Prof. Dr. Sven Strickroth

Einführung in die Programmierung

Wintersemester 2021/22

Grundkonzepte der Programmierung V: Funktionaler Algorithmus und Rekursion

Übersicht

1. Funktionaler Algorithmus
2. Rekursion
3. Lambda-Ausdruck
    a. Streams
4. Beispiel: Rekursive Grammatik

Funktionaler Algorithmus

- Funktionen stellen eine Abbildung von Elementen aus dem Definitionsbereich auf Elemente aus dem Bildbereich dar, haben also einen Rückgabewert
- keine Schleifen, nur Rekursion
- In rein funktionalen Sprachen gibt es keine Zuweisungen und keine Seiteneffekte.
  - keine Anweisungen, sondern nur Ausdrücke
  - Funktionen liefern bei gleichen Eingaben immer den gleichen Wert zurück
  - „Das ganze Programm ist eine Funktion bzw. Verschachtelung von Funktionen“
  - → Kern der Vorlesung „Programmierung und Modellierung“

- Java ist KEINE funktionale Sprache
  - enthält aber funktionale Konzepte (insbesondere seit Java 8/9)
  - Funktionale Algorithmen ermöglichen in manchen Fällen schlankeren und besser wiederverwendbaren Quellcode

Reinfunktionale Prozedur / Funktion

- Funktion im mathematischen Sinne
  - keine Seiteneffekte
  - immutable

- add: $\text{Integer} \times \text{Integer} \to \text{Integer}, (x, y) \to x + y$

```
static int add(int x, int y) {
  return x+y;
}
```

- mul: $\text{Integer} \times \text{Integer} \to \text{Integer}, (x, y) \to x \cdot y$

```
static int mul(int x, int y) {
  return x*y;
}
```

- sig: $\text{Integer} \to \text{Integer}, x \to \begin{cases} 1, \text{falls } x \geq 0 \\ -1, \text{falls } x < 0 \end{cases}$ (hier nur vereinfacht)

```
static int sig(int x) {
  return x >= 0 ? 1 : -1; // bedingter Ausdruck, vgl. nächste Folie
}
```

- Funktionen können für Berechnungen „verschachtelt“ werden:
  add(mul(3, 4), add(3, 5))

Bedingter Ausdruck

- Manchmal wird auch ein bedingter Ausdruck

```
static int sig(int x) {
  return x >= 0 ? 1 : -1;
}
```

- statt einer if-Anweisung verwendet (beides ist äquivalent)

```
static int sig(int x) {
  if (x >= 0)
    return 1;
  else
    return -1;
}
```

- Bedingter Ausdruck in Java:
- <Bedingung> ? <Dann-Wert> : <Sonst-Wert>
- <Bedingung> ist ein Ausdruck vom Typ boolean, die Ausdrücke <Dann-Wert> und <Sonst-Wert> haben einen beliebigen aber identischen Typ.

Übersicht

1. Funktionaler Algorithmus
2. Rekursion
3. Lambda-Ausdruck
    a. Streams
4. Beispiel: Rekursive Grammatik

Rekursion

„Um Rekursion zu verstehen, muss man zuerst einmal Rekursion verstehen.“

Rekursion

Rekursion

Eine Funktion F ist rekursiv, falls F vom Rumpf der Definition von F aufgerufen wird.

- Rekursion bietet sich immer dann an, wenn man die Lösung eines Problems auf die Lösung gleichartiger, aber kleinerer Teilprobleme, zurückführen kann.
  - z. B. Induktiv definierte Berechnungen und Datenstrukturen
- Wichtig: Sicherstellung der Terminierung!

```
Funktion f(int n) : int
  Wenn nicht fertig:
    Gib zurück: Berechnung mit f(n-1)
  Gib zurück: Konstante
```

Beispiel: Fakultätsfunktion

- $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n$
- Intuitiv: Einfach durch Wiederholungsanweisung berechenbar
- Anderer Ansatz:
  „Wenn man Ergebnis von x=(n-1)! kennt, kann man n! einfach als n*x berechnen.“

$$n! = \begin{cases} 1 & \text{falls } n \leq 1 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static int fac(int n) {
  if (n <= 1)
    return 1;
  else
    return n * fac(n - 1);
}
```

Aufrufhierarchie


```
public static int fac(int n) {
  if (n <= 1)
    return 1;
  else
    return n * fac(n - 1);
}
```

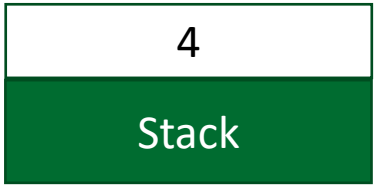
Stack

Aufrufhierarchie

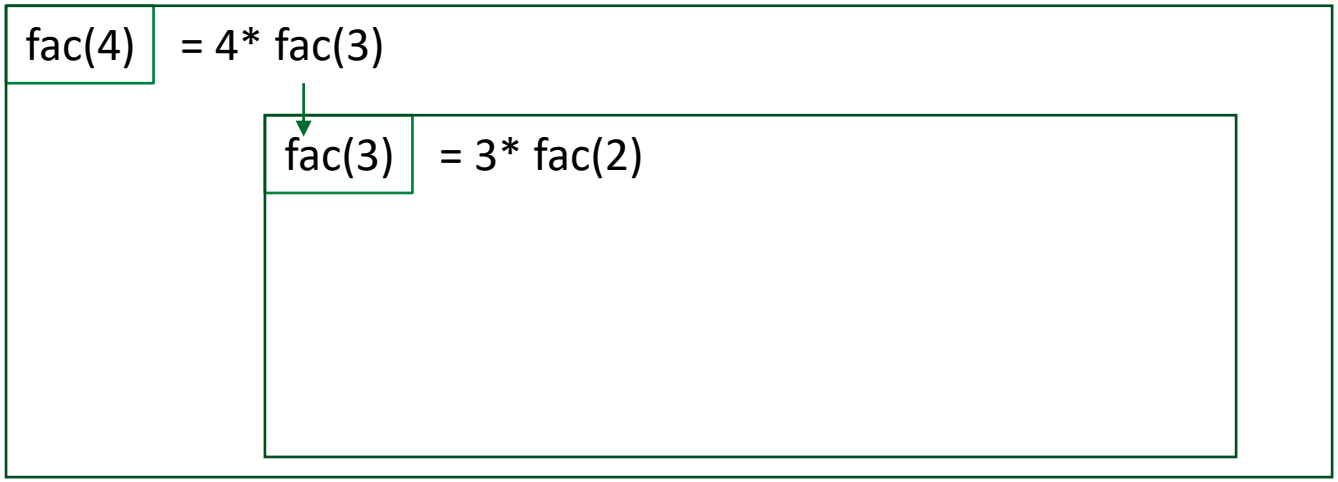
fac(4)	= 4* fac(3)

```
public static int fac(int n) {
  if (n <= 1)
    return 1;
  else
    return n * fac(n - 1);
}
```

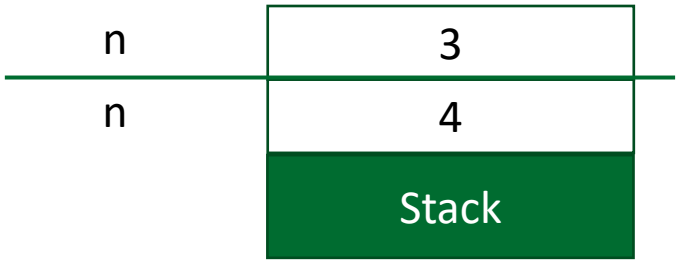
n



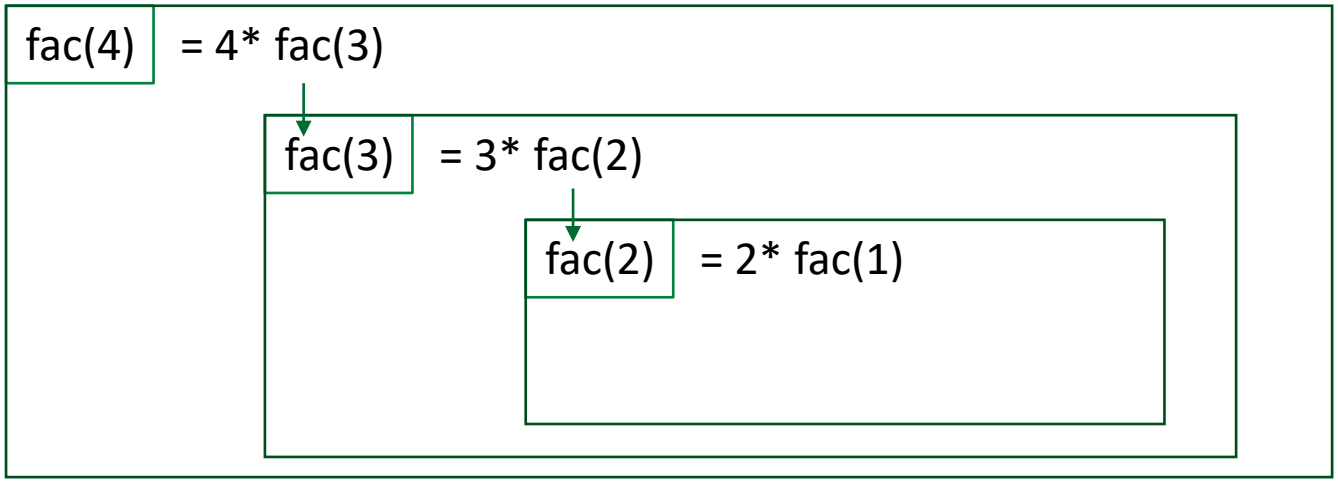
Aufrufhierarchie



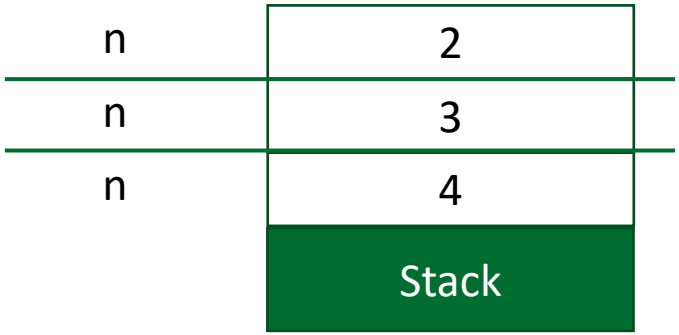

```
public static int fac(int n) {
  if (n <= 1)
    return 1;
  else
    return n * fac(n - 1);
}
```



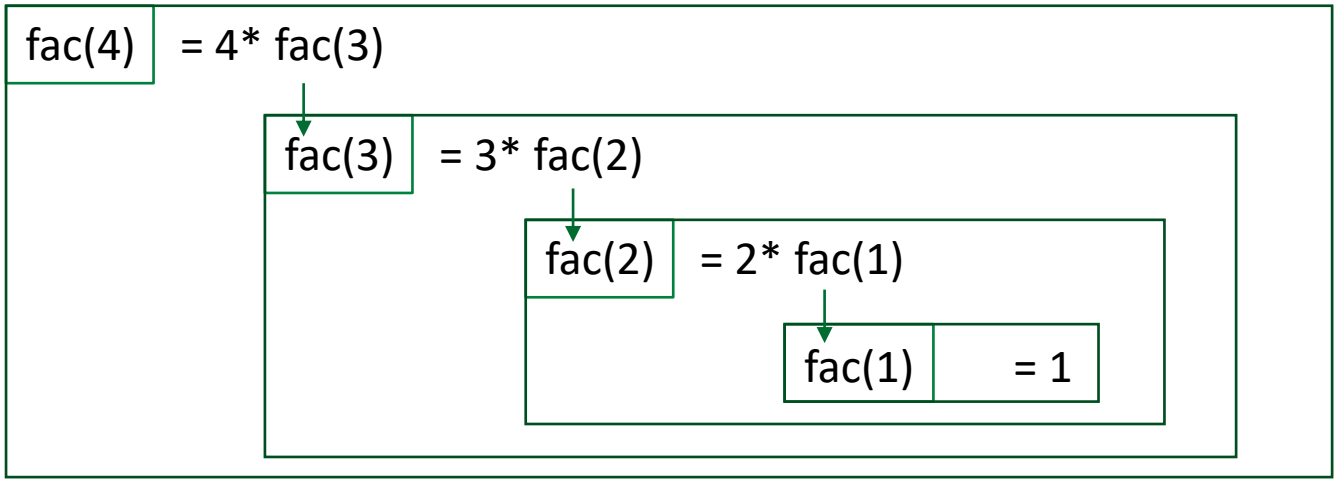
Aufrufhierarchie



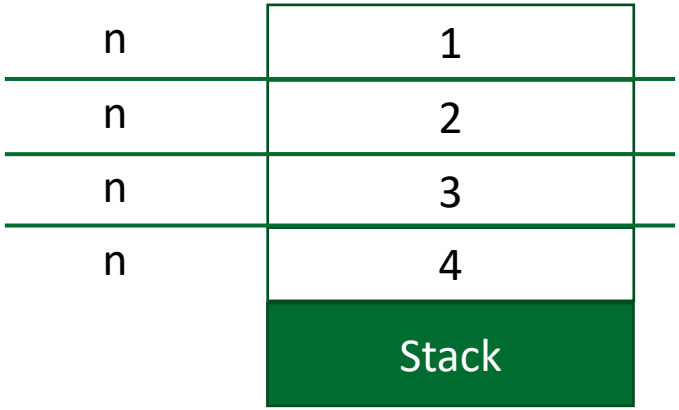

```
public static int fac(int n) {
  if (n <= 1)
    return 1;
  else
    return n * fac(n - 1);
}
```



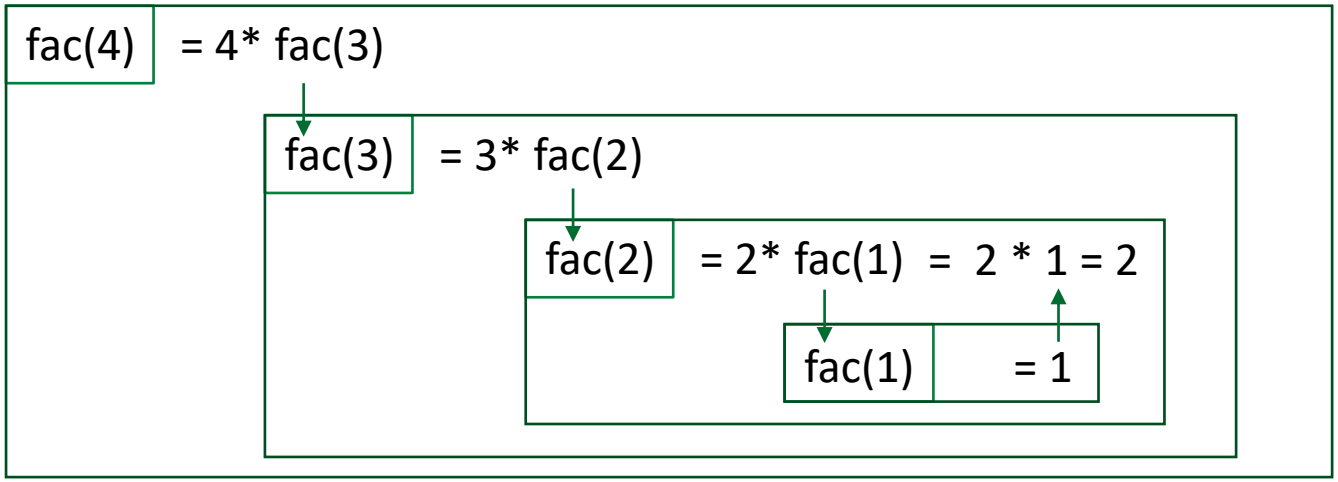
Aufrufhierarchie



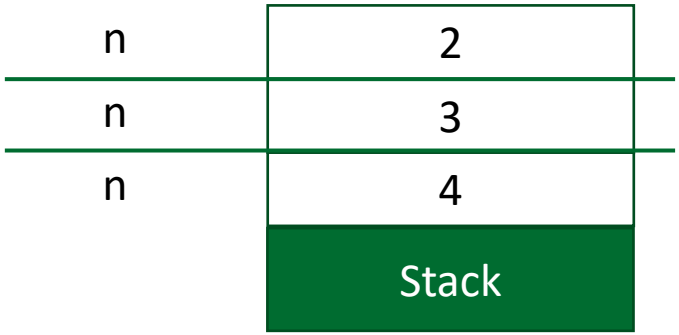

```
public static int fac(int n) {
  if (n <= 1)
    return 1;
  else
    return n * fac(n - 1);
}
```



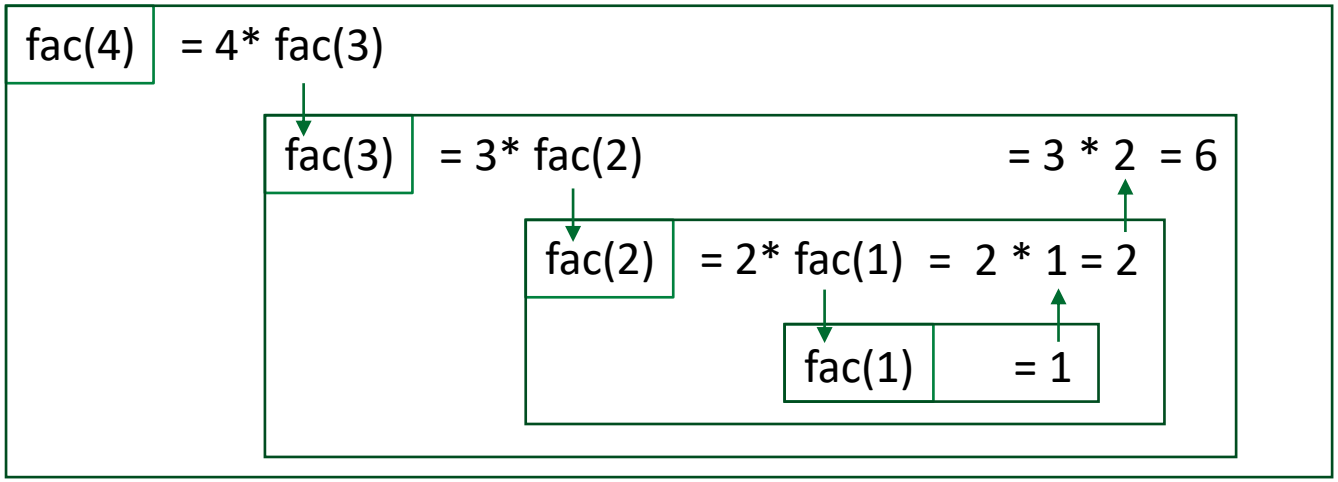
Aufrufhierarchie



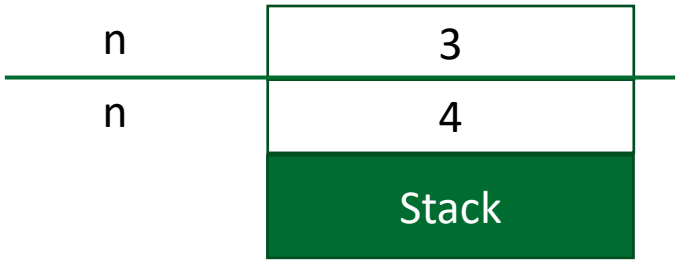

```
public static int fac(int n) {
  if (n <= 1)
    return 1;
  else
    return n * fac(n - 1);
}
```



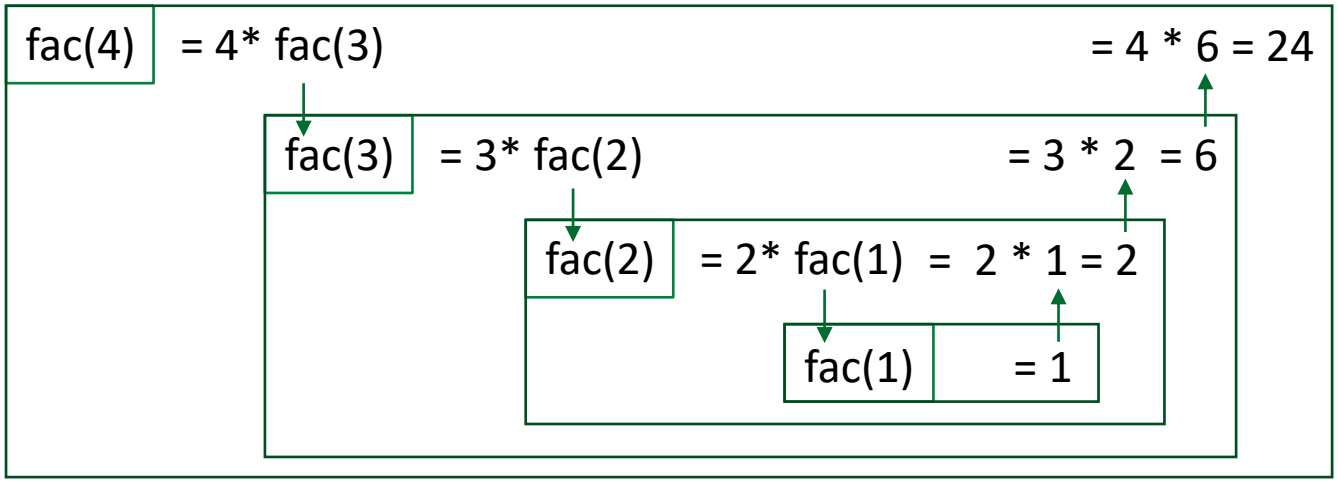
Aufrufhierarchie




```
public static int fac(int n) {
  if (n <= 1)
    return 1;
  else
    return n * fac(n - 1);
}
```

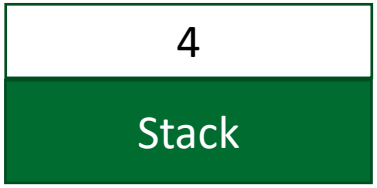


Aufrufhierarchie

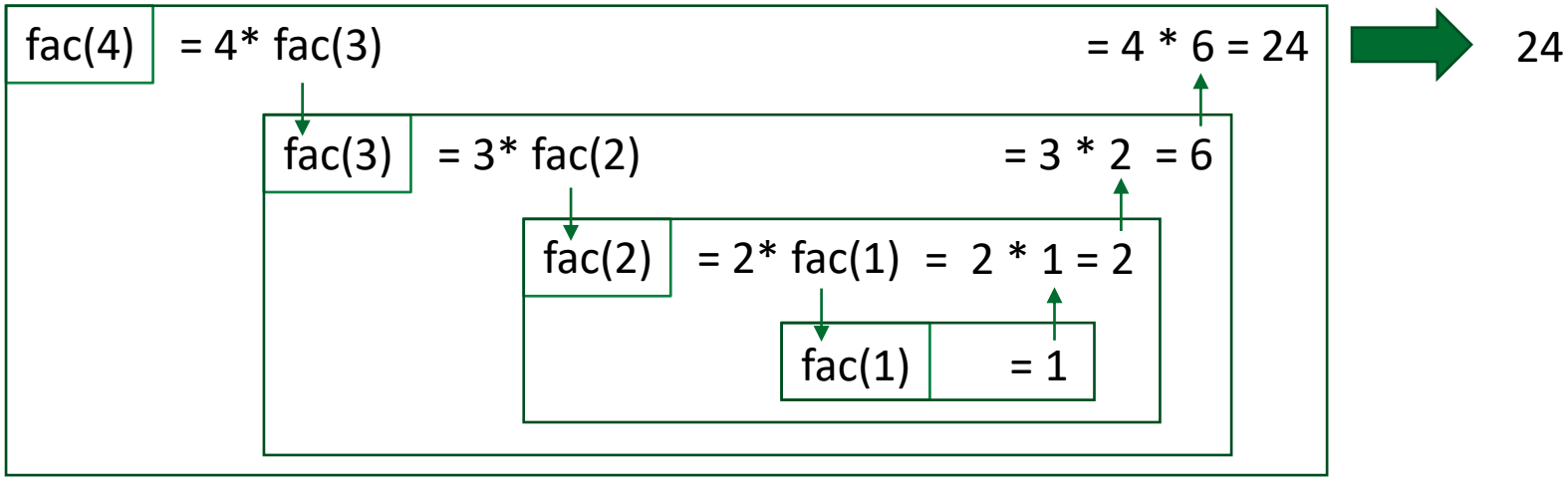



```
public static int fac(int n) {
  if (n <= 1)
    return 1;
  else
    return n * fac(n - 1);
}
```

n



Aufrufhierarchie



24

```
public static int fac(int n) {
  if (n <= 1)
    return 1;
  else
    return n * fac(n - 1);
}
```

Stack

Nicht-terminierende Rekursion in der Praxis

```
public static int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n - 1);
}
```

- Was passiert, wenn n negativ ist (z.B. fac(-7))?

```
% java FakultaetRekNeg
Fakultät von -7
Exception in thread "main" java.lang.StackOverflowError
at FakultaetRekBNeg.fac(FakultaetRekBNeg.java:9)
at FakultaetRekBNeg.fac(FakultaetRekBNeg.java:9)
at FakultaetRekBNeg.fac(FakultaetRekBNeg.java:9)
at FakultaetRekBNeg.fac(FakultaetRekBNeg.java:9)
at FakultaetRekBNeg.fac(FakultaetRekBNeg.java:9)
…
```

Terminierungsbeweis (Prinzip)

```
public static int fac(int n) {
  return (n <= 1) ? 1 : n*fac(n-1);
}
```

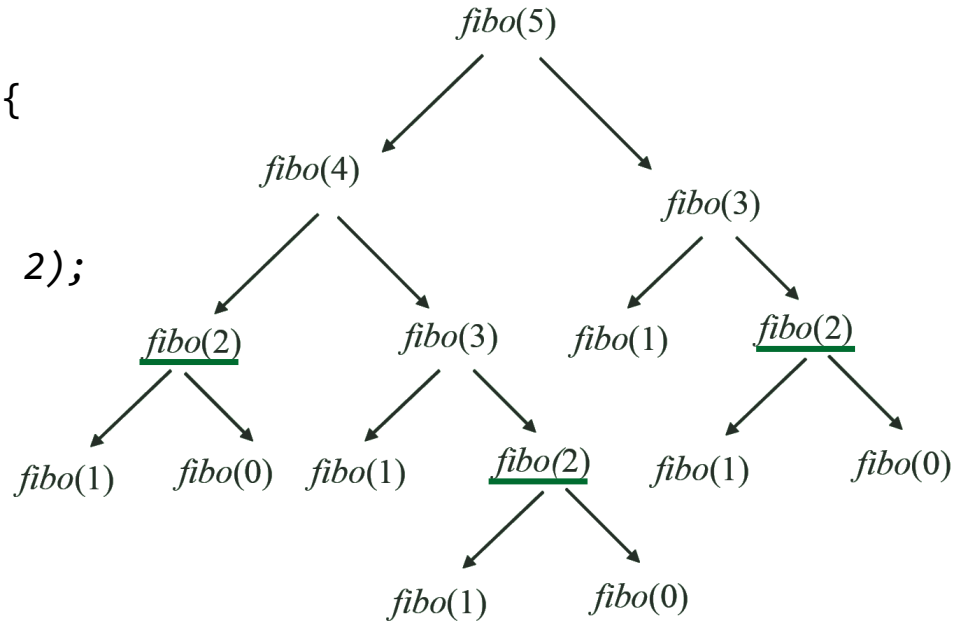
- *Terminierung* ist eine oft erwünschte Eigenschaft von Algorithmen.
- Wir beweisen für die Funktion *fac*, dass sie immer terminiert, wenn sie auf eine Zahl >= 0 angewendet wird.
- Beweis durch vollständige Induktion über *n*
  - **Induktionsanfang** n = 0:
    fac(0) liefert durch eine Fallunterscheidung direkt das Ergebnis
  - **Induktionsschritt** n → n+1:
    Ind.-Annahme: fac(n) benötigt endlich viele Schritte
    Falls n > 0: fac(n+1) = (n+1)*fac(n+1–1) = (n+1)*fac(n),
    also so viele Schritte wie fac(n) (endlich) und eine Operation.
    Falls n = 0: wie Induktionsanfang.

Fibbonacci-Zahlen

- $$fibo(n) = \begin{cases} 1, \text{falls } n = 0 \\ 1, \text{falls } n = 1 \\ fibo(n-1) + fibo(n-2), \text{falls } n \geq 2 \end{cases}$$

- Für n>=0: 1, 1, 2, 3, 5, 8, 13, 21, …

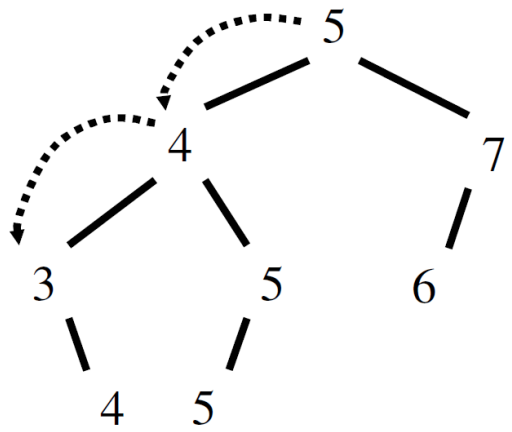
- ```
  public static long fibo(int n) {
    if (n == 0)
      return 1;
    if (n == 1)
      return 1;
    return fibo(n - 1) + fibo(n - 2);
  }
  ```



- Exzessive Neuberechnungen
  → nicht optimal (ohne weitere Optimierungen)
  ```

Weitere Beispiele

- Berechnung des größten gemeinsamen Teilers (Euklidischer-Algorithmus)
  - ```
    GGT(a,b)
      wenn b = 0 dann
         Ergebnis = a
      sonst
         Ergebnis = GGT(b, a % b)
    ```
- Binäre Suche
- Traversierung von Bäumen
- QuickSort, MergeSort, …
- …


    ```

Zusammenfassung Rekursion

- Rekursive Verfahren beruhen auf der Zerlegung eines Problems in kleinere gleichartige Probleme.

- Mit Hilfe der Rekursion lassen sich einige Probleme sehr viel eleganter und kompakter lösen als mit der Iteration.
- Für jede iterative Realisierung kann man einfach eine rekursive Implementierung angeben. Umgekehrt ist das nicht immer so einfach.

- Auf Abbruchbedingung/Basisfall aufpassen!

- Rekursionstiefe ist in Java limitiert
(in funktionalen Sprachen gibt es Optimierungen, z.B. Endrekursion)

- Weitere Details zu Rekursion und Funktionalen Sprachen
→ Vorlesung „Programmierung und Modellierung“

Übersicht

1. Funktionaler Algorithmus
2. Rekursion
3. Lambda-Ausdruck
    a. Streams
4. Beispiel: Rekursive Grammatik

Motivation: Array sortieren

- Wir haben ein Array und wollen dieses Sortieren

```
String[] names = { "Svenja", "Uwe", "Fabian", "Christina" };
java.util.Arrays.sort(names);
System.out.println(java.util.Arrays.toString(names));
→ [Christina, Fabian, Svenja, Uwe]
```

- `java.util.Arrays.sort` benutzt das `java.util.Comparable`-Interface, das von der Klasse `String` implementiert wird:

  `int String.compareTo(String s)`

  - wenn: this.equals(s) → 0
    sonst: lexikographischer Vergleich nach Unicode-Tabelle
    wenn der String this vor s kommt → < 0
    wenn der String this nach s kommt → > 0

  - Beispiel: "AAAaaaa".compareTo("Test") → < 0

Das Comparable-Interface (leicht vereinfacht)

```
public interface Comparable {
  int compareTo(Object other);
}
```

- Wenn o mit other vergleichbar ist, dann sollte gelten:
  - `o.compareTo(other) < 0`, falls o kleiner other
  - `o.compareTo(other) == 0`, falls o gleich other
  - `o.compareTo(other) > 0`, falls o größer als other
- Vom Prinzip können wir die Sortieralgorithmen aus der letzten Vorlesung benutzen und einfach `x < y` durch `x.compareTo(y) < 0` ersetzen.

Parametrisierung von Algorithmen

- Was wenn wir jetzt nach der Länge der Namen oder rückwärts sortieren wollen? Müssen wir dann einen komplett neuen Sortieralgorithmus schreiben oder von String erben?
- Idee:
  Wir übergeben `java.util.Arrays.sort` als Parameter eine Funktion, die die Vergleiche durchführt und jeweils einen Wert <0, ==0, >0 zurückgibt.
  → Parametrisierung von Algorithmen
- In Java können keine Funktionen als Parameter übergeben werden
  - aber Objekte, die bestimmte Methoden (eines Interfaces) implementieren
- Beispiel (leicht vereinfacht):

```
public static void sort(Object[] a, Comparator c)
```

  mit:

```
public interface Comparator {
  int compare(Object o1, Object o2);
}
```

  Rückgabewerte von `compare` wie bei `compareTo` des `Comparable`-Interfaces.

- Verallgemeinerung: Mit diesem Ansatz kann man auch Arrays beliebiger Klassen nach beliebigen Attributen sortieren.

Erster Ansatz für eine Implementierung

- Wir schreiben eine Klasse StringLengthComparator für den Vergleich:

```
public class StringLengthComparator implements
                                          java.util.Comparator {
  @Override
  public int compare(Object o1, Object o2) {
    return Integer.compare(((String)o1).length(),
            ((String)o2).length());
  }
}
```

In Kapitel 11 lernen wir eine schönere Lösung kennen

-
```
String[] names = { "Svenja", "Uwe", "Fabian", "Christina" };
java.util.Arrays.sort(names, new StringLengthComparator());
System.out.println(java.util.Arrays.toString(names));
→ [Uwe, Fabian, Svenja, Christina]
```

- Nachteil:
  - neue Klasse, die wir evtl. nur genau einmal brauchen

Zweiter Ansatz für eine Implementierung

- Wir definieren die Methode direkt an der passenden Stelle mit Hilfe einer sogenannten „anonymous inner class“

- ```
  String[] names = { "Svenja", "Uwe", "Fabian", "Christina" };
  java.util.Arrays.sort(names, new java.util.Comparator() {
    @Override
    public int compare(Object o1, Object o2) {
      return Integer.compare(((String)o1).length(),
                             ((String)o2).length());
    }});
  System.out.println(java.util.Arrays.toString(names));
  → [Uwe, Fabian, Svenja, Christina]
  ```

- Wir können direkt im Quellcode, wo wir eine Instanz einer Implementierung eines Interfaces brauchen, dieses in Java direkt spezifizieren:
  ```
  new Interfacename() {
   // Implementierung aller Methoden des Interfaces
  }
  ```

- Das ist schon deutlich kürzer und eleganter, aber noch immer viel „overhead“ für eine Implementierung einer einzigen Methode
  ```

Funktionales Interface & Lambda-Ausdruck

- Seit Java 8 gibt es eine neue Syntax für in-place-Funktionsobjekte → Lambda-Ausdrücke (angelehnt an Funktionsdefinitionen)
- Ein Lambda-Ausdruck ist eine kompakte Schreibweise für eine namenlose Funktion eines Interfaces mit genau einer abstrakten Methodendefinition (→ funktionales Interface).
- Beispiel:

```
String[] names = { "Svenja", "Uwe", "Fabian", "Christina" };
java.util.Arrays.sort(names,
  (s1, s2) -> Integer.compare(s1.length(), s2.length())
);
System.out.println(java.util.Arrays.toString(names));
→ [Uwe, Fabian, Svenja, Christina]
```

- Erinnerung: Das *Comparator*-Interface hat genau die Methode `int compare(Object o1, Object o2)` → funktionales Interface
- Im Prinzip steht der Lambda-Ausdruck für ein „frisch“ erzeugtes Objekt für das geforderte Interface, dessen einzige Methode im Rumpf wie durch den Lambda-Ausdruck beschrieben implementiert ist.

Lambda-Ausdruck

- Allgemeine Syntax:
  - (Parameter) -> { Anweisungen }
- Beispiel: Addition zweier Werte
  - (double x, double y) -> { return x+y; }
- Beispiel: Multiplikation und Ausgabe zweier Werte
  - (double x, double y) -> { System.out.println(x*y); }

- Die Datentypen der Parameter können weggelassen werden, wenn der Java-Compiler diese eindeutig ableiten kann.
- Geschweifte Klammern können weggelassen werden, wenn die Anweisung nur aus „einer Zeile“ besteht.
- „return“ kann weggelassen werden, wenn zusätzlich statt einer Anweisung ein Ausdruck verwendet wird.

Beispiele „klassich“

```
interface IntTester { boolean test(int x); }
public class IstGerade implements IntTester {
  public boolean test(int x) { return x % 2 == 0; }
}

public class ClassicExample {
  public static boolean fueralle(int[] a, IntTester p) {
    boolean result = true;
    for (int x : a)
      result &= p.test(x);
    return result;
  }
  public static void main(String[] args) {
    int a[] = { 2, 4, 6, 8, 10, 122 };
    int b[] = { 4, 25, 289 };
    System.out.println(fueralle(a, new IstGerade())); // → true
    System.out.println(fueralle(b, new IstGerade())); // → false
  }
}
```

Beispiele „mit Lambdas“

- Verwendung von Lambdas im Beispiel

```
int a[] = { 2, 4, 6, 8, 10, 122 };
int b[] = { 4, 25, 289 };
System.out.println(fueralle(a, x -> x < 1000)); // → true
System.out.println(fueralle(b, x -> {
  int q = (int) Math.sqrt(x);
  return x == q * q;
})); // → true

IntTester kleinerAlsHundert = x -> x < 100;
System.out.println(kleinerAlsHundert.test(50)); // → true
System.out.println(fueralle(a, kleinerAlsHundert)); // → false
```

- Es gibt auch eine Kurzschreibweise mit doppeltem Doppelpunkt, um direkt eine Funktion mit geeigneter Signatur (wie im funktionalem Interface gefordert) anzugeben:
  - Für Instanzmethoden kann anstatt
    `System.out.println(fueralle(b, x -> obj.someFunction(x)));`
    auch
    `System.out.println(fueralle(b, obj::someFunction));`
    geschrieben werden
  - Für statische Methoden kann anstatt
    `System.out.println(fueralle(b, x -> Klasse.someFunction(x)));`
    auch
    `System.out.println(fueralle(b, Klasse.someFunction));`
    geschrieben werden.

Übersicht

1. Funktionaler Algorithmus
2. Rekursion
3. Lambda-Ausdruck
    a. Streams
4. Beispiel: Rekursive Grammatik

Streams („neu“ seit Java 8)

- Streams sind Folgen von Werten.
- Ein Stream besteht aus drei Elementen:
  1. Erzeugung (Quelle)
  2. Mehrere Transformationen der Elemente
  3. Aggregation/“Terminal Operation“
- Stream-Verarbeitung kann seriell oder auch parallel erfolgen.

- Idee einer Pipeline:

- Beispiel (sortierte Ausgabe der Quadrate der geraden Zahlen in a):

```
int[] a = {3, 2, 1, 4};
java.util.Arrays.stream(a)                  // Erzeugung
    .filter(x -> x%2 == 0)                  // Transformation
    .map(x -> x*x)                          // Transformation
    .sorted()                               // Transformation
    .forEach(System.out::println);          // Terminal Operation
```

Stream Erzeugung (Quelle)

- java.util.stream.Stream.of("live", "long", "and", "prosper")
- java.util.Arrays.stream(a) mit Array a
- java.util.stream.IntStream.range(int startInclusive, int endExclusive)
- Collections des Java Collection-Framework (vgl. Kapitel 11)
- …

Stream Transformatoren

- .filter(…)
  - reicht nur Elemente weiter, wenn Test true ist
- .map(...)
  - wendet eine Funktion auf alle Elemente an und gibt das Ergebnis weiter
- .limit(long n)
  - beendet den Stream nach *n* Elementen
- .skip(long n)
  - „überspringe“ *n* Elemente im Stream

- .sorted()
- .sorted(Comparator c)
- .distinct()
  - Duplikate werden entfernt

- …

- https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Stream Aggregation/“Terminal Operation“

- .forEach(…)
  - führt eine Anweisung/Block für jedes Element des Streams aus
- .toArray()
  - sammelt alle Elemente des Streams und gibt sie als Array zurück

- .count()
  - zählt alle ankommenden Elemente und gibt am Ende des Streams deren Anzahl (als long) zurück
- .allMatch(…)
  - gibt einen boolean zurück, der genau dann true ist, wenn die Bedindung für alle Elemente des Streams erfüllt ist
- .noneMatch(…)
  - gibt einen boolean zurück, der angibt, ob keines der Elemente des Streams die Bedingung erfüllt

- …

- https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Übersicht

1. Funktionaler Algorithmus
2. Rekursion
3. Lambda-Ausdruck
   a. Streams
4. Beispiel: Rekursive Grammatik

Beschreibung der Syntax einer Programmiersprache

- Programmiersprachen sind formale Sprachen, d. h. deren Syntax wird durch eine formale Grammatik definiert.
- Eine formale Grammatik ist ein Regelsystem mit dem die wohlgeformten Sätze einer Sprache gebildet werden können.
- Zur Definition einer formalen Grammatik oft verwendet:
  - Backus-Naur-Form (BNF)
  - Erweitere Backus-Naur-Form (EBNF)
  - …

- Grundprinzip der BNF:
  - Syntax ist durch eine Menge von Regeln festgelegt
    linkeSeite ::= rechteSeite
  - Jede Regel besteht aus einer linken und einer rechten Seite
  - Auf der rechten Seite können die „Zeichen des Programmcodes“ und *Platzhalter* stehen.
  - Ausgehend von einer Startregel, können alle gültigen Sätze abgeleitet (gebildet) werden.
  - Bei der Anwendung einer Regel wird jeweils die linke Seite durch die rechte Seite ersetzt.
  - Das geschieht solange, bis keine Platzhalter mehr vorhanden sind

Formale Grammatik (1)

- Beispiel einer BNF
  - Regeln zur Bildung gültiger Sätze
    - <SATZ> ::= <SUBJEKT> <PRÄDIKAT> <ADJEKTIV> "."
    - <SUBJEKT> ::= "Niels" | "Inge"
    - <PRÄDIKAT> ::= "arbeitet" | "erzählt"
    - <ADJEKTIV> ::= "lange" | "viel"
- Gültige Sätze dieser Sprache
  - Niels arbeitet lange.
  - Niels arbeitet viel.
  - Niels erzählt lange.
  - Niels erzählt viel.
  - Inge arbeitet lange.
  - Inge arbeitet viel.
  - Inge erzählt lange.
  - Inge erzählt viel.
- Ungültiger Satz dieser Sprache
  - Inge arbeitet wenig.

Formale Grammatik (2)

- Nicht mehr ersetzbare Symbole heißen Terminalsymbole
  - Zeichen der beschriebenen Sprache, oft kursiv oder in Anführungszeichen dargestellt
  - Die Menge aller Terminalsymbole heißt: Alphabet
- Alle ersetzbaren Symbole heißen Nicht-Terminalsymbole (die „Platzhalter“)
  - oft in Großbuchstaben und/oder in spitzen Klammern dargestellt

- BNF
  - ::= : „Zuweisungsoperator“ der linken zur rechter Seite einer Produktion
  - | : Alternative
  - ε : leeres Wort
- Erweiterung der BNF zu einer EBNF
  - ( ) : Klammerung
  - { } : Wiederholung (kein mal, einmal und beliebig oft; manchmal auch „(…)*“)
  - [ ] : Option (kein mal oder einmal)
  - (…)+ : Mindestens einmal

- Beispiel natürliche Zahlen (ohne führende Null, z.B. zur Definition von Integer-Literalen):
  - ```
    <NichtNullZiffer> ::= "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
    <Ziffer> ::= "0" | <NichtNullZiffer>
    <Dezimalzahl> ::= "0" | <NichtNullZiffer> { <Ziffer> }
    ```

- → Details in späterer Vorlesung zu formalen Grammatiken
    ```

Anwendungsfall Java

- Java ist über eine EBNF definiert (vereinfachtes Beispiel):

- ```
  <statement> ::= <type_expression> <ident> [= <expression>];
              ::= { (<statement>)+ }
              ::= if (<expression>) <statement>
                    [else <statement>]
              ::= for (<expression>; <expression>; <expression>)
                    <statement>
              ::= …
  ```

Spezifikation (sehr technisch; als Beispiel):
- https://docs.oracle.com/javase/specs/
  ```

Zusammenfassung

- Funktionale Algorithmen
  - keine Schleifen, nur Rekursion
- Rekursion
  - „Selbstaufruf“ einer Funktion/Methode im Rumpf
  - Rekursive Verfahren beruhen auf der Zerlegung eines Problems in kleinere gleichartige Probleme.
- Lambda-Ausdrücke
  - kurze Syntax zur in-place-Definition von anonymen Funktionen
  - in Java: erzeugen ein „frisches“ Objekt für ein funktionales Interface, dessen einzige Methode wie durch den Lambda-Ausdruck beschrieben implementiert wird
  - praktisch zur Parametrisierung von Algorithmen
- Streams
  - erlauben kurze Beschreibung komplexer Operationen
- BNF
  - Beispiel, wie die Syntax von Programmiersprachen mit einer rekursiven Grammatik beschrieben werden kann

Prof. Dr. Sven Strickroth

Ludwig-Maximilians-Universität München
Institut für Informatik
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67
80538 München

Telefon: +49-89-2180-9300
sven.strickroth@ifi.lmu.de