- Keine Variablen/Konstanten
  - $x = 3y$
  - $2 \cdot \pi^e$

$\Rightarrow$ Wir implementieren Parser für String-Eingaben

Expression Parsing

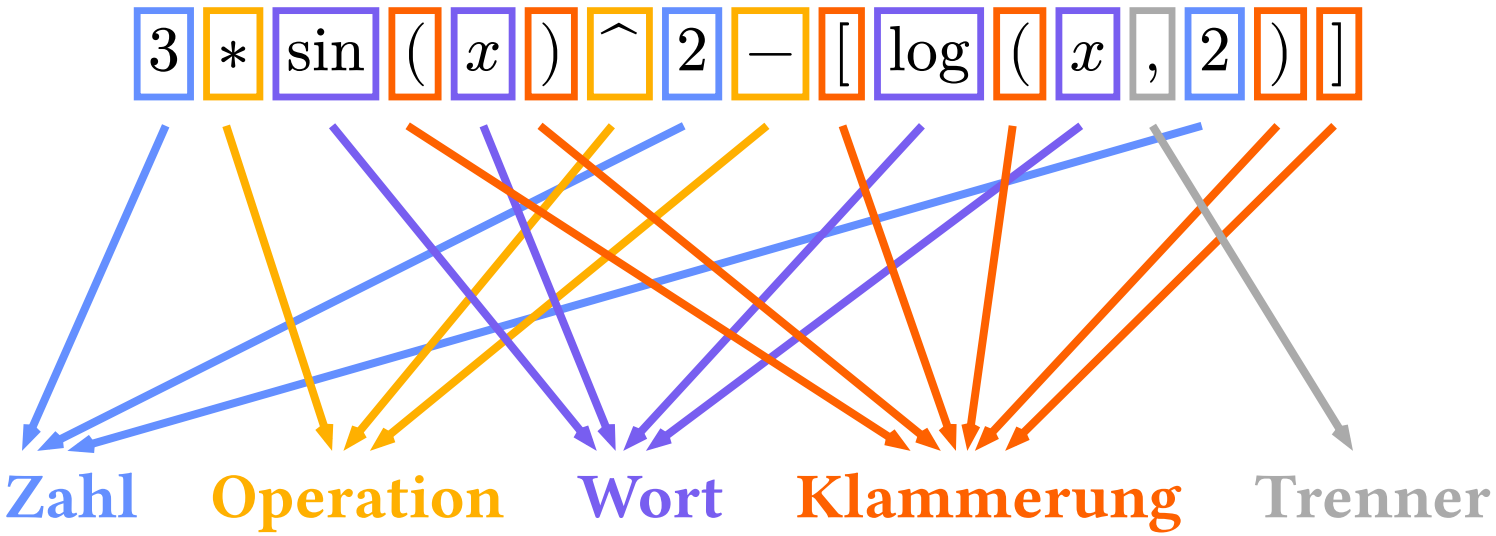
Schritte

1. Tokenization
2. Infix $\rightarrow$ Postfix
3. Expression-Tree
4. Evaluierung des Expression-Tree

Tokenization

Zerlegung der Eingabe in einzelne “Tokens”

Beispiel



Tokenization


```
public class ExpressionToken
{
  public ExpressionToken(TokenType type, string value, int startPos, int endPos)
  {
      this.StartPosition = startPos;
      this.EndPosition = endPos;
      this.Value = value;
      this.Type = type;
  }

  public int Length { get => this.EndPosition - this.StartPosition + 1; }
  public bool IsValid { get => this.Type.IsValid(); }
  public int StartPosition { get; }
  public int EndPosition { get; }
  public TokenType Type { get; }
  public string Value { get; }
}
```

Tokenization

Eingabe von links nach rechts lesen und Tokens erzeugen

```
public IEnumerable<ExpressionToken> Tokenize(string input) {

    List<ExpressionToken> tokens = new List<ExpressionToken>();
    input = input.Replace(" ", "").Replace("\t", "");
    int pos = 0;

    while(pos < input.Length) {
        if(new [] { '(', '[', '{' }.Contains(input[pos])) {
            tokens.Add(new ExpressionToken([...]));
        } else if (new [] { ')', ']', '}' }.Contains(input[pos])) {
            tokens.Add(new ExpressionToken([...]));
        } else if ([...]) {
            [...]
        }
        pos++;
    }
}
```

Tokenization

Aufgepasst!

- Look-Ahead für mehrsymbolige Tokens (bspw. für “>=”)
- Post-Processing für Funktions-Tokens
- Post-Processing für unäres “—”

“Post-Processing für”:

- **“Funktionstokens”: Anzahl Parameter feststellen**
- **“unäres —”: kann man auch im Shunting-Yard-Algorithmus beachten, vorherige Zuordnung zu zugehöriger Zahl ist aber einfacher :)**

Infix- → Postfix-Notation

Infix-Notation?

Klassische Notation mit Operatoren zwischen Operanden

Bsp.: $3 * \sin(x)\hat{}2 - [\log(x, 2)]$

Auch als "Reverse-Polish-Notation" bekannt

Postfix-Notation?

Operatoren folgen auf Operanden

- Evaluierung von rechts nach links
- Keine Klammerung nötig

Bsp.: $3\ x\ \sin\ 2\ \hat{}\ *\ x\ 2\ \log\ -$

Präfix-Notation ist für unsere zwecke auch klammerlos und kann einfach evaluiert werden, Postfix-Notation ist aber der öfter verwendete Standard

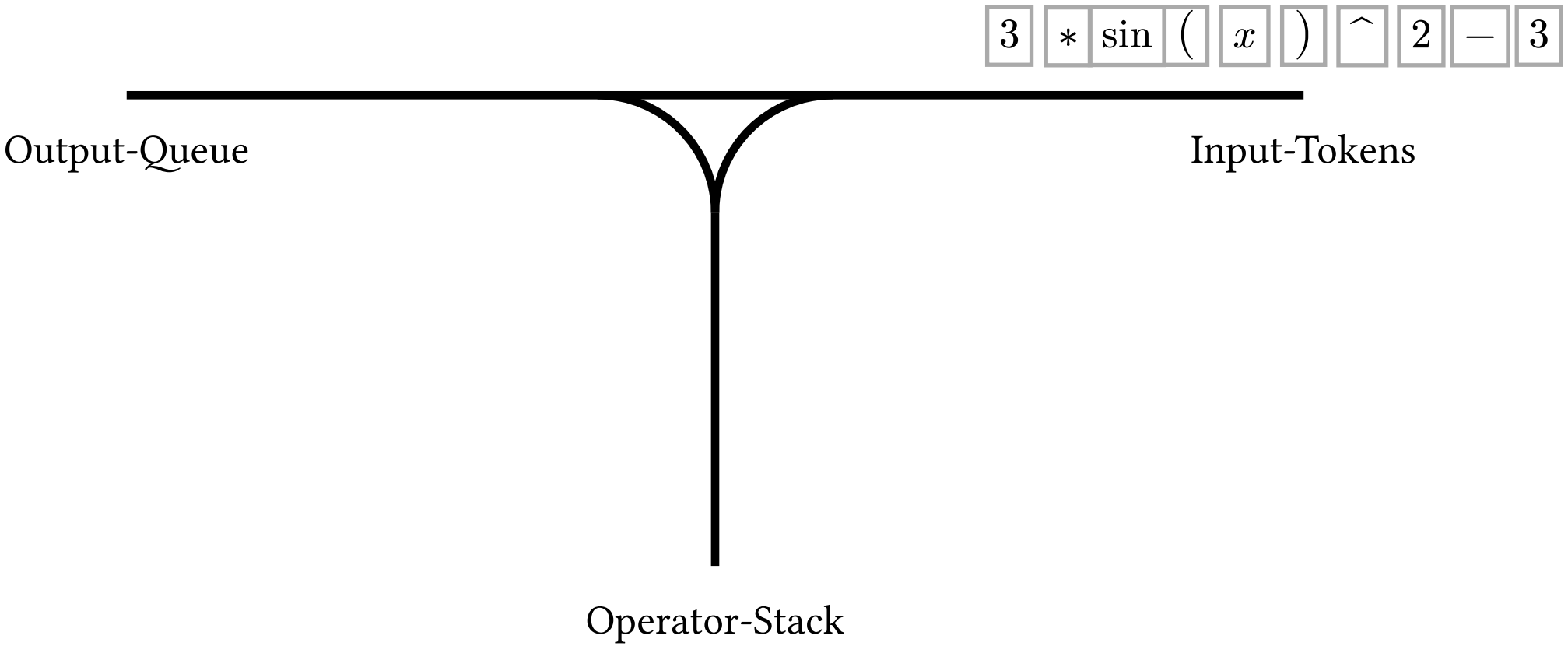
Infix- → Postfix-Notation

Gleich folgt ein Beispiel für den Shunting-Yard-Algorithmus, Folgendes ist hierzu wichtig:

- **Eingabe ist der Infix-Ausdruck des Nutzenden**
- **Ausgabe ist die dazurehörige Postfix-Notation**
- **Eingabetokens werden nacheinander aus der Input-Liste genommen**
  - **Operanden (Zahlen, Variablen, Konstanten) werden sofort in die Output-Queue geschoben**
  - **Operatoren werden auf den Stack gelegt, falls**
    - **dieser leer ist**
    - **der neue Operator schwächer bindet, als der, der oben auf dem Stack liegt**
    - **oder Präzedenz beider Operatoren gleich und Stack-Top ist linksassoziativ (klammerung beginnt in Ausdruck mit mehreren dieser Operatoren implizit links, bspw. $+, -, \cdot, /$, rechtsassoziativ sind bspw. $+, \cdot, \hat{}$)**
- **sobald die Input-Liste und der Operator-Stack leer sind, endet der Algorithmus**
- **Klammern löschen sich gegenseitig aus ⇒ kein expliziter Klammerungstest nötig, da bei falscher Klammerung immer eine Klammer auf dem Stack verbleibt ⇒ Fehlerzustand leicht erkennbar**

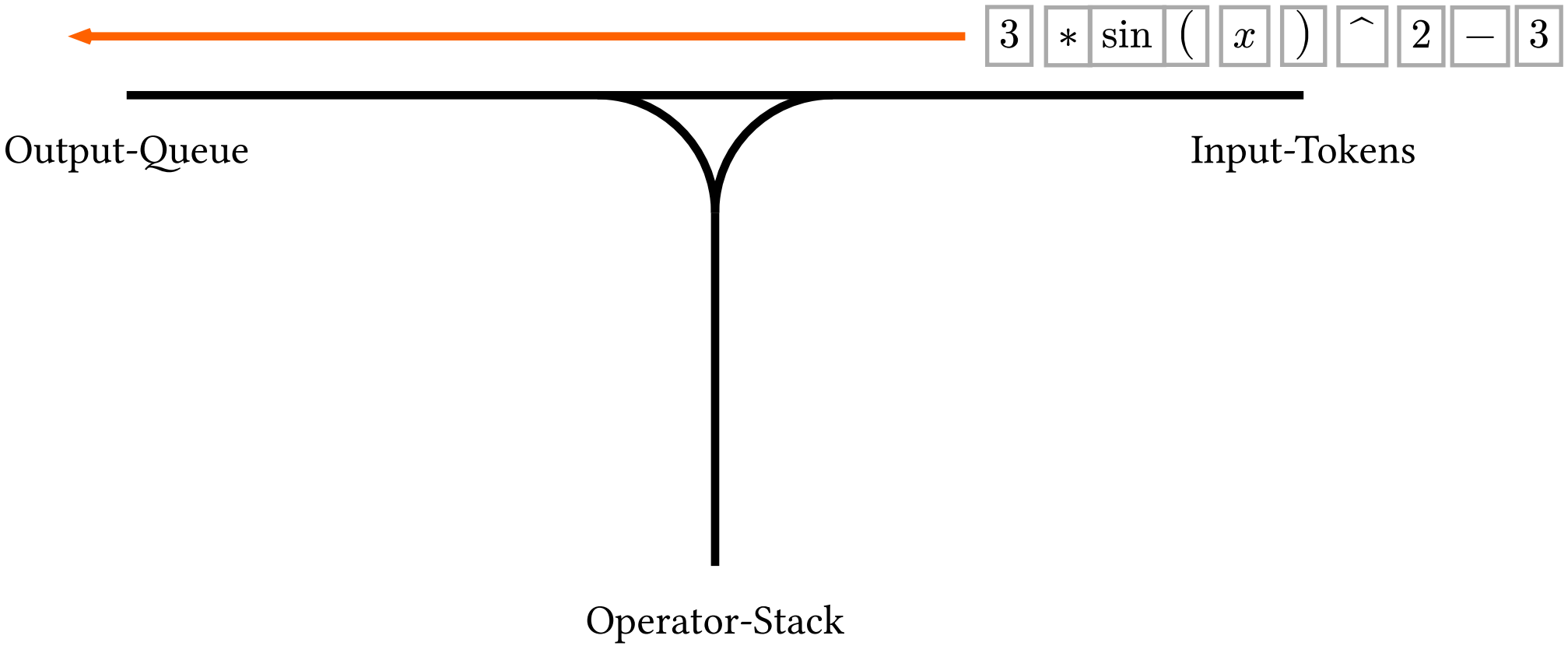
Infix- → Postfix-Notation

Algorithmus: Shunting Yard



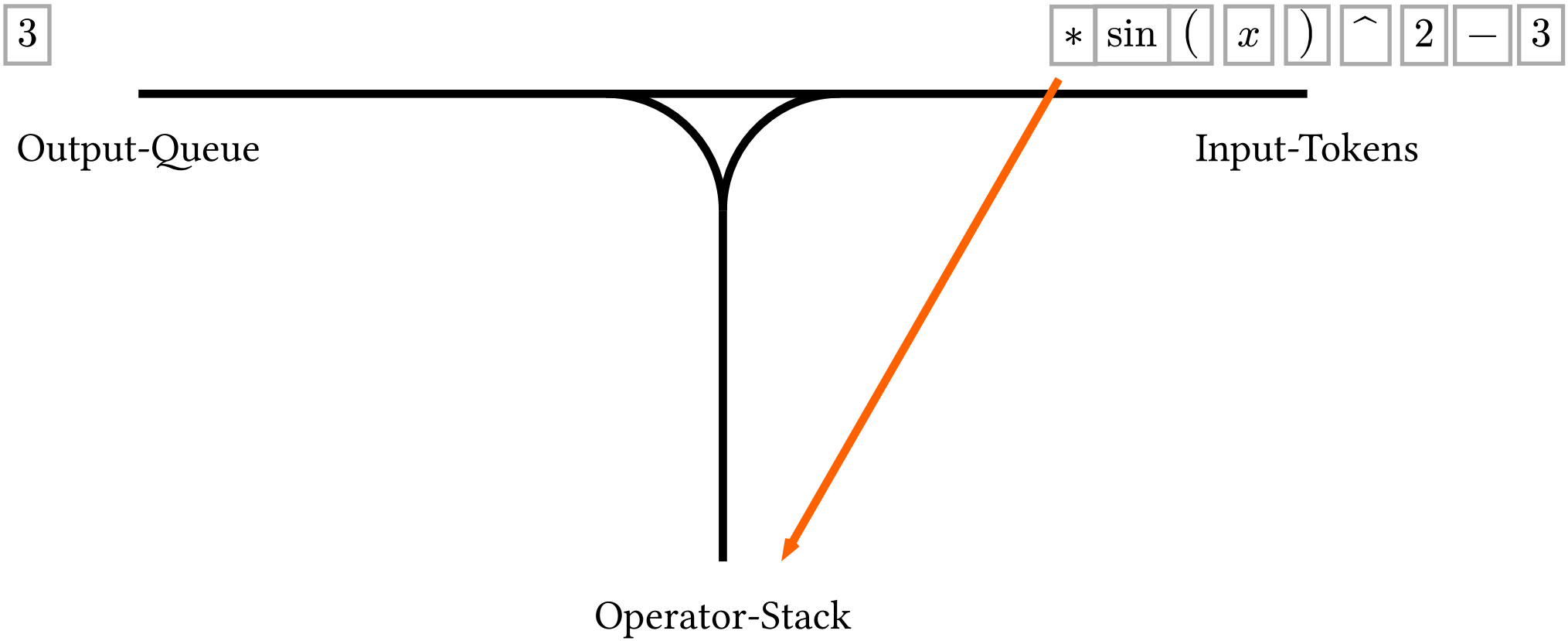
Infix- → Postfix-Notation

Algorithmus: Shunting Yard



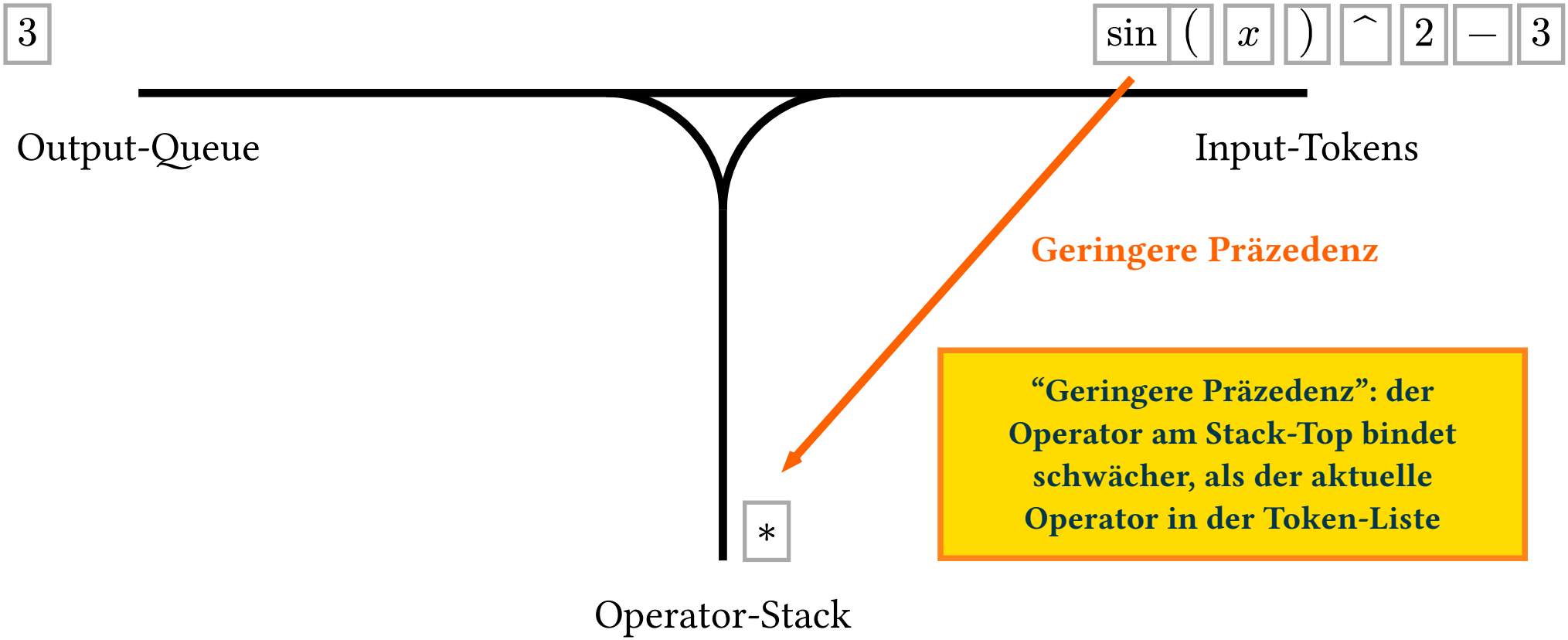
Infix- → Postfix-Notation

Algorithmus: Shunting Yard



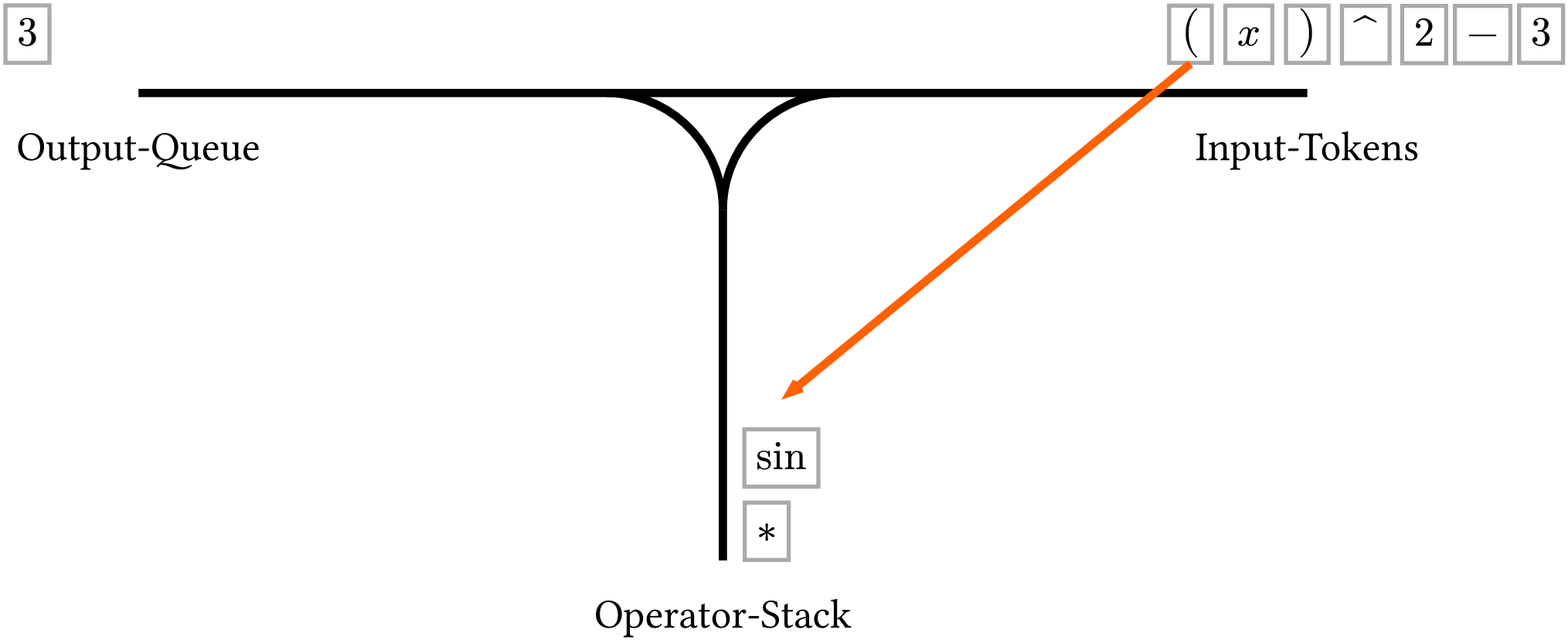
Infix- → Postfix-Notation

Algorithmus: Shunting Yard



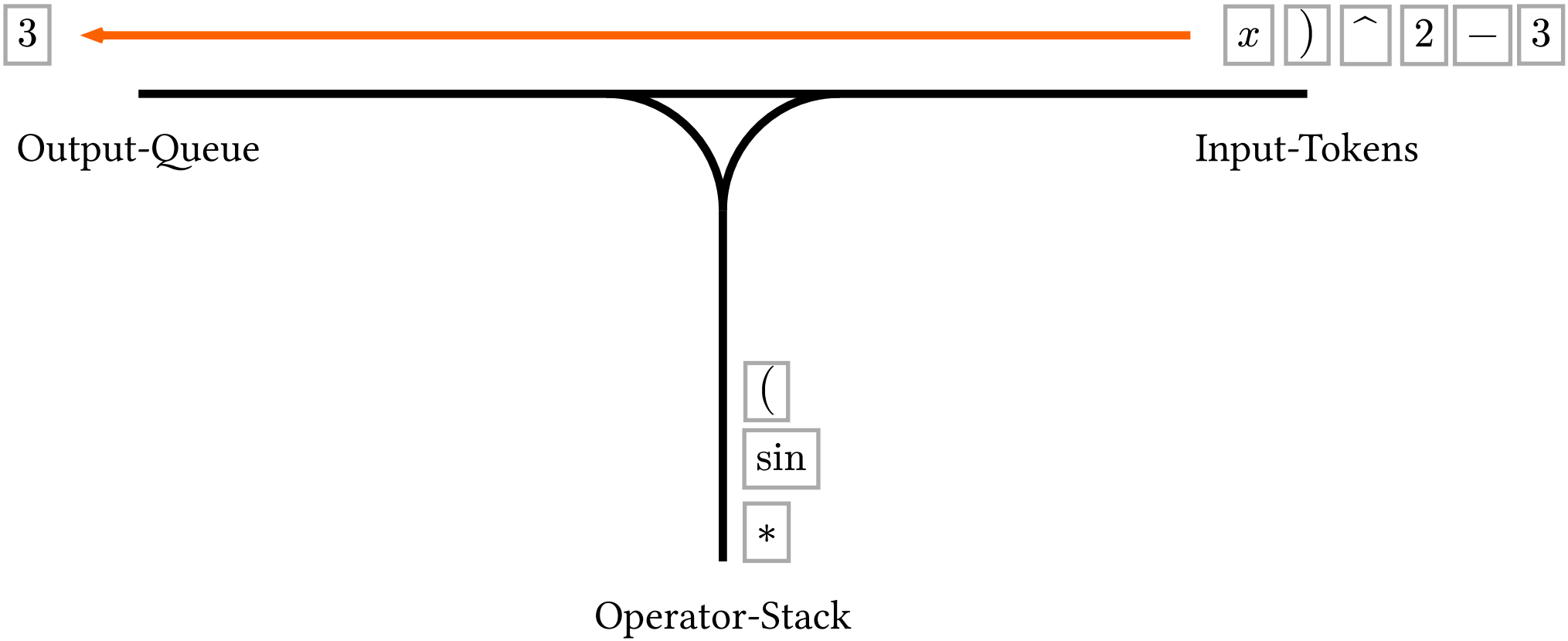
Infix- → Postfix-Notation

Algorithmus: Shunting Yard



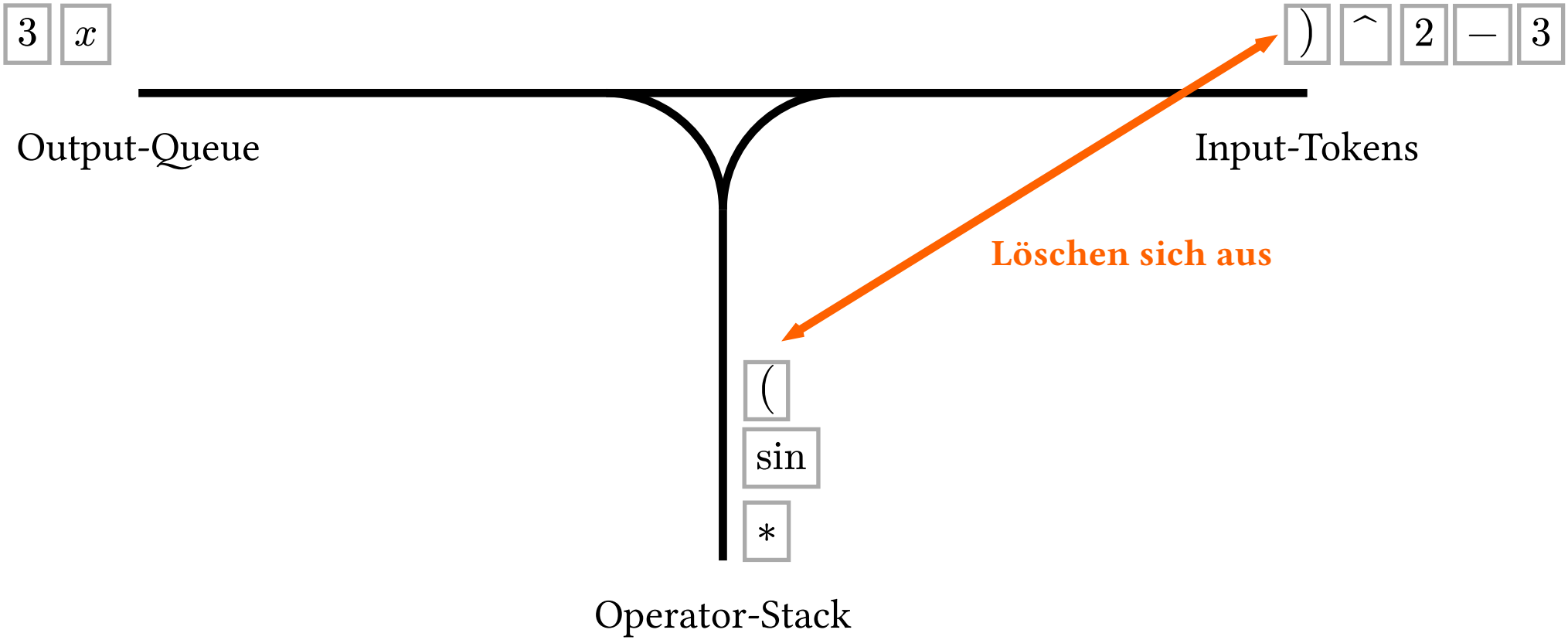
Infix- → Postfix-Notation

Algorithmus: Shunting Yard



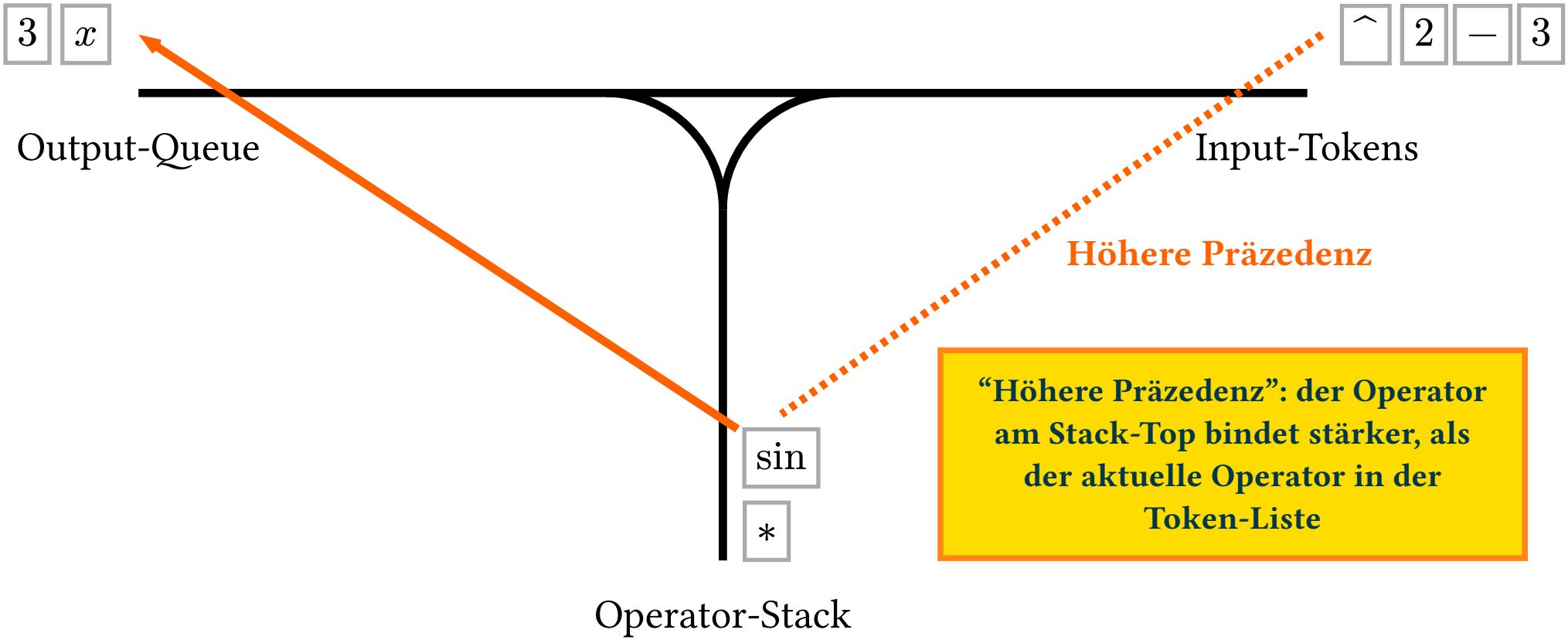
Infix- → Postfix-Notation

Algorithmus: Shunting Yard



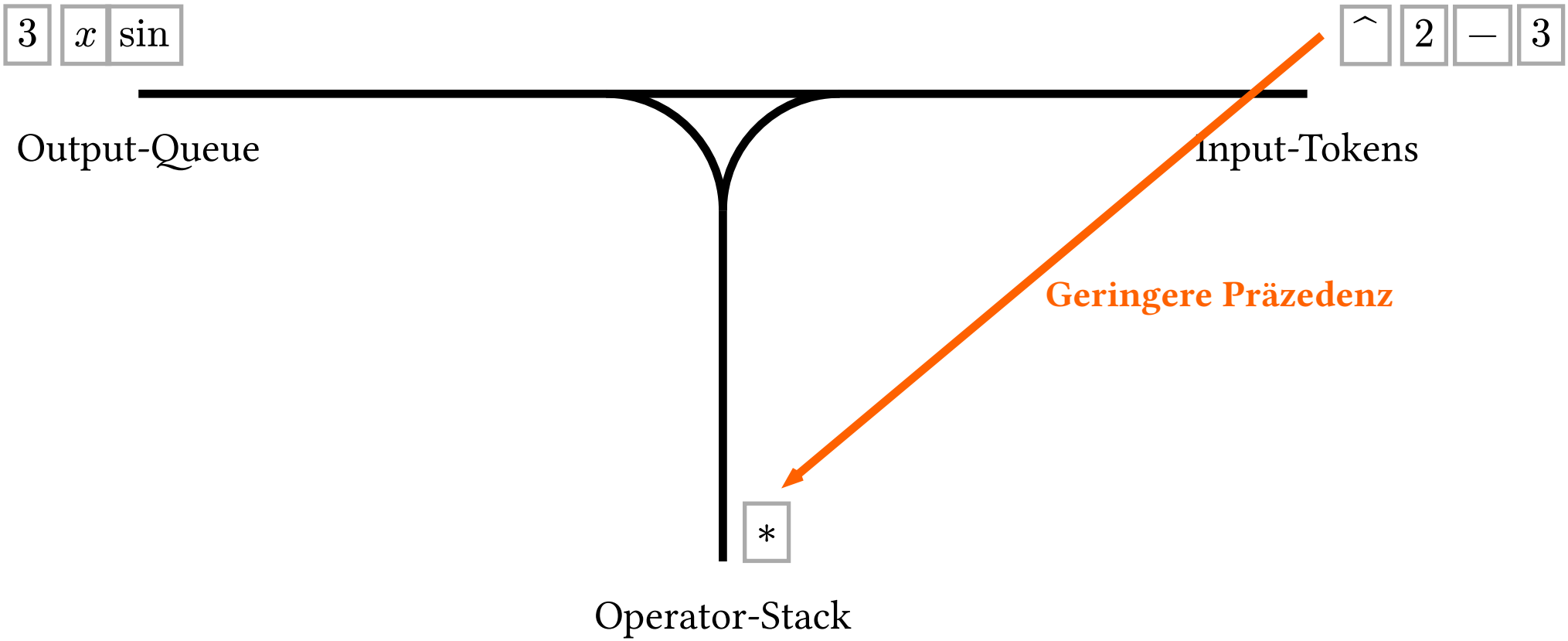
Infix- → Postfix-Notation

Algorithmus: Shunting Yard



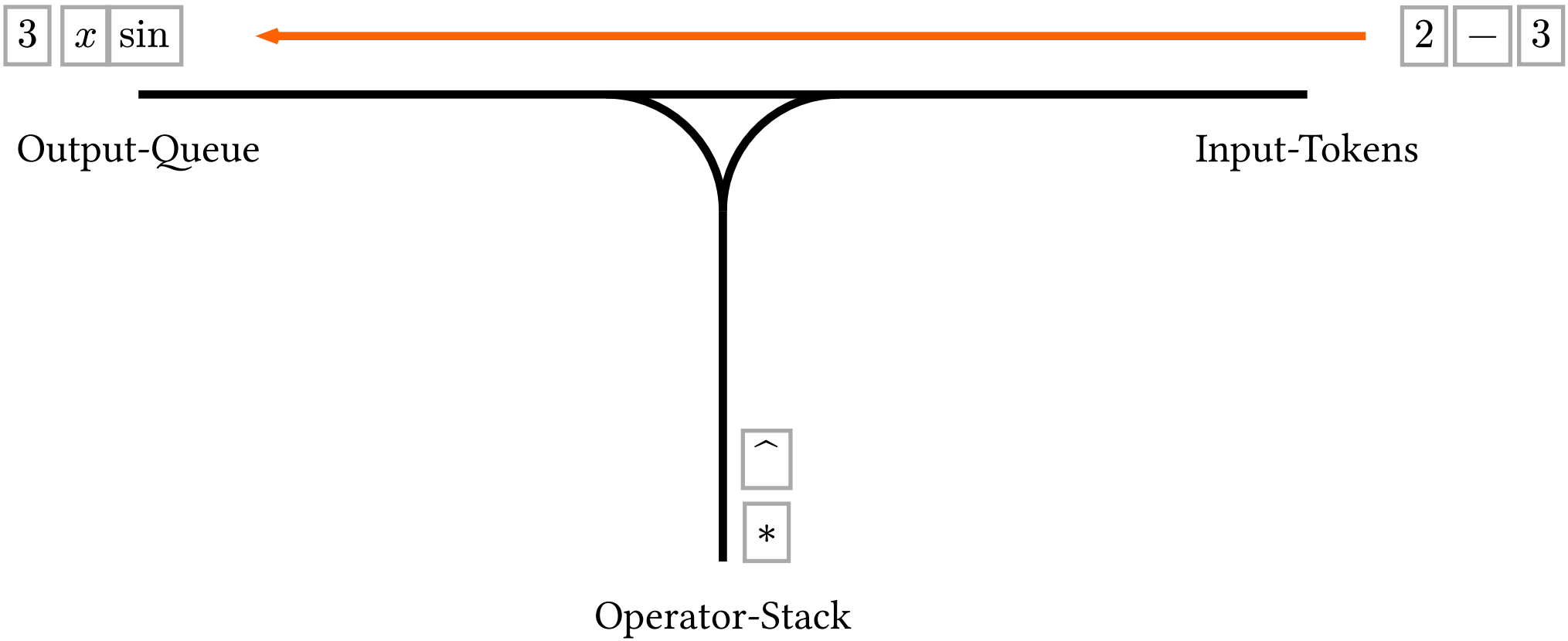
Infix- → Postfix-Notation

Algorithmus: Shunting Yard



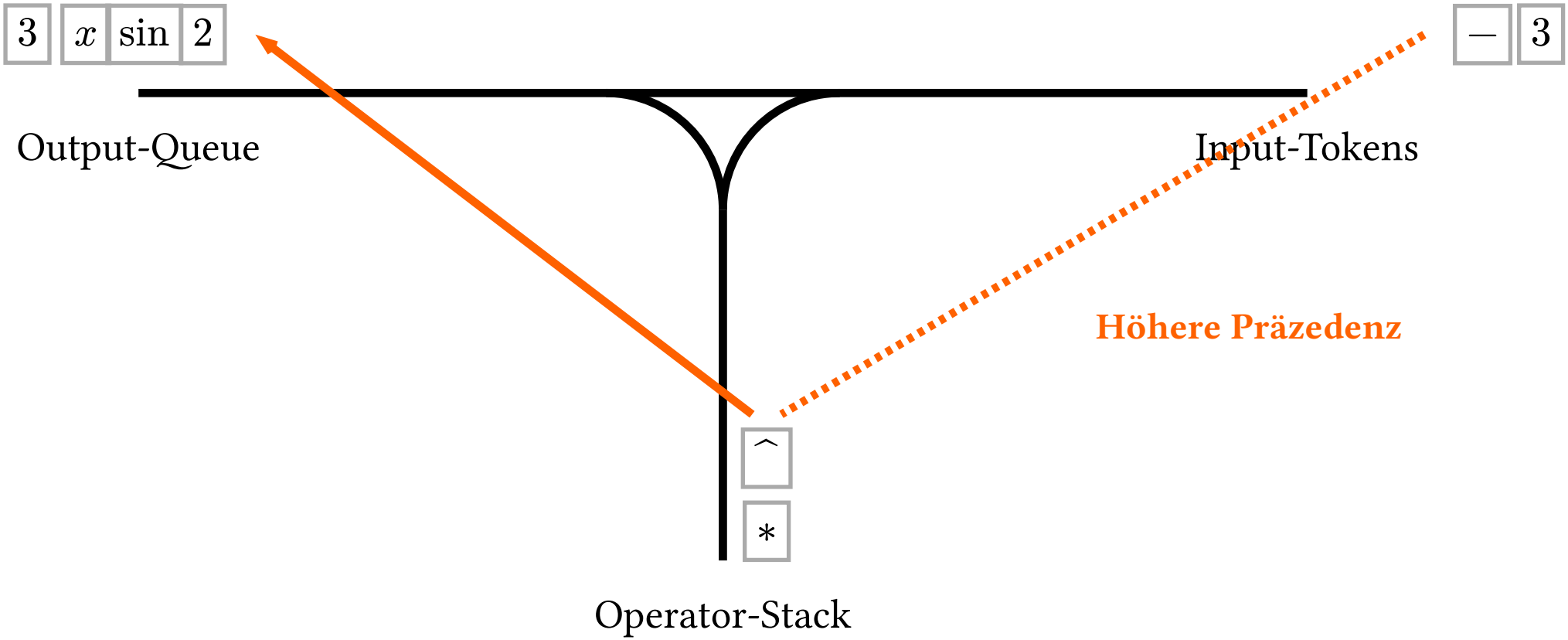
Infix- → Postfix-Notation

Algorithmus: Shunting Yard



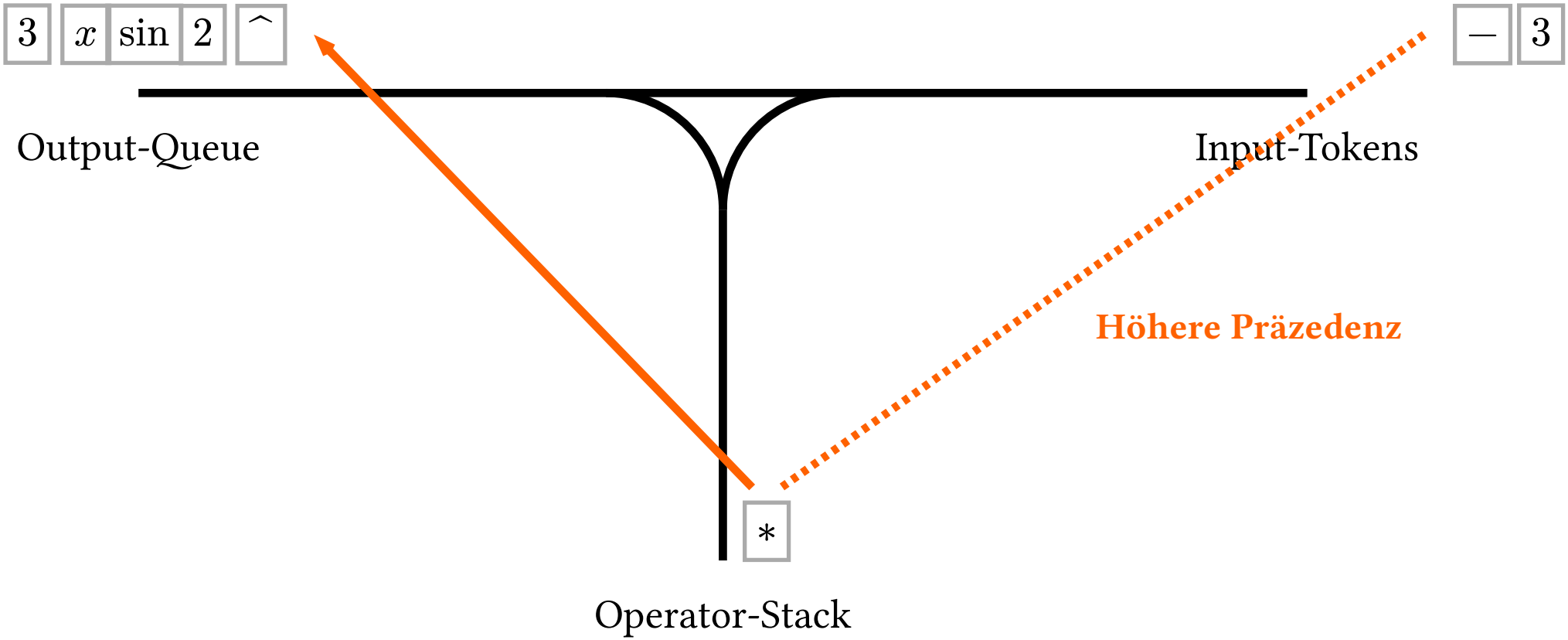
Infix- → Postfix-Notation

Algorithmus: Shunting Yard



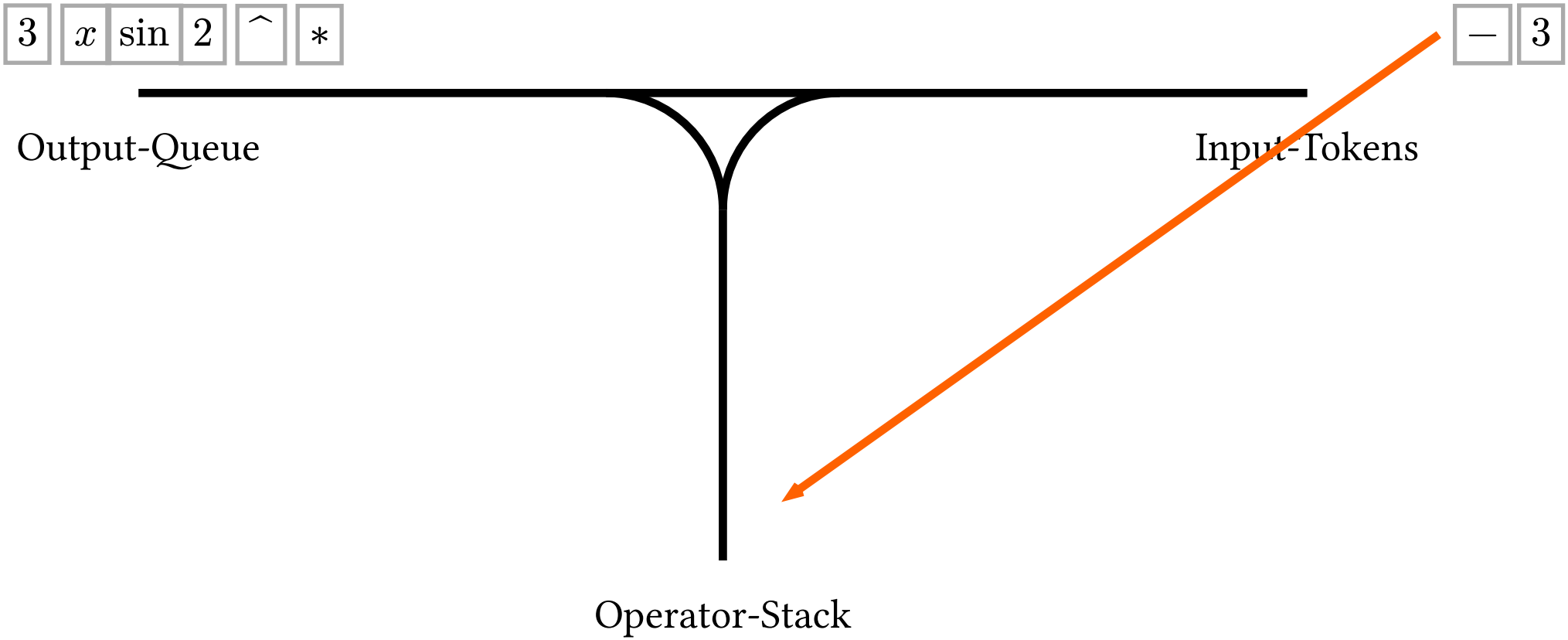
Infix- → Postfix-Notation

Algorithmus: Shunting Yard



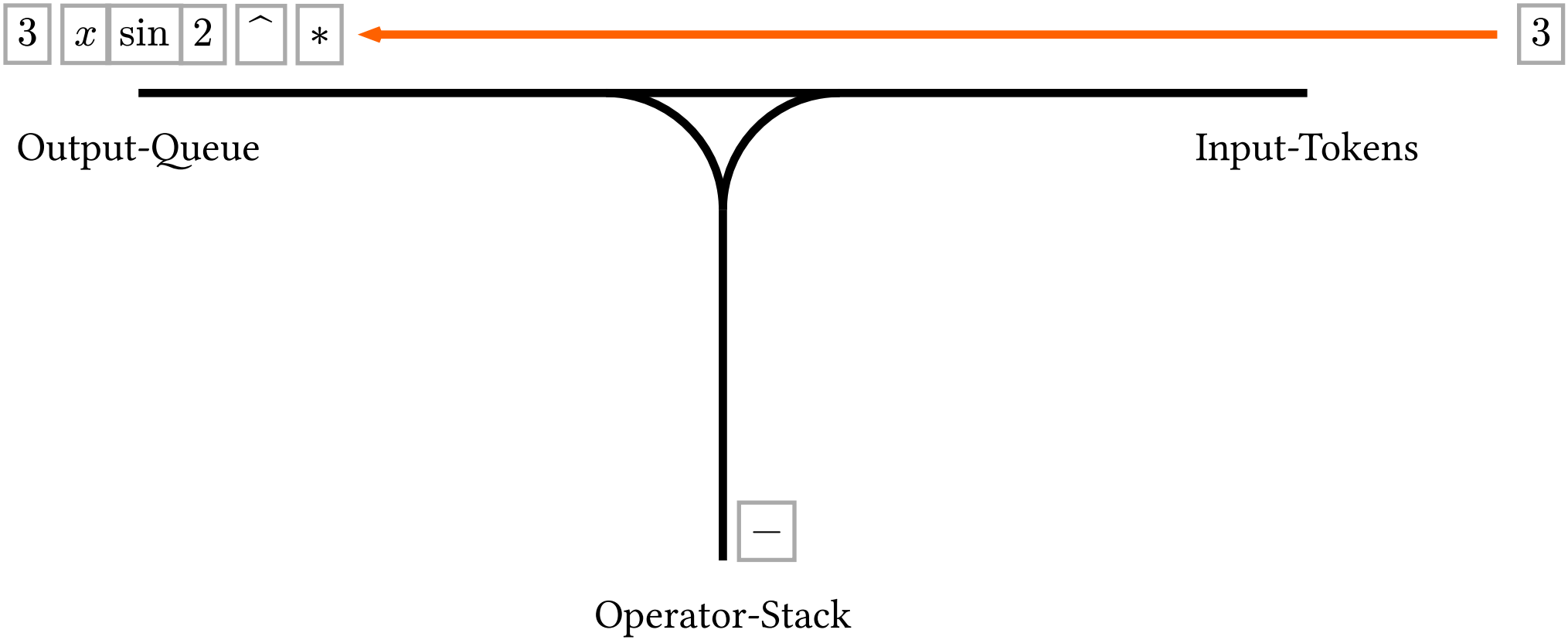
Infix- → Postfix-Notation

Algorithmus: Shunting Yard



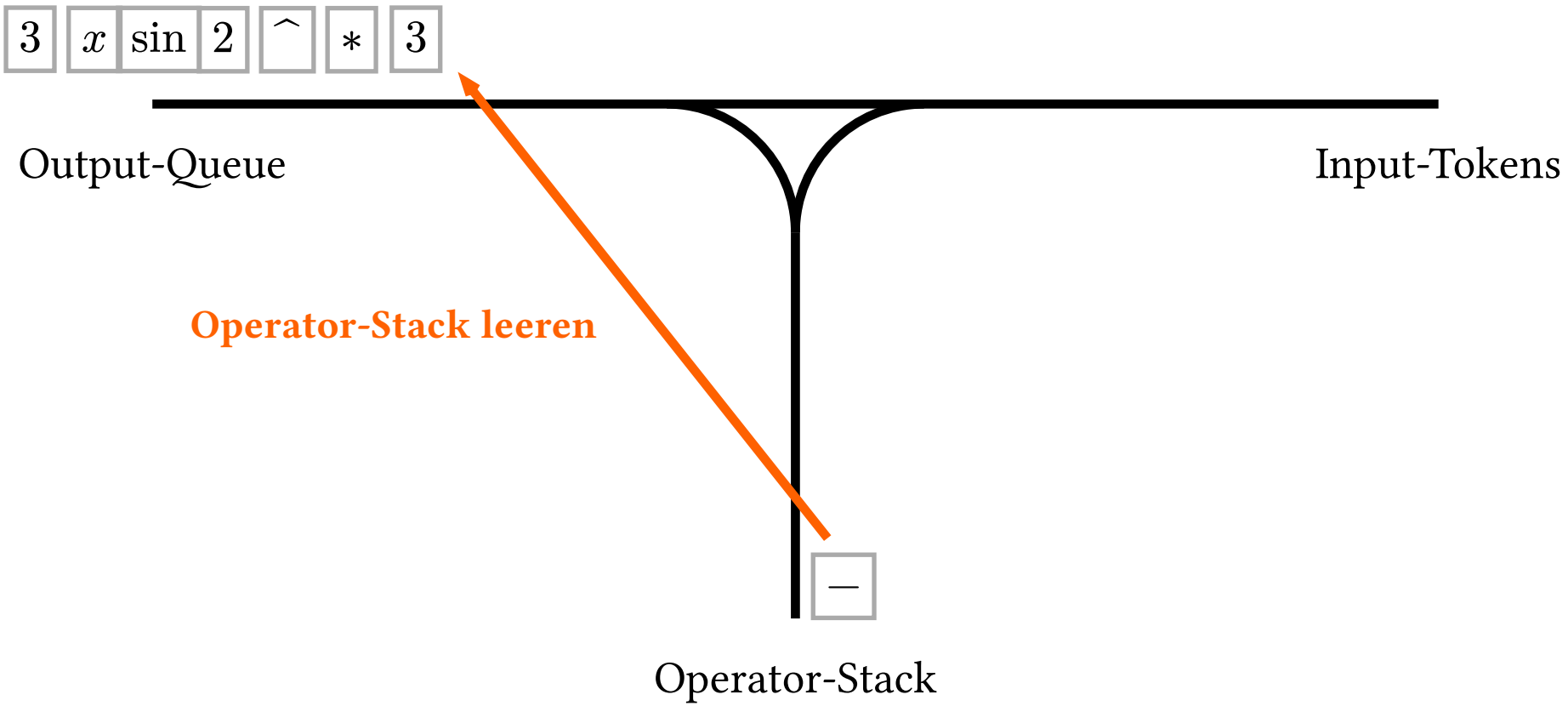
Infix- → Postfix-Notation

Algorithmus: Shunting Yard



Infix- → Postfix-Notation

Algorithmus: Shunting Yard



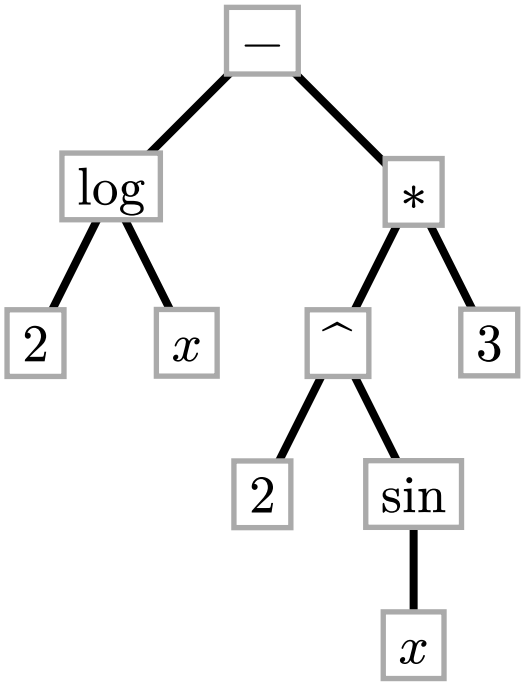
Infix- → Postfix-Notation

Algorithmus: Shunting Yard

Tada, wir sind fertig!

Postfix-Notation → Expression-Tree

Postfix-Notation: 3 x $\sin$ 2 $\hat{}$ $*$ x 2 $\log$ $-$



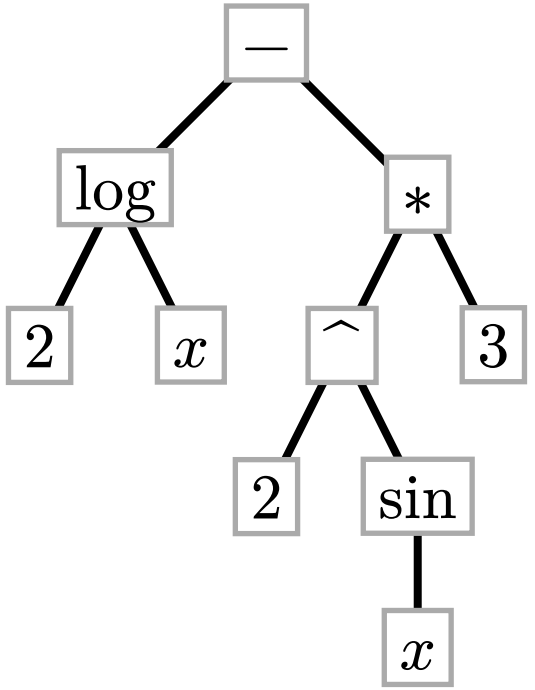
Baumbildungsalgorithmus:

- **Notation von links nach rechts durchgehen**
- **Operanden werden auf Stack gepackt**
- **Operatoren nehmen stets genau so viele Operanden vom Stack, wie sie für ihre Operation benötigen**
- **Fehlerzustand, falls Operator nicht genügend Operanden auf Stack hat oder, falls am Ende noch Operanden übrig bleiben**
- **Implementierung zu finden in: LunarMathematics/Expressions/Parsing/PostfixToTreeParser.cs**

Was stellen wir damit an?

- Ergebnis per Rekursion ermitteln
- Nach Variablen auflösen
- Scriptsprache basteln
- uvm. …

Scriptsprache bitte sinnvoll d.h. nicht so, wie Stand 17.12.2025 bei der Präsentation gezeigt, implementieren :)



Achtung! Paremeterreihenfolge bei Operationen/Funktionen beachten!

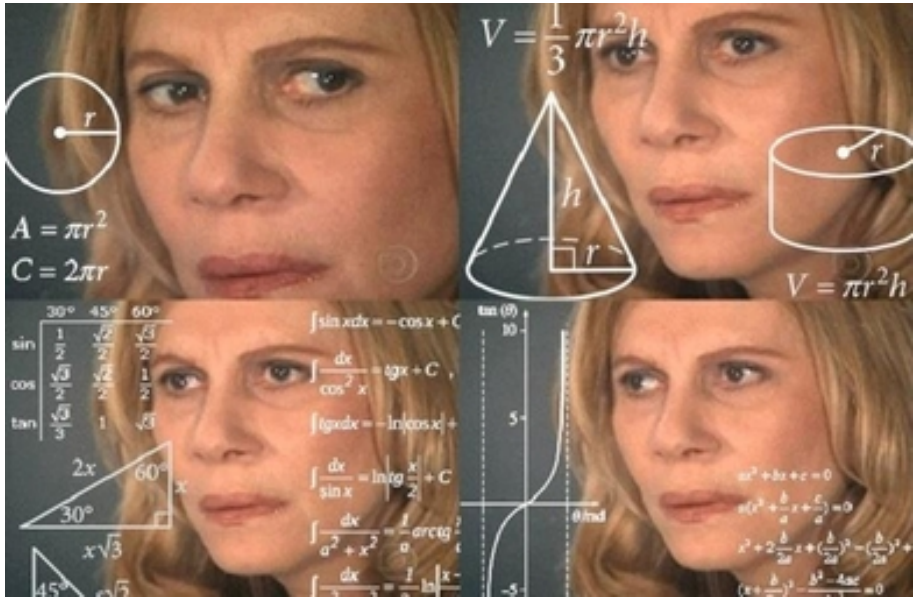
Zahlendarstellung

Ziel

- Möglichst große Zahlen darstellen können
- Beliebige Anzahl an Nachkommastellen
- Geringer Implementierungsaufwand
- Effiziente Rechenoperationen

Motivation ist, dass double (bzw. in C# auch “decimal”) einen zu kleinen Wertebereich für bspw. 100! besitzen;

… es folgen einige Schnapsideen :)



Idee 1: String-Basiert

```
public class Number {
    private string decimalRepresentation = [...];

    [...]
}
```

Probleme

- Arith. Algorithmen müssen vollständig selbst implementiert werden
- Division/Modulo kann unendlich lang dauern
- Laufzeit der arith. Algorithmen recht hoch
- Sehr großer RAM-Fußabdruck

Idee 2: Zwei Byte-Arrays

```
public class Number {
    private byte[] fractionPart = [...];
    private byte[] wholePart = [...];
    private byte sign;

    [...]
}
```

Probleme

- Arith. Algorithmen müssen vollständig selbst implementiert werden
- Division/Modulo kann unendlich lang dauern
- Manuelles Management der Array-Längen nötig

Idee 3: Byte-Array mit Komma-Indikator

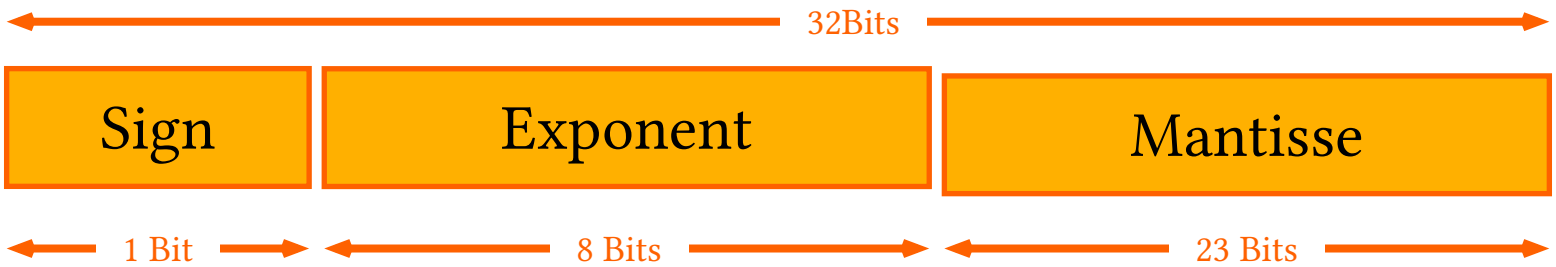
```
public class Number {
    private int commaPosition = [...];
    private byte[] digits = [...];
    private byte sign;

    [...]
}
```

Probleme

- Arith. Algorithmen müssen vollständig selbst implementiert werden
- Division/Modulo kann unendlich lang dauern
- Operationen mit verschiedenen Komma-Positionen aufwändig

Idee 4: Floatingpoint-Math

Single Precision nach IEEE 754




```java
public class Number {
    private BigInteger mantissa = [...];
    private int exponent = [...];

    [...]
}
```

Erste sinnvolle Idee, aber Trade-off bei Präzision, da Fracions, wie $\frac{1}{3}$ zu endlicher Abfolge von 0.333... wird, statt als Fracion beibehalten zu werden

Verwende BigInteger-Implementierung der jeweiligen Sprache

Exponentenverschiebung

Warum?

Wichtig für richtige Rechenergebnisse

Anwendung der Potenzgesetze:

$$3,1234 \cdot 10^3 + 5,01 \cdot 10^5 = 3,1234 \cdot 10^3 + 501 \cdot 10^3 = (3,1234 + 501) \cdot 10^3$$

```
private static Tuple<Number, Number> WithAlignedExponents(Number a, Number b)
{
    int d = Math.Abs(a.exponent - b.exponent);
    if (a.exponent > b.exponent)
    {
        a = new Number(a.mantissa * BigInteger.Pow(Number.Base, d), b.exponent);
    }
    else if (a.exponent < b.exponent)
    {
        b = new Number(b.mantissa * BigInteger.Pow(Number.Base, d), a.exponent);
    }
    return new Tuple<Number, Number>(a, b);
}
```

Rechenoperationen

```
1  public static Number operator +(Number first, Number second)
2  {
3      (first, second) = Number.WithAlignedExponents(first, second);
4      return new Number(first.mantissa + second.mantissa, first.exponent);
5  }
6
7  public static Number operator -(Number first, Number second)
8  {
9      (first, second) = Number.WithAlignedExponents(first, second);
10     return new Number(first.mantissa - second.mantissa, first.exponent);
11 }
12
13 public static Number operator *(Number first, Number second)
14 {
15     int exp = first.exponent + second.exponent;
16     return new Number(first.mantissa * second.mantissa, exp);
17 }
```

Implementierung der Operationen zu sehen in: LunarMathematics/Numbers/Number.cs

Rechenoperationen

```
1  public static Number operator /(Number first, Number second)
2  {
3      if (second.mantissa.IsZero)
4          throw new DivideByZeroException();
5
6      const int scale = 100;
7      BigInteger scaled = first.mantissa * BigInteger.Pow(Number.Base, scale);
8      BigInteger quotient = scaled / second.mantissa;
9      int exponent = first.exponent - second.exponent - scale;
10
11     return new Number(quotient, exponent);
12 }
13
14 public static Number operator %(Number first, Number second)
15 {
16     (first, second) = Number.WithAlignedExponents(first, second);
17     return new Number(first.mantissa % second.mantissa, first.exponent);
18 }
```

Implementierung der Operationen zu sehen in: LunarMathematics/Numbers/Number.cs

Finale Anmerkungen

- Dieses Projekt ist aus reinem Interesse entstanden, in der heutigen Zeit würde man das Parsing eher über spezialisierte Libraries abwickeln
- Richtige CAS-Systeme implementieren Zahlendatentypen, die periodische Brüche oder Vielfache von Konstanten nicht auflösen müssen und daher bei langen Rechnungen bessere Ergebnisse liefern
- Der Stand 17.12.2025 implementierte Math-Interpreter ist aus zeitlichen Gründen nicht gut designed; ihm fehlt u.a. die Möglichkeit, mehrzeilige Skripte nativ zu parsen, sowie ein Stack, sodass es zu keinen StackOverflow-Exceptions der CLR kommt
- StackOverflow-Exceptions der CLR sind non-recoverable, weshalb die Anwendung in jedem Fall ohne Fehlerbehandlung abstürzt, falls eine solche Exception auftritt
- Mit einem eigenen Zahlendatentyp wird auch die erneute Implementierung oft genutzter Funktionen (bspw. $\sin$, $\cos$, $\tan$, $\sqrt{x}$, ...) nötig. Für bspw. die trigonometrischen Funktionen existieren Polynomapproximationen
- Für manche Funktionen ist es außerdem sinnvoll bekannte Rechenregeln (bspw. Logarithmus-Basiswechsel oder $\cos(x) = \sin(x + \frac{\pi}{2})$) anzuwenden, um Implementierungsaufwand zu sparen
- Neuimplementierungen der Funktionen zu finden in: LunarMathematics/Numbers/NMath.cs

Projekt zu finden unter:

Info-Gitlab

https://gitlab2.informatik.uni-wuerzburg.de/s457701/mathparser

GitHub

https://github.com/lunardoggo/LunarMathematics

Side-Note: Das Projekt kann sich seit der Präsentation weiterentwickelt haben :)