TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Lernziele

- Spezifikationen für Prozeduren lesen und verfassen
- Testfälle aus Spezifikationen ableiten
- Prozedurtests in JUnit implementieren, ausführen und die Ergebnisse interpretieren
- Rekursive Prozeduren zu charakterisieren.
- Terminierung von rekursiven Prozeduren mit Hilfe von geeigneten Abstiegsfunktionen zu beweisen

Spezifikation von Prozedureigenschaften

Spezifikation prozeduraler Programme

Wir betrachten hier grundlegende Techniken zur

- Spezifikation von Prozedureigenschaften
- Testen von Prozedureigenschaften

- Vertiefung Spezifikation in Modul "Formale Grundlagen der Programmierung"
- Vertiefung Testen in SE2, "Grundlagen des Software Engineering", "Software-Qualitätssicherung"
- Vertiefung Verifikation in "'Spezifikation und Verifikation mit Logik höherer Ordnung"

Warum Spezifikationen?

Spezifikationen sind wichtig

- zur Dokumentation,
- zum Testen durch dynamisches Prüfen,
- als Grundlage für die Verifikation mit Beweis.

Begriffsklärung: Vorzustand, Nachzustand

Den Zustand vor der Ausführung einer Prozedur nennen wir den **Vorzustand**, den Zustand nach Ausführung den **Nachzustand**.

Der Vorzustand beschreibt die aktuellen Parameter und den Inhalt der globalen Variablen vor Ausführung.

Der Nachzustand beschreibt den Inhalt der globalen Variablen nach Ausführung und das Ergebnis (sofern existent).

Zu den globalen Variablen zählen dabei auch Eingaben und Ausgaben auf Konsole (repräsentiert durch die globalen Variablen `System.in` und `System.out`), Dateien, etc.

Begriffsklärung: Vor-, Nachbedingung

Prozedureigenschaften lassen sich durch Vor- und Nachbedingungen beschreiben:

- Die **Vorbedingung** formuliert Anforderungen an den Vorzustand; wenn die Vorbedingung gilt, muss die Prozedur ohne Fehler terminieren.
- Die **Nachbedingung** formuliert die Eigenschaften des Nachzustands
  - → in Abhängigkeit vom Vorzustand (z.B. Parameterwerte);
  - → unter der Voraussetzung, dass beim Aufruf die Vorbedingung gilt.

Begriffsklärung: Prozedurspezifikation

Eine **Prozedurspezifikation** besteht aus:

- einer Vorbedingung:

  `requires` <Beschreibung>

- einer Einschränkung von Seiteneffekten:

  `modifies` <Liste von Variablen>

- einer Nachbedingung:

  `ensures` <Beschreibung>

Begriffsklärung: Prozedurspezifikation

Eine **Prozedurspezifikation** besteht aus:

- einer Vorbedingung:

  `requires <Beschreibung>`

- einer Einschränkung von Seiteneffekten:

  `modifies <Liste von Variablen>`

- einer Nachbedingung:

  `ensures  <Beschreibung>`

Eine Implementierung ist **korrekt** bezüglich einer Spezifikation, wenn für jeden Aufruf der Prozedur gilt:
Wenn die Vorbedingung zu Beginn des Aufrufs gilt, dann terminiert die Prozedur ohne Fehler, während des Aufrufs passieren nur die spezifizierten Seiteneffekte und nach dem Aufruf gilt die Nachbedingung.

In welcher Sprache schreiben wir Spezifikation?

„Although natural language is the ideal notation for most aspects of human communication, from love letters to introductory programming language manuals, there are cases where it is not appropriate. Software specifications, for example, require more rigorous formalism. [...]
In fact, mathematical specification of a problem usually leads to a better natural-language description. This is because formal notations naturally lead the specifier to raise some question that might have remained unasked, and thus unanswered, in an informal approach.“(Betrand Meyer, 1980)

Beispiel

```
/* Berechnet die Fakultaet von x.

   requires   0 ≤ x ≤ 12
   modifies   \nothing
   ensures    Ergebnis ist die Fakultaet von x
*/

public static int fac(int x) {
 int[] facres =
     {1,1,2,6,24,120,720,5040,40320,
      362880,3628800,39916800,479001600};
  return facres[x];
}
```

```
/* Berechnet die Fakultaetsfunktion fuer Zahlen zwischen 0
  und 12.

   modifies Eingabe und Ausgabe
   ensures  Das Programm druckt zunaechst "Parametereingabe:"
            und liest dann ein int n ein.
            Es gibt dann die Fakultaet von n aus, falls
            n groessergleich 0 und kleinergleich 12 ist.
            Andernfalls gibt es aus, dass die Berechnung
            fuer n nicht definiert ist.
*/

public static void main(String[] args ) {
  StdOut.println("Parametereingabe:");
  int n = StdIn.readInt();
  if( n < 0 || n > 12 ) {
    StdOut.println("Fuer "+ n + " nicht definiert");
  } else {
    StdOut.println("fac("+n+") = "+ fac(n));
  }
}
```

Beispiel: Arrays

```
/* Testet, ob ein Array sortiert ist.

   requires  f != null && laenge == f.length
   modifies  \nothing
   ensures   Ergebnis ist true, falls das Array aufsteigend
             sortiert ist.
             Andernfalls ist das Ergebnis false.
 */

public static boolean isSorted (int[] f, int laenge) {
  for(int i=0; i < laenge-1; i++) {
    if (f[i] > f[i+1]) {
      return false;
    }
  }
  return true;
}
```

Beispiel: Seiteneffekte

```
/* Vertauscht die Elemente des Arrays an Position i und j

   requires a != null && 0 ≤ i < a.length
                      && 0 ≤ j < a.length
   modifies a
   ensures  a[j] enthaelt den urspruenglichen Wert von a[i]
            und a[i] den urspruenglichen Wert von a[j]
*/

public static void swap(double[] a, int i, int j) {
    double t = a[i];
    a[i] = a[j];
    a[j] = t;
}
```

Beispiel: Seiteneffekte II

```
/* Vertauscht die Elemente des Arrays an Position i und j

   requires a != null && 0 ≤ i < a.length
                      && 0 ≤ j < a.length
   modifies a
   ensures  a[j] == \old(a[i])  && a[i] == \old(a[j])
*/

public static void swap(double[] a, int i, int j) {
    double t = a[i];
    a[i] = a[j];
    a[j] = t;
}
```

Bei Prozeduren, die globale Variablen oder referenzierte Objekte / Arrays modifizieren, bezeichnet `\old(...)` in der Nachbedingung den Wert vor Ausführung der Prozedur.

Beispiel: Seiteneffekte III

```
/* Vertauscht die Elemente des Arrays an Position i und j

   requires a != null && 0 ≤ i < a.length && 0 ≤ j < a.length
   modifies a
   ensures
      fuer alle k in [0, a.length-1] gilt:
            falls k == i, dann a[k] == \old(a[j])
            falls k == j, dann a[k] == \old(a[i])
            sonst gilt a[k] == \old(a[k])
*/

public static void swap(double[] a, int i, int j) {
    double t = a[i];
    a[i] = a[j];
    a[j] = t;
}
```

Warum sind Seiteneffekte wichtig? I

```
/*
   requires  a != null && ar.length > 0
   ensures   \result ist der Median der Eintraege in a
*/

public static double median(double[] a){
  ...
}
```

Angenommen, die Prozedur `median` wird in folgendem Code verwendet:

```
double[] punktzahl = ...;
String[] student = ...;
// hier gilt: punktzahl[i] gehoert zu student[i]
(fuer alle passenden i)
int bestehensgrenze = median(punktzahl); // fiktiv!

for (int i = 0; i < bestanden.length; i++) {
  if (punktzahl[i] >= bestehensgrenze) {
    StdOut.println(student[i] + " hat bestanden.");
  } else {
    StdOut.println(student[i] + " hat nicht bestanden.");
  }
}
```

Warum sind Seiteneffekte wichtig? II

```
/* requires  a != null && ar.length > 0
   ensures   \result ist der Median der Eintraege in a */

public static double median(double[] a){
  Arrays.sort(a); // Sortiert das Array a um!!
  if (a.length % 2 == 0) {
    return (a[a.length/2 - 1] + a[a.length/2]) / 2.0;
  } else {
    return (a[a.length/2]);
  }
}
```

Aufgabe

Was machen die folgenden beiden Prozeduren?
Geben Sie jeweils eine Spezifikation an!

```
public static double max(double[] a) {
  double max = a[0];
  for (int i = 1; i < a.length; i++) {
    if (a[i] > max) {
      max = a[i];
    }
  }
  return max;
}

public static int minPos(double[] a, int low, int high) {
  double min = a[low];
  int minPos = low;
  for (int i = low; i < high; i++){
    if (min > a[i]) {
      min = a[i];
      minPos = i;
    }
  }
  return minPos;
}
```

Lösungsvorschlag

```
/* Berechnet das Maximum der Arrayeintraege.

   requires  a != null  && a.length > 0
   modifies  \nothing
   ensures   Ergebnis ist groessergleich als alle
             Array-Eintraege und entspricht dem Wert
             (mind.) einer der Array-Eintraege
*/

/* Berechnet das Maximum der Arrayeintraege.

   requires  a != null  && a.length > 0
   modifies  \nothing
   ensures   Fuer int i in [0,a.length-1] : \result ≥ a[i]
             und es existiert ein int i, sodass
                  \result == a[i]
*/
```

Lösungsvorschlag

```
/* Ermittelt die Position des kleinsten Wertes eines Arrays
   aus dem Indexbereich zwischen low und high

   requires  a != null && 0 ≤ low < a.length
                       && high <= a.length
   modifies  \nothing
   ensures   Ergebnis ist die Position des kleinsten Wertes
             von a aus dem Indexbereich von [low, high-1]
*/

/* Ermittelt die Position des kleinsten Wertes eines Arrays
   aus dem Indexbereich [low,high-1]

   requires  a != null && 0 ≤ low < a.length
                       && high <= a.length
   modifies  \nothing
   ensures   Fuer int i in [low,high-1] : a[\result] ≤ a[i]
*/
```

Achten Sie auf Ihre Formulierungen!

- “ensures Das Ergebnis liefert die richtige Position”

Achten Sie auf Ihre Formulierungen!

- “ensures Das Ergebnis liefert die richtige Position”
- “requires Das Array muss int-Werte enthalten”

Achten Sie auf Ihre Formulierungen!

- “ensures Das Ergebnis liefert die richtige Position”
- “requires Das Array muss int-Werte enthalten”
- “requires Das Array muss sortiert sein”

Achten Sie auf Ihre Formulierungen!

- “ensures Das Ergebnis liefert die richtige Position”
- “requires Das Array muss int-Werte enthalten”
- “requires Das Array muss sortiert sein”
- “ensures Das Array wird zuerst kopiert und die Kopie wird sortiert”

Testverfahren

Softwaretests

Softwaretests sind eine der wichtigsten Maßnahmen zur Qualitätssicherung in der Software-Entwicklung.

„Ein Test [...] ist der überprüfbare und jederzeit wiederholbare Nachweis der Korrektheit eines Softwarebausteines relativ zu vorher festgelegten Anforderungen “(Denert, 1991)

„Program testing can be used to show the presence of bugs, but never show their absence! “(Edsger W. Dijkstra)

Komponententests

- **Komponententests** (Unit-Tests) testen die funktionale Anforderungen an einzelne Software-Komponenten
- Ist eine Funktion korrekt implementiert?
- Verschiedene Eingaben verwenden
- Vorbedingung $\Rightarrow$ Testeingabe
  Nachbedingung $\Rightarrow$ Ergebnis
- Erstellen der Testfälle am besten **vor** dem Implementieren (*test-driven development*)

Unit-Tests für Java

- Weitverbreitetes Framework zum Testen von Java-Programmen
- JUnit-Bibliothek auf der Vorlesungsseite

Beispiel: GCD.java

```
import static org.junit.Assert.*;
import org.junit.Test;

// Greatest common divisor; groesster gemeinsamer Teiler
public class GCD {
   public static int gcd(int a, int b){
      int x = a;
      int y = b;
      while (y != 0){
        int t = y;
        y = x % y;
        x = t;
      }
      return x;
    }

   @Test
   public void test1() {
     assertEquals("gcd von 5 und 10", 5, gcd(5,10));
   }

   @Test
   public void test2() {
     assertEquals(1, gcd(29,311));
   }
}
```

Ausführen der Tests

- Beim Kompilieren muss die Test-Bibliothek dem Klassenpfad (*class path*) hinzugefügt werden:

```
javac -cp junitrunner.jar GCD.java
```

- Die Tests können dann folgendermaßen ausgeführt werden:

```
java -jar junitrunner.jar GCD
```

- `junitrunner.jar` muss im gleichen Verzeichnis liegen!

Ergebnisse von Testläufen

- Falls alle Tests korrekte Ergebnisse liefern:

```
java -jar junitrunner.jar GCD
2 Tests erfolgreich ausgeführt!
Zeit: 6ms
```

- Falls ein Test fehlschlägt:
  Abändern des Algorithmus' von GCD in `while (b == 0)...`

```
> java -jar junitrunner.jar GCD
Failed: test2(GCD): gcd von 29 und 311
expected:<1> but was:<29>
 ... in class GCD line 21
1 von 2 Tests fehlgeschlagen.
Zeit: 8ms
```

Testmethodik

Ziel: Möglichst hohe Abdeckung des Codes durch Testfälle

- Jede Funktion sollte mit verschiedenen Eingaben getestet werden
- Randfälle sind besonders wichtig!
  - Bei Integer-Werten: 0, 1, -1, ...
  - Bei Arrays: Leere Arrays, einelementige Arrays, ...
  - Bei Strings: Leerer String, Strings der Länge 1, ...
- Möglichst jeder Ausführungspfad sollte durch die Tests abgedeckt werden.
- Verzweigungen geben Hinweise, wie Testfälle auszuwählen sind, um eine vollständige Abdeckung zu erhalten.

Aufgabe

Schreiben Sie drei Testfälle für folgende Prozedur!

```
/* Ermittelt die Position des kleinsten Wertes eines Arrays
   aus dem Indexbereich [low,high-1]

   requires  a != null && 0 ≤ low < a.length && high <= a.length
   ensures   Fuer int i in [low,high-1] : a[\result] ≤ a[i]
*/

public static int minPos(double[] a, int low, int high) {
    // ...
  }

@Test
public void test() {
  ...
  assertEquals(..., minPos(...));
}
```

Ideen I

```
@Test
  public void testStandard() {
      double[] a = {1,7,4,9,5,9,10};
      assertEquals(2,minPos(a,1,4));
  }
  @Test
  public void testNegativeEntries() {
      double[] a = {1,7,4,-9,5,9,10};
      assertEquals(3,minPos(a,0,7));
  }
 @Test
  public void testMinAtLow() {
      double[] a = {1,0,4,9,5,9,10};
      assertEquals(1,minPos(a,1,4));
  }
 @Test
  public void testMinAtHigh() {
      double[] a = {1,7,4,9,5,9,0};
      assertEquals(6,minPos(a,0,7));
  }
```

Ideen II

```
@Test
public void testHighSmallerThanLow() {
    double[] a = {1,4,7,9,5,4,10};
    assertEquals(5, minPos(a,5,3));
}

@Test
public void testOneElementArray() {
    double[] a = {3};
    assertEquals(0, minPos(a,0,1));
}

@Test
public void testMultipleMins() {
    double[] a = {1,7,4,9,1,9,10};
    int minpos = minPos(a,0,7);
    assertEquals(true, minpos == 0 || minpos == 4);
}
```

Test von Seiteneffekten

- Ein-/Ausgabe ist schwierig zu Testen mit JUnit
  $\Rightarrow$ System- und Integrationstests
- Testbare Seiteneffekte sind Modifikation von globalem Zustand
  (referenzierte Arrays und andere Objekte)

Beispiel

```
/* Multipliziert alle Eintraege im Array mit dem Wert factor
 *    requires ar != null
 *    modifies ar
 *    ensures  fuer alle in in [0,ar.length):
 *               ar[i] == \old(ar[i])*factor
 */
public static void scale(int[] ar, int factor) {
  for (int i=0; i<ar.length; i++) {
    ar[i] = ar[i] * factor;
  }
}

@Test
public void scaleTest() {
  int[] eingabe = {1, 2, 3};
  scale(eingabe, 2);
  int[] erwartet = {2, 4, 6};
  assertArrayEquals(erwartet, eingabe);
}
```

Test für die Abwesenheit von Seiteneffekten

```
/* Testet, ob ein Array sortiert ist.
      requires   f != null && laenge == f.length
      ensures
         \result == true, falls
           fuer alle int i in [0,laenge-2] gilt: f[i] ≤ f[i+1]
         \result == false, sonst
 */
  public static boolean istSortiert (int[] f, int laenge) {
    ...
  }

  @Test
  public void testModifications() {
    int[] a = {1,7,4,9,5,9,10};
    int[] copy = {1,7,4,9,5,9,10};
    assertEquals(false, istSortiert(a,7));
    assertEquals(true, Arrays.equals(a,copy));
  }
```

Prozeduren und Spezifikationen

- Spezifizieren ist oft anspruchsvoller als Programmieren.
- Der Aufrufer einer Prozedur ist dafür verantwortlich, dass die Vorbedingung gilt
- Korrekte Implementierung garantiert dann, dass die Nachbedingung erfüllt ist.
- Eine Prozedur ist **total**, wenn sie für alle Eingabewerte terminiert, die der Typ der Eingabe (insbesondere Parameter) umfasst und die dabei keinen Fehler meldet. Andernfalls ist eine Prozedur **partiell**.
- Die Spezifikation einer partiellen Prozedur sollte immer eine `requires`-Klausel enthalten.

Rekursion

Das Ende einer WG-Party!

- Sie laufen an der Küche vorbei und jemand gibt Ihnen den Auftrag:

  Spülen Sie das Geschirr!

- Was tun Sie?

Das Ende einer WG-Party!

- Sie laufen an der Küche vorbei und jemand gibt Ihnen den Auftrag:

  Spülen Sie das Geschirr!

- Was tun Sie?
- Idee!
  - Sie spülen ein einziges Teil
  - Dann suchen Sie die nächste Person, um ihr/ihm zu sagen, dass sie den Abwasch erledigen soll.

Das Ende einer WG-Party!

- Sie laufen an der Küche vorbei und jemand gibt Ihnen den Auftrag:

  Spülen Sie das Geschirr!

- Was tun Sie?
- Idee!
  - Sie spülen ein einziges Teil
  - Dann suchen Sie die nächste Person, um ihr/ihm zu sagen, dass sie den Abwasch erledigen soll.
- Die nächste Person macht es genauso: Sie spült ein Teil und bittet den/die Nächste ...
- Die letzte Person findet ein leeres Spülbecken vor.

Rekursive Algorithmen

Eine Prozedur, die einen Teil der Aufgabe selbst löst und dann den Rest erledigt, indem sie sich selbst aufruft, wird **rekursive Prozedur** genannt.

Beispiel: Geschirrspülen

- Falls das Spülbecken leer ist, ist nichts mehr zu tun.
- Andernfalls:
  - Spüle ein Teil;
  - Finde die nächste Person und sage ihr "Erledige den Abwasch".

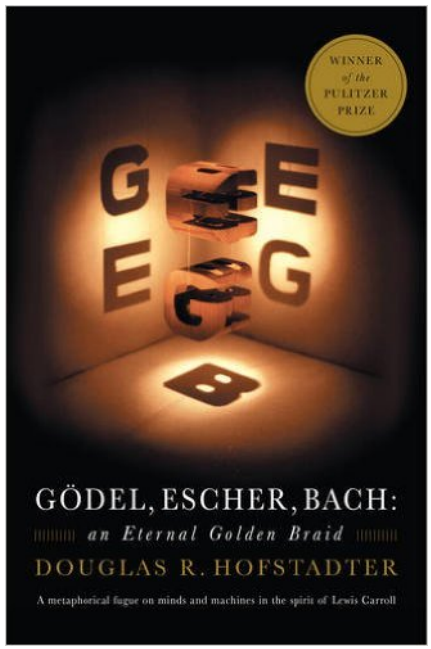
Beispiel: Sortieren eines Papierstapels

- Falls der Stapel aus einem Blatt Papier besteht, so ist er schon sortiert.
- Andernfalls:
  - Teile den Stapel in zwei kleinere Stapel;
  - Sortiere die kleineren Stapel;
  - Füge die beiden sortierten Stapel wieder zu einem Stapel zusammen.

Warum funktioniert der Trick?

- Rekursion ist ein einfaches und mächtiges Prinzip, mit dem viele Probleme gehandhabt werden können.
- Die **Problemgröße** im aktuellen Aufruf kann durch einen diskreten Wert n beschrieben werden (z.B. Anzahl der Geschirrteile, Größe des Stapels).
- Die Größe des Teilproblems im rekursiven Aufruf **muss kleiner sein** als n.
- Die Zerlegung in Teilprobleme **bricht irgendwann ab** (z.B. wenn das Spülbecken leer ist, wenn der Stapel nur aus einem Blatt Papier besteht).

Weihnachtswunschzettel!



Douglas Hofstadter: Gödel, Escher, Bach - Ein Endloses Geflochtenes Band. Verlag Klett-Cotta.

Begriffsklärung: Rekursion

- Eine Methodendeklaration m heißt **direkt rekursiv**, wenn der Methodenrumpf einen Aufruf von m enthält.
- Eine Menge von Methodendeklarationen heißen **verschränkt rekursiv** oder **indirekt rekursiv** (engl. *mutually recursive*), wenn die Deklarationen gegenseitig voneinander abhängen.
- Eine Methodendeklaration heißt **rekursiv**, wenn sie direkt rekursiv ist oder Element einer Menge verschränkt rekursiver Methoden ist.

Beispiel: Verschränkte Rekursion

```
public static boolean istGerade(int n) {
  if (n == 0) {
    return true;
  } else {
    return istUngerade(n-1);
  }
}

public static boolean istUngerade(int n){
  if (n == 0) {
    return false;
  } else {
    return istGerade(n-1);
  }
}
```

Beispiel: Fakultätsfunktion

$$n! = \begin{cases} n * (n-1)! & \text{falls } n > 0 \\ 1 & \text{falls } n = 0 \end{cases}$$

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

- **Basisfall** liefert Wert ohne nachfolgenden rekursiven Aufruf.
- **Rekursionsschritt** verwendet Ergebnis von rekursiven Methodenaufrufen für (andere) Parameterwerte für die Berechnung im aktuellen Methodenaufruf.

Eine rekursive Methodendeklaration m heißt **linear rekursiv**, wenn in jedem Ausführungszweig höchstens ein Aufruf von m auftritt.

Beispiel: Fibonacci-Folge

Jede Fibonacci-Zahl ist die Summe ihrer beiden Vorgänger:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 ...

$$fib(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ fib(n-1) + fib(n-2) & \text{für } n \geq 2 \end{cases}$$

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

Eine rekursive Methodendeklaration m, die nicht linear rekursiv ist, heißt **kaskadenartig rekursiv**.

Repetitive Rekursion

Eine linear rekursive Methodendeklaration m heißt **repetitiv rekursiv** (auch endrekursiv, engl. *tail recursive*), wenn jeder Aufruf von m in m der letzte auszuwertende Ausdruck ist.

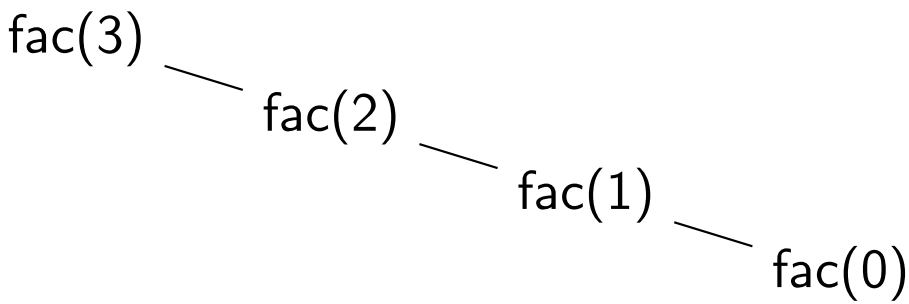
```
// groesster gemeinsamer Teiler von m und n
public static int ggT(int m, int n) {
  if (m == n) {
    return m;
  } else if (m > n) {
    return ggT(m-n, n);
  } else {
    return ggT(m, n-m);
  }
}
```

Umwandlung in repetitiv-rekursive Variante

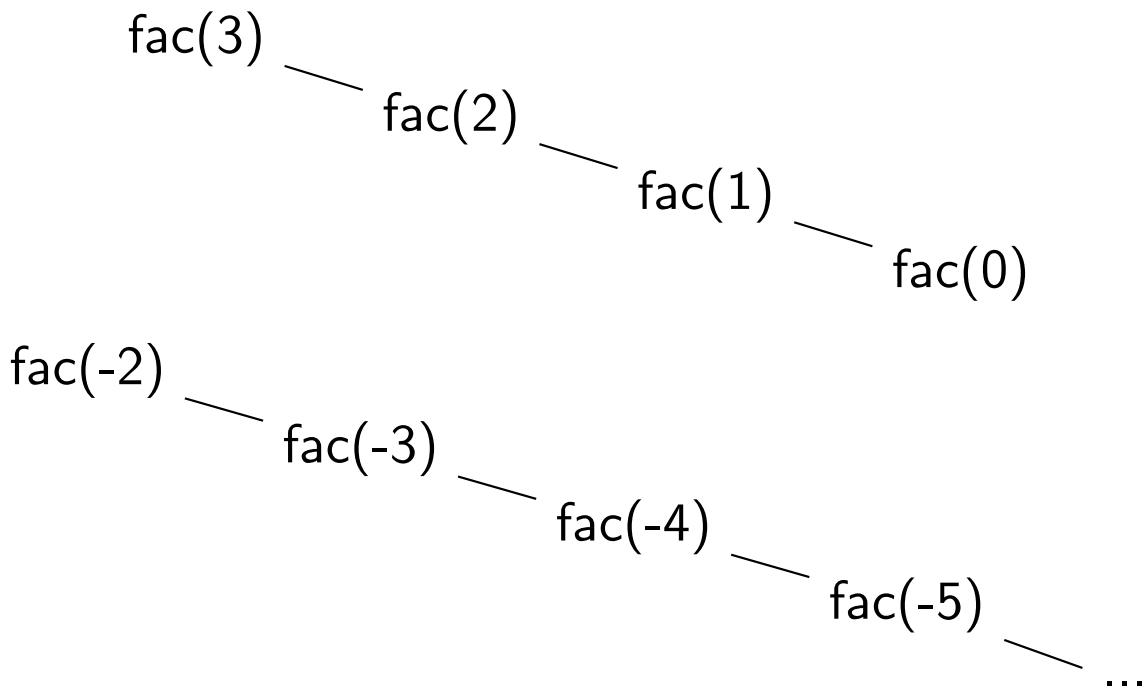
```
public static int fac(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fac (n-1);
    }
}


// repetitiv-rekursive Variante
public static int fac(int n) {
    return fac(n, 1);
}
public static int fac(int n, int x) {
    if (n == 0) {
        return x;
    } else {
        return fac(n-1, n * x);
    }
}
```

Zur Terminierung von rekursiven Funktionen



Zur Terminierung von rekursiven Funktionen



Terminierungsbeweise I

Idee: Messe den Abstand zum Basisfall!
Sei `f` eine rekursive Prozedur, die durch eine Implementierung folgender Art gegeben ist:

```
public static t f(t_1 x_1, ..., t_k x_k) {
  ... f(e_1, ..., e_k) ...
}
```

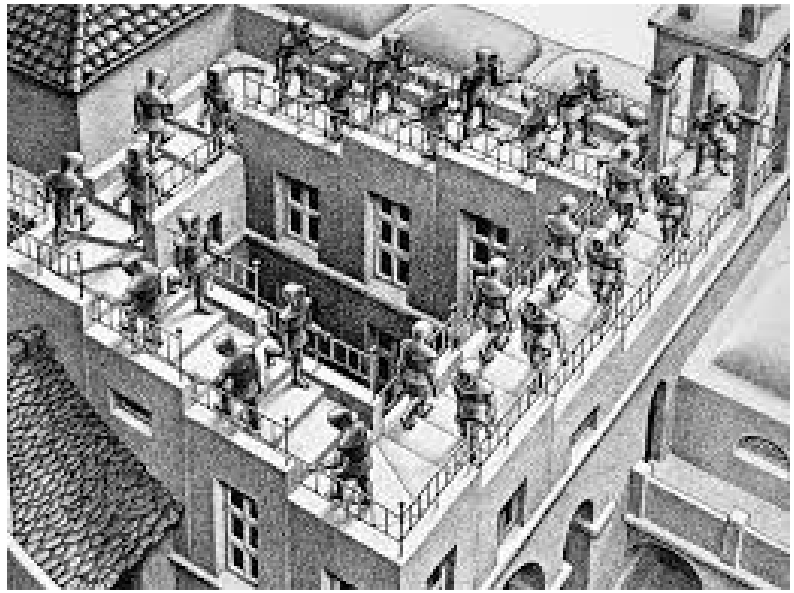
Schema:

1. Man definiert mit einer Vorbedingung, welche Parameter erlaubt sind. Für diese gültigen Parameter soll die Prozedur terminieren.
2. Wir zeigen, dass der gültige Parameterbereich nicht verlassen wird. Das heißt, für jeden rekursiven Aufruf begründen wir, warum die neuen Parameter die Vorbedingung erfüllen.

Terminierungsbeweise II

3. Wir definieren eine **Abstiegsfunktion** h, welche den Abstand der Parameterwerte zu einem Basisfall misst. Dazu muss h jeder gültigen Kombination von Parameterwerten eine natürliche Zahl ($\mathbb{N}_0$) zuweisen. D.h. für die Prozedur `f` oben ist h eine Funktion $h : t_1 \times \cdots \times t_k \to \mathbb{N}_0$.
4. Wir begründen, warum der Wert von h bei jedem rekursiven Aufruf für die neuen Parameterwerte echt kleiner ist als für die Parameterwerte des aktuellen Aufrufs, also:

$$h(x_1, \ldots, x_k) > h(e_1, \ldots, e_k)$$



- Da in $\mathbb{N}_0$ jede echt absteigende Kette endlich ist, ist eine unendliche Folge von rekursiven Aufrufen durch die Voraussetzungen ausgeschlossen.
- Wenn andere Ursachen zur Nichtterminierung (zum Beispiel Schleifen) ausgeschlossen werden können, dann ist damit die Terminierung der rekursiven Prozedur gezeigt.

Beispiel: Fakultät

```
1       public static int fac(int n) {
2         if (n == 0) {
3           return 1;
4         } else {
5           return n * fac (n-1);
6         }
7       }
```

1. Die Prozedur `fac` terminiert für positive Eingabewerte. Wie geben also die Vorbedingung `requires n >= 0` an.
2. Im rekursiven Aufruf in Zeile 5 wissen wir wegen der `if`-Anweisung, dass `n != 0` und daher gilt `n > 0`. Also ist `n-1 >= 0` und die Vorbedingung gilt.
3. Wir wählen $h(n) = n$ als Abstiegsfunktion. Da `n >= 0` für gültige Parameter gilt, bildet h die Parameter nach $\mathbb{N}_0$ ab.
4. Für den rekursiven Aufruf in Zeile 5 ist zu zeigen: $h(n) > h(n-1)$
   Dies gilt offensichtlich: $h(n) = n > n - 1 = h(n-1)$

Beispiel: ggT I

```
1     // groesster gemeinsamer Teiler von m und n
2     public static int ggT(int m, int n) {
3       if (m == n) {
4         return m;
5       } else if (m > n) {
6         return ggT(m-n, n);
7       } else {
8         return ggT(m, n-m);
9       }
10    }
```

1. Vorbedingung: `requires m > 0 && n > 0`
2. Vorbedingung gilt für die rekursiven Aufrufen:
   1. Aufruf in Zeile 5: `m > n`, also ist `m-n > 0`.
   2. Aufruf in Zeile 7: Es gilt weder `m == n`, noch `m > n`.
      Also gilt `n > m` und damit ist `n-m > 0`.

Beispiel: ggT II

3. Abstiegsfunktion: $h(m,n) = m + n > 0$ für $m, n > 0$
4. Die Abstiegsfunktion wird für rekursive Aufrufe echt kleiner:
   1. Aufruf in Zeile 5:
      $h(m,n) = m + n \overset{\text{weil } n>0}{>} m = m - n + n = h(m-n, n)$
   2. Aufruf in Zeile 7:
      $h(m,n) = m + n \overset{\text{weil } m>0}{>} n = m + n - m = h(m, n-m)$

Verallgemeinerung der Abstiegsfunktion

- Bisher: Abstiegsfunktion bildet nach $\mathbb{N}_0$ ab
- Neben den natürlichen Zahlen mit der Ordnung $<$ gibt es noch andere (partiell) geordnete Mengen, in denen es keine unendlichen absteigenden Ketten gibt.

Definition: Ordnung I

Eine Teilmenge R von $M \times N$ heißt eine (binäre) **Relation**.

Gilt $M = N$, dann nennt man R **homogen**.

Eine homogene Relation heißt:

- **reflexiv**, wenn für alle $x \in M$ gilt: $(x, x) \in R$
- **antisymmetrisch**, wenn für alle $x, y \in M$ gilt:
  wenn $(x, y) \in R$ und $(y, x) \in R$, dann $x = y$
- **transitiv**, wenn für alle $x, y, z \in M$ gilt:
  wenn $(x, y) \in R$ und $(y, z) \in R$, dann $(x, z) \in R$

Eine reflexive, antisymmetrische und transitive homogene Relation auf $M \times M$ heißt eine **partielle Ordnungsrelation**.

Definition: Ordnung II

Eine Menge M mit einer Ordnungsrelation R heißt eine **partielle Ordnung**.

Meist benutzt man Infixoperatoren wie $\leq$ (oder $\subseteq$) zur Darstellung der Relation und schreibt

$$x \leq y \quad \text{statt} \quad (x, y) \in R$$

und

$$x < y \quad \text{statt} \quad x \leq y \ \text{ und } \ x \neq y$$

Definition: Kette

Sei $(M, \leq)$ eine partielle Ordnung.

Eine Folge $x_0, x_1, x_2, \ldots$ heißt eine **absteigende** Kette, wenn überall $x_i \geq x_{i+1}$ gilt.

Falls sogar überall $x_i > x_{i+1}$ gilt, dann nennt man die Kette **echt absteigend**.

Eine **noethersche Ordnung** ist eine partielle Ordnung, die keine unendlichen echt absteigenden Ketten enthält.

Beispiel: Lexikographische Ordnung auf Zahlenpaaren

$$(x_1, y_1) \leq_{lex} (x_2, y_2)$$

genau dann, wenn

$$x_1 < x_2 \quad \text{oder} \quad (x_1 = x_2 \text{ und } y_1 \leq y_2)$$

- Vergleiche zuerst die erste Komponente; bei Gleichheit der ersten Komponente entscheidet die zweite Komponente über die Ordnung der Zahlenpaare.
- Beispiel: $(1, 42) \leq_{lex} (2, 0)$ und $(3, 7) \leq_{lex} (3, 9)$, aber $(2, 1) \not\leq_{lex} (1, 100)$.
- Die lexikographische Ordnung auf $\mathbb{N}_0 \times \mathbb{N}_0$ ist noethersch.

Beispiel: Ackermann-Funktion I

```
1  static long ackermann(long m, long n) {
2    if (m == 0) {
3      return n + 1;
4    } else if (n == 0) {
5      return ackermann(m - 1, 1);
6    }
7    return ackermann(m - 1, ackermann(m, n - 1));
8  }
```

- In jedem rekursiven Aufruf wird entweder der erste Parameter kleiner oder der erste Parameter bleibt unverändert und der zweite wird kleiner.

Beispiel: Ackermann-Funktion II

1. Vorbedingung: `requires m >= 0 && n >= 0`
2. Vorbedingung für rekursive Aufrufe ist erfüllt.
   - Aufruf in Zeile 5: $m - 1 \geq 0$, weil $m \neq 0$ und $m \geq 0$ gilt.
   - Äußerer Aufruf in Zeile 7: $m - 1 \geq 0$; außerdem ist der Rückgabewert der `ackermann`-Funktion immer mindestens 1.
   - Innerer Aufruf in Zeile 7: $n - 1 \geq 0$, weil $n \neq 0$ nach if-Bedingung.
3. Abstiegsfunktion: $h(m, n) = (m, n)$
   mit noethersche Ordnung $(\mathbb{N} \times \mathbb{N}, \leq_{lex})$.
4. Für die rekursiven Aufrufe gilt dann:
   1. Aufruf in Zeile 5:
      $h(m - 1, 1) = (m - 1, 1) <_{lex} (m, 0) = (m, n) = h(m, n)$
   2. Äußerer Aufruf in Zeile 7:
      $h(m - 1, a(m, n - 1)) = (m - 1, a(m, n - 1)) <_{lex} (m, n) = h(m, n)$
   3. Innerer Aufruf in Zeile 7:
      $h(m, n - 1) = (m, n - 1) <_{lex} (m, n) = h(m, n)$

Rekursion vs. Iteration

Für jede Methode, die ein Problem mit Hilfe von Rekursion löst, kann eine Alternativimplementierung gefunden werden, die das gleiche Problem iterativ, d.h. mit Schleifenkonstrukten löst.

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

// iterative Variante
public static int fac(int n) {
    int result = 1;
    int i = n;
    while (i != 0) {
     result = i * result;
     i--;
    }
    return result;
}
```

Warum Rekursion?

- Oft einfacherer und eleganterer Code, wenn man Probleme auf rekursiven Datenstrukturen löst
- Manche Programmiersprachen (z.B. Haskell, Ocaml, Erlang etc.) bieten keine Sprachmittel für Schleifen an.
- Programmierer müssen daher beide Varianten verstehen und anwenden können!
- Terminierungsbeweise sind bei der Entwicklung von Qualitätssoftware sehr wichtig, und zwar unabhängig vom verwendeten Modellierungs- bzw. Programmierparadigma.
- Es sollte zur Routine der Softwareentwicklung gehören, den zulässigen Parameterbereich festzulegen und dafür Terminierung zu zeigen.