

# Weitere Beispiele

Haskell:

```
fibrep 0 last res = 0
fibrep 1 last res = res
fibrep n last res = fibrep (n-1) res (last+res)
fib n = fibrep n 0 1
```

Java:

```
int fib(int n) {
    int last = 0, res = 1;
    for(int i=1;i<n;i++) {
        int next = last+res;
        last = res;
        res = next;
    }
    return res;
}
```

## Weitere Beispiele (2)

Java:

```
public class Fib {
    static int fib(int n) {
        int last = 0;
        int res = 1;
        for(int i=1;i<n;i++) {
            int next = last+res;
            last = res;
            res = next;
        }
        return res;
    }
    public static void main(String[] args) {
        System.out.println("Result = "+fib(20));
    }
}
```

## Weitere Beispiele (3)

### Haskell

```
ggT 0 n = n
ggT m n = ggT (n `mod` m) m
```

### Java

```
int ggT(int m,int n) {
    while(m!=0) {
        int next = n % m;
        n = m;
        m = next;
    }
    return n;
}
```

## Weitere Beispiele (4)

<b>Prozedural</b>	<b>Funktional</b>
Wertezuweisung	best. Wertebindungen
Schleifen	best. Rekursion
Ein-/Ausgabe	(Monaden)
globale Variable	(Monaden)

## Weitere Beispiele (5)

Wichtige Programmtransformationen:

- rekursive Prozeduren zu *iterativen* Programmen
- d.h. Rekursion zu Schleifen
- iterative Programme oft effizienter
- optimierende Programmtransformation
- kann automatisiert werden
- Programmtransformation analog zu Umformung math. Ausdrücke

# Längeres Transformationsbeispiel

## Haskell

```
collatzStep n =  
  if n<=1 then 1 else  
  if odd n then 3*n+1 else  
  n `div` 2  
collatz n = position 1 (iterate collatzStep n)
```

## Java

```
int collatz(int n) {  
  int steps = 0;  
  while(n>1) {  
    steps += 1;  
    if(n%2==1)  
      n = 3*n+1;  
    else  
      n = n/2;  
  }  
  return steps;  
}
```

# Längeres Transformationsbeispiel (2)

## Haskell

```
collatz_seq = map collatz [0..]
collatz_max = scanl max 0 collatz_seq
high_watermarks = map (1+) (findIndices (uncurry (<)) (zip collatz_max
    (tail collatz_max)))
```

## Java

```
int [] high_watermarks(int n) {
    int [] result = new int[n];
    int i = 0, high = 0, k = 1;
    while(i < n) {
        int next = collatz(k);
        if(next > high) {
            high = next;
            result[i++] = k;
        }
        k++;
    }
    return result;
}
```

## Unterabschnitt 4.0.4

# Programmablauf

# Argumentübergabe

Haskell:

```
ite b y n = if b then y else n
fac n = ite (n==0) 1 (n*fac(n-1))
```

Java:

```
int ite(boolean b,int y,int n) {
    if(b) return y;
    else return n;
}
int fac(int n) {
    return ite(n==0,1,n*fac(n-1));
}
```

## Argumentübergabe (2)

Java:

```
int f(int x) {  
    print(x);  
    return x;  
}  
int g(int x,int y,int z) {  
    print(99);  
    return x+y;  
}  
void h() {  
    print(g(f(7),f(11),f(13)));  
}
```

## Argumentübergabe (3)

Haskell:

```
import System.IO.Unsafe (unsafePerformIO)
debug x y = unsafePerformIO (do print x; return y)
```

```
f x = debug x x
g x y z = debug 99 (f x + f y)
h = g 1 2 3
main = print h
```

# Argumentübergabe (4)

## Haskell vs Java Funktionsargumente

- Haskell Funktionsaufruf (call-by-need, lazy evaluation, non-strict evaluation)
  - ▶ nimm die Argumente und wandle sie in ein Versprechen (think, promise) um
  - ▶ ruf die Funktion mit dem Thunk auf
  - ▶ nur wenn der Wert gebraucht wird, evaluiere den Thunk
  - ▶ nach der Evaluierung, speichere den Wert
  - ▶ der Thunk wird im Lexical Scope des ursprünglichen Ausdrucks evaluiert
- Java Funktionsaufruf (call-by-value, eager evaluation, strict evaluation)
  - ▶ für jedes Argument, evaluiere den entsprechenden Ausdruck
  - ▶ rufe die Funktion mit den berechneten Werten als Argument auf
  - ▶ Evaluierung der Argumente in Java ist links-nach-rechts
  - ▶ eigentlich: call-by-sharing wegen Objektreferenzen

# Argumentübergabe (5)

## Konsequenzen

- Java Argumentübergabe ist schneller/einfacher
- In Haskell, `if...then...else`, `&&`, `||` können als Funktionen definiert werden
- In Java, `?...:....`, `&&` und `||` sind sog. *Special Forms*

# Ausführung eines Programmes

Beispiel: Abzahlung in Java im Debugger

Beachte:

- Schritte
- Variablenzustand
- Steuerzustand

# Ausführung eines Programmes (2)

## Haskell

```
abzahlung zinsen zahlung rest =  
  if rest<=zahlung then 1  
  else 1 + abzahlung zinsen zahlung ((rest-zahlung)*(1.0+zinsen/100.0))
```

## Java

```
int abzahlung(double zinsen, double zahlung, double rest) {  
  int monate = 1;  
  while(rest>zahlung) {  
    print("monat "+monate+" restbetrag "+rest);  
    rest -= zahlung;  
    rest *= (1.0 + zinsen/100.0);  
    monate += 1;  
  }  
  return monate;  
}
```

## Begriffsklärung: (Zustände)

Jeder Schritt bei der Ausführung eines Algorithmus führt von einem **Ausführungszustand** zum **Nachfolgezustand**. Ein Ausführungszustand ist gekennzeichnet durch

- den **Speicherzustand** (im Wesentlichen der Zustand der Variablen);
- den **Steuerungszustand** (vereinfacht gesagt, die Stelle im Programm, an der die Ausführung angekommen ist).

Ein Ausführungsschritt führt zu einer Zustandsveränderung, also einer Veränderung von Speicher- und/oder Steuerungszustand.

# Begriffsklärung: (Aktion)

In einem Ausführungsschritt wird üblicherweise eine **Aktion** ausgeführt. Aktionen sind:

- Zuweisungen an Variablen
- Kommunikation mit der Umgebung (Ein- und Ausgabe)?

Die Aktion bestimmt nachfolgende Steuerungszustände bzw. die Terminierung des Algorithmus.

# Begriffsklärung: (Ablauf)

Der **Ablauf** eines Algorithmus zu gegebenen Eingaben wird charakterisiert durch

- die Sequenz der Ausführungszustände,
- die Sequenz der ausgeführten Aktionen.

# Algorithmus

- Algorithmen sind Verallgemeinerungen von solchen prozeduralen Programmen.
- unabhängig von der Beschreibungstechnik / Programmiersprache
- analoge Schritte und Zustände in allen Beschreibungen
- formale oder informelle Beschreibung

# Begriffsklärung: (Effizienz)

- Effizienz = geringer Aufwand
- Aufwand: Anzahl der Schritte oder verbrauchter Speicher
- Aufwand in Abhängigkeit von Eingabe daten

## Begriffsklärung: (Effizienz) (2)

Mit den gemachten Präzisierungen lässt sich der Aufwand einer Ausführung quantifizieren:

- Zeitkomplexität: Wie viele Schritte braucht der Algorithmus in Abhängigkeit von den Eingabewerten?
- Raumkomplexität: Wie viel Speicherplatz braucht der Algorithmus in Abhängigkeit von den Eingabewerten?

# Debugging

- Verständnis der Zustände, Abfolge, Anzahl von Schritten sehr wichtig
- Ansätze
  - ▶ interaktives Debugging
  - ▶ Tracing
  - ▶ Profiling
- Methoden
  - ▶ spezielle Software
  - ▶ automatisches Instrumentieren (tracing, profiling)
  - ▶ manuelles Instrumentieren (print, Zähler)

# Debugging (2)

Beispiele:

- Instrumentierung von Fib, FibLoop mit Zählern
- manuelles Tracing von Fib, FibLoop mit print

# Debugging (3)

- Wichtig: Verstehen der Schritte und Zustände von Programmen
- Arbeiten mit *legacy code*, Bibliotheken
- interaktives vs. nicht-interaktives Debugging
- persistentes vs. nicht-persistentes Debugging (assert, “-d”)
- Debugging, Tracing, Profiling, Assertions auswählen/anwenden