TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Lernziele

- Rekursive Prozeduren zu charakterisieren.
- Terminierung von rekursiven Prozeduren mit Hilfe von geeigneten Abstiegsfunktionen zu beweisen.
- Lexikographische Ordnungen auf Zahlenpaaren anzuwenden.
- Iterative und rekursive Varianten eines Algorithmus anzugeben und zu vergleichen.

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".

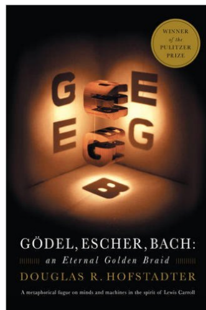
Beispiel: Sortieren eines Papierstapels

- 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);
  }
}
```

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

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**.

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

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.

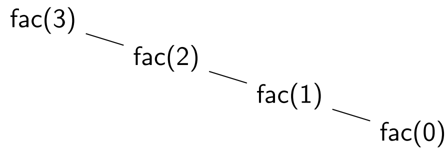
```
// groesster gemeinsamer Teiler von m und n
public static int ggT(int m, int n) {
  if (m == n) {
    return m;
  } else if (m > n) {
    return ggT(m-n, n);
  } else {
    return ggT(m, n-m);
  }
}
```

Umwandlung in repetitiv-rekursive Variante

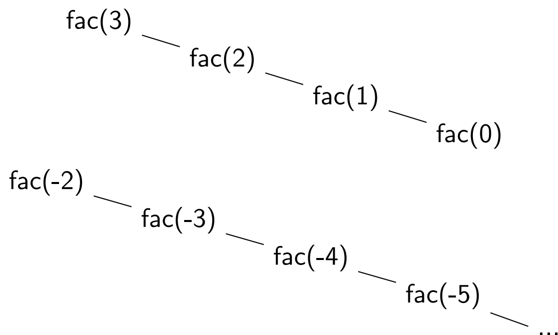
```
public static int fac(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fac (n-1);
    }
}


// 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);
    }
}
```

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

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) ...
}
```

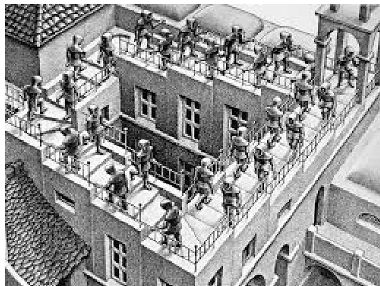
Schema:

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:
      $h(m, n) = m + n \overset{\textit{weil } n>0}{>} m = m - n + n = h(m - n, n)$
   2. Aufruf in Zeile 7:
      $h(m, n) = m + n \overset{\textit{weil } m>0}{>} n = m + n - m = h(m, n - m)$

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

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

- 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.