

## Kapitel 4

### Übersicht (2)

- Einführung in die Algorithmenanalyse
- Algorithmenklassen & -entwicklung
- Verifikation prozeduraler Programme**
- Spezifikation von Prozedureigenschaften
- Verifikation von Prozeduren

### Übersicht

- 4. Prozedurales Programmieren
- Java Programmierung
- Programmablauf
- Speicher und Datenstrukturen
- Speicherorganisation und CPU
- Boxen und Pfeile
- Variablen, Datenstrukturen, Speicher
- Prozeduraufrufe
- Listenalgorithmen in Java
- Destruktive Bäume
- Manuelle Speicherverwaltung
- Felder als Listen
- Felder im Speicher
- Garbage Collection
- Effizienz
- Algorithmen in prozeduraler Formulierung

### Abschnitt 4.1

## Java Programmierung

# Java

Position Nov 2010	Position Nov 2009	Delta in Position	Programming Language	Ratinge Nov 2010	Delta Nov 2009	Status
1	1		Java	18.509%	+0.14%	A
2	2	==	C	16.717%	-0.80%	A
3	4	↑	C++	9.497%	-0.50%	A
4	3	↓	PHP	7.813%	-2.38%	A
5	6	↑	C#	5.708%	+0.36%	A
6	7	↑	Python	5.678%	+1.01%	A
7	5	↓↓	(Visual) Basic	5.479%	-2.70%	A
8	13	↑↑↑↑	Objective-C	3.191%	+2.30%	A
9	8	↓	Perl	2.472%	-1.02%	A
10	10		Ruby	1.907%	-0.50%	A
11	9	↓↓	JavaScript	1.684%	-1.25%	A
12	11	↓	Delphi	1.638%	-0.49%	A
13	17	↑↑↑↑	Lisp	1.097%	+0.47%	A
14	23	↑↑↑↑↑	Transact-SQL	0.793%	+0.38%	A
15	15	==	Pascal	0.784%	+0.13%	A
16	29	↑↑↑↑↑↑	Ada	0.695%	+0.38%	B
17	36	↑↑↑↑↑↑↑	NXT-G	0.682%	+0.45%	B
18	14	↓↓↓	SAS	0.669%	-0.15%	B
19	30	↑↑↑↑↑	RPG (OS/400)	0.658%	+0.37%	B
20	12	↓↓↓	PL/SQL	0.655%	-0.25%	B

Position	Programming Language	Ratinge
21	MatLab	0.636%
22	Lua	0.612%
23	ADP	0.597%
24	Object Pascal	0.586%
25	Go	0.548%
26	Scheme	0.508%
27	Fortran	0.477%
28	IS	0.422%
29	COBOL	0.404%
30	Logo	0.392%
31	CL (OS/400)	0.371%
32	APL	0.366%
33	R	0.365%
34	JavaPKScript	0.365%
35	R	0.365%
36	JScript.NET	0.350%
37	C Shell	0.327%
38	ActionScript	0.326%
39	Scratch	0.325%
40	IDL	0.325%
41	Visual Basic .NET	0.325%
42	Smalltalk	0.312%
43	Allegro	0.311%
44	Prolog	0.300%
45	Erlang	0.297%
46	Smalltalk	0.286%
47	Fortran	0.285%
48	AWK	0.285%
49	Perl	0.279%
50	Scala	0.232%

# Funktional / Prozedural

## Funktionale Programmierung

- Gleichungen
- Ergebnisse
- mathematische Variablen
- Evaluierungsreihenfolge weniger wichtig

## Prozedurale Programmierung

- Anweisungen
  - Schritte
  - Zustandsveränderungen von Variablen
  - Abfolge der Anweisungen sehr wichtig
- Algorithmen sind eher prozedural (Anweisungen, Schritte, ...)

# Haskell vs Java

Haskell:

```
fib x =
  if x==0 then 0 else
  if x==1 then 1 else
  fib (x-1) + fib (x-2)
```

Java:

```
int fib(int x) {
  if(x==0) return 0; else
  if(x==1) return 1; else
  return fib(x-1)+fib(x-2);
}
```

Java erlaubt ebenfalls Funktionen und Rekursion.

# Haskell vs Java (2)

Haskell:

```
fib x =
  if x==0 then 0 else
  if x==1 then 1 else
  fib (x-1) + fib (x-2)
```

Java:

```
int fib(int x) {
  return x==0 ? 0 :
  x==1 ? 1 :
  fib(x-1)+fib(x-2);
}
```

Ähnliche Konstrukte bedeuten manchmal etwas anderes.

## Haskell vs Java (3)

Haskell	Java
Funktion	Funktion
rekursiver Aufruf	rekursiver Aufruf
if ... then ... else	... ? ... : ...
n/a	return
n/a	if ... then ... else

## Haskell vs Java (4)

- Haskell hat Ausdrücke (expressions), die Werte liefern
- Java hat Ausdrücke (expressions), die Werte liefern UND Anweisungen (statements), die Befehle ausführen

## Prozedurale Programmierung in Java

Konstrukte	Beispiele
Blöcke	{ x; y; z; ... }
Variablendeklaration	int x; int y = 17;
Wertezuweisung	x = 3;
Prozeduren	void p(int x,int y) { ... }
Funktionen	p(3,4); int f(int x,int y) { ... return 42; }
Bedingungen	int x = f(); if(...) ... if(...) ... else ... switch(...) { case ....: ...; break; }
Schleifen	while(...) ... do ... while(...) for(...;...;...) ... break;

## Prozedurale Programmierung in Java (2)

Javaprogramme enthalten Sequenzen von Deklarationen und Anweisungen:

```
// initialisierte Variablendeklaration
int x = 17;
print(x);
// Wertezuweisung
x = 42;
print(x);
```

- Deklarationen nehmen die Form "typ name;" oder "typ name = initialer\_wert".
- Typen von Variablen müssen explizit deklariert werden
- Zuweisungen haben die Form "name = neuer\_wert;".
- Zuweisungen müssen den korrekten Typ haben

## Prozedurale Programmierung in Java (3)

Ausdrücke sind ähnlich wie in Haskell

```
5 + 89
45 / 6
47474747L * a
23+3
d == 78 + e
7 >= 45 == true
abs(a) + abs(d)
sein | !sein
true || tuEs()
a > b ? a : b
```

©2010

TU Kaiserslautern

526

## Prozedurale Programmierung in Java (4)

Sichtbarkeit von Variablen (Scope): Variablen sind nur sichtbar in den Blöcken, wo sie deklariert sind.

```
int x = 17;
print(x);
{
    int y = 42;
    print(y);
}
print(y); // FEHLER
```

©2010

TU Kaiserslautern

527

## Prozedurale Programmierung in Java (5)

Variable Shadowing: Variablen in inneren Blöcken überdecken Variablen in äußeren Blöcken.

```
int x = 17;
print(x);
{
    int x = 42;
    print(x);
}
print(x);
```

©2010

TU Kaiserslautern

528

## Prozedurale Programmierung in Java (6)

Funktionen

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

- Funktionen werden als “type name(argumente) { ... }” definiert.
- Funktionswerte müssen mit “return” zurückgegeben werden.
- Der Körper einer Funktion muss ein Block sein.

©2010

TU Kaiserslautern

529

## Prozedurale Programmierung in Java (7)

### Prozeduren

```
void print_bottles(int n) {
    print(""+n+" bottles of beer on the wall");
}
```

- Prozeduren sind wie Funktionen, werden aber mit dem speziellen Typ "void" deklariert.
- Prozeduren werden für Seiteneffekte ausgeführt: Ein-/Ausgabe, Wertezuweisungen

©2010

TU Kaiserslautern

530

## Prozedurale Programmierung in Java (8)

### Seiteneffekte und Purity

```
int f() {
    return 99;
}
int g(int x) {
    return x+1;
}
```

*f* und *g* sind "reine" Funktionen ("pure"): jeder Aufruf mit demselben Argument liefert dasselbe Ergebnis.

©2010

TU Kaiserslautern

531

## Prozedurale Programmierung in Java (9)

```
int x;
int g() {
    return x;
}
void h(int y) {
    x = y;
}
```

- *g* bezieht sich auf eine Variable in einem äußeren Scope
- *h* ist eine Prozedur mit einer Wertezuweisung als Seiteneffekt
- *g* liefert unterschiedliche Werte, je nach dem womit *h* aufgerufen wird
- funktionale Programmierung verwendet reine Funktionen

©2010

TU Kaiserslautern

532

## Prozedurale Programmierung in Java (10)

```
int x;
void f() {
    int y;
    x = y;
}
```

- *x* ist eine *globale Variable*
- *y* ist eine *lokale Variable*

©2010

TU Kaiserslautern

533

## Prozedurale Programmierung in Java (11)

### Lexical vs Dynamic Scoping

```
void f() {
    print(x);
}
void g() {
    int x = 3;
    f();
}
```

- Das obige Programm ist ein Fehler in Java: die Variable  $x$  ist nicht in  $f$  sichtbar. Dies heißt "lexical scoping".
- In manchen anderen Sprachen ist die Variable  $x$  in  $g$  im Funktionsaufruf  $f$  sichtbar. Dies heißt "dynamic scoping".

©2010

TU Kaiserslautern

534

## Prozedurale Programmierung in Java (12)

### Bedingte Anweisungen

```
if(n==1)
    print("1 bottle of beer on the wall");
else
    print(""+n+" bottles of beer on the wall");
```

- if...then...else ist eine Anweisung, kein Ausdruck
- jeder Zweig enthält eine Anweisung
- wenn mehrere Anweisungen benötigt werden, benutze "{...}"

```
if(n==1) {
    print("1 bottle of beer on the wall");
} else {
    print(""+n+" bottles of beer on the wall");
}
```

©2010

TU Kaiserslautern

535

## Prozedurale Programmierung in Java (13)

### Bedingte Anweisungen

```
switch(n) {
    case 0: print("zero"); break;
    case 1: print("one"); break;
    case 2: print("two"); break;
    default: print("many"); break;
}
```

©2010

TU Kaiserslautern

536

## Prozedurale Programmierung in Java (14)

### while-Schleifen

```
int i=99;
while(i>0) {
    print(""+i+" bottles of beer on the wall");
    i = i-1;
}
```

### do-while-Schleifen

```
int i=99;
do {
    print(""+i+" bottles of beer on the wall");
    i = i-1;
} while(i>0);
```

©2010

TU Kaiserslautern

537

## Prozedurale Programmierung in Java (15)

```

for-Statement
for(int i=99;i>0;i=i-1) {
    print(""+i+" bottles of beer on the wall");
}

break-Statement
int i=99;
for(;;) {
    print(""+i+" bottles of beer on the wall");
    i = i -1;
    if(i==0) break;
}

```

©2010

TU Kaiserslautern

538

## Prozedurale Programmierung in Java (16)

Java hat einige Ausdrücke mit Seiteneffekten.

```

x = i++;
x = ++i;
x = i--;
x = --i;
x = (i+=1);
x = (i=3);

```

©2010

TU Kaiserslautern

539

## Java Datentypen

- einfache Datentypen (Werte): int, float, double, char
- einfache Datentypen: String
- zusammengesetzte Datentypen: Felder (arrays)
- benutzerdefinierte Datentypen / Verbundtypen: class

©2010

TU Kaiserslautern

540

## Java Datentypen (2)

Felder werden ähnlich wie Listen in Haskell gebraucht:

```

int[] x = new int[1000];
String[] s = new String[];
int[] squares = {0,1,4,9,16,25,49,64};

```

©2010

TU Kaiserslautern

541

## Java Datentypen (3)

Listen in Haskell werden i.A. mit Rekursion oder Funktionen, wie `foldl`, bearbeitet.  
 Felder in Java werden i.A. mit Schleifen bearbeitet (Index fängt mit 0 an):

```
void sum(int[] a) {
    int total = 0;
    for(int i=0; i<a.length; i++) {
        total = total + a[i];
    }
    return total;
}
```

©2010

TU Kaiserslautern

542

## Java Datentypen (4)

```
void fill(int[] a, int value) {
    for(int i=0; i<a.length; i++) {
        a[i] = value;
    }
}
```

Beachte:

- Referenz als Argument übergeben, Feld wird in der Prozedur modifiziert
- Feldelemente sind wie "nummerierte Variablen": `a[0]`, `a[1]`, ...
- Feldelemente werden von 0 aufsteigend bis `a.length` nummeriert
- links vom "=" kann nicht nur eine Variable stehen, sondern auch ein Feldelement
- Ausdrücke, die links vom "=" stehen können, bezeichnen wir als "l-value"

©2010

TU Kaiserslautern

543

## Java Datentypen (5)

null

```
String a = "hello";
int[] l = {1, 2, 3};
int x = 99;

a = null;
l = null;
x = null; // FALSCH
```

Der Wert "null" ist auch der Initialwert für alle Variablen von einem Typ an den "null" zugewiesen werden kann.

```
int[] l;
if(l==null) print("OK");
```

©2010

TU Kaiserslautern

544

## Java Datentypen (6)

Haskell:

```
data IntList = IL Int IntList | NULL
x = IL 3 NULL
```

Java:

```
public class IntList {
    int head;
    IntList tail;
};
IntList x = new IntList();
x.head = 3;
x.tail = null;
```

- nur ein Konstruktor, plus "null"
- Konstruktor = "new" Typname
- Initialisierung der Werte durch Zuweisung
- Selektor = `.name`

©2010

TU Kaiserslautern

545



## Java Datentypen (7)

Haskell:

```
data IntList = IL Int IntList | NULL
head IL x _ = x
tail IL _ t = t
sum l = if l==NULL then 0 else head l + sum (tail l)
```

Java: (ungewöhnlich)

```
public class IntList {
    int head;
    IntList tail;
};
int sum(IntList l) {
    if(l==null) return 0;
    return l.head + sum(l.tail);
}
```

©2010

TU Kaiserslautern

546

## Objektreferenzen

Java unterscheidet zwischen Werten und Objektreferenzen. Diese

Unterscheidung existiert auch in Haskell, kann aber nicht beobachtet werden. In Java können wir dies durch Wertezuweisung beobachten.

©2010

TU Kaiserslautern

547

## Objektreferenzen (2)

Vergleiche...

```
int x = 3;
int y = x;
y = 17;
print(x);
```

Mit...

```
int[] x = {3};
int[] y = x;
y[0] = 17;
print x[0];
```

©2010

TU Kaiserslautern

548

## Objektreferenzen (3)

Zahlen werden als Werte übergeben.

Felder und Verbundtypen werden als Referenzen übergeben:

- die Referenz selbst kann nicht verändert werden
- das Object, auf das die Referenz verweist, kann verändert werden

Zeichenketten werden ebenfalls als Referenzen übergeben, können aber nicht verändert werden. Daher sind sie von Werten nicht zu unterscheiden.

©2010

TU Kaiserslautern

549

## Sonderregeln (Java)

Prozedurale Programmierung in Java ist etwas komplizierter; benutzen Sie dieses Muster:

```
public class MeinProgramm {
    static int x = 3;
    static void p() {
        System.out.println("Hello, World (" + x + ")");
    }
    public static void main(String[] args) {
        p();
    }
}

$ javac MeinProgramm.java
$ java MeinProgramm
```

©2010

TU Kaiserslautern

550

## Sonderregeln (Java) (2)

Für Ein-/Ausgabe verwenden Sie...

```
public class IO {
    public static int readInt(){...}
    public static String readString(){...}
    public static char readChar(){...}
    public static void print(Object o){...}
    public static void println(Object o){...}
}

public class MeinProgramm {
    public static void main(String[] args) {
        IO.println("Hello, World!");
    }
}
```

©2010

TU Kaiserslautern

551

## Erinnerung

- Java / Haskell Analogien: Funktionen, Rekursion, Verbundtypen
- analoge Rekursionsformen
- analoge statische Typendeklaration
- neue Konstrukte in Java: Zustand, Schleifen
- neue Datentypen: Felder (Liste ist einfach Verbundtyp)

©2010

TU Kaiserslautern

552

## Erinnerung (2)

- Ausdrücke (insb. Bitarithmetik)
- Variablen (Sichtbarkeit/Scope, Verschattung/Shadowing)
- Seiteneffekte (reine Funktionen, globale Variablen, Programmzustand)
- Lexical Scoping vs. Dynamic Scoping
- Werte und Referenztypen, null

©2010

TU Kaiserslautern

553

## Äquivalenz

Java:

```
int f(int x) {
  int i=10;
  while(i>0) {
    x = x+2;
    i = i-1;
  }
  return x
}
```

Haskell:

```
f x = f_loop 10 x
f_loop 0 x = x
f_loop i x = f_loop (i-1) (x+2)
```

©2010

TU Kaiserslautern

554

## 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;
}
```

©2010

TU Kaiserslautern

555

## 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));
  }
}
```

©2010

TU Kaiserslautern

556

## 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;
}
```

©2010

TU Kaiserslautern

557

## Weitere Beispiele (4)

Prozedural	Funktional
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 4.1.1

### Programmablauf

©2010

TU Kaiserslautern

562

## 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));
}
```

©2010

TU Kaiserslautern

563

## 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)));
}
```

©2010

TU Kaiserslautern

564

## 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
```

©2010

TU Kaiserslautern

565

## 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 (thunk, 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

©2010

TU Kaiserslautern

566

## 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*

©2010

TU Kaiserslautern

567

## Ausführung eines Programmes

Beispiel: Abzahlung in Java im Debugger

Beachte:

- Schritte
- Variablenzustand
- Steuerzustand

©2010

TU Kaiserslautern

568

## 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;
}
```

©2010

TU Kaiserslautern

569

## 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

©2010

TU Kaiserslautern

574

## 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?

©2010

TU Kaiserslautern

575

## 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)

©2010

TU Kaiserslautern

576

## Debugging (2)

Beispiele:

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

©2010

TU Kaiserslautern

577



## 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

©2010

TU Kaiserslautern

578

## Seiteneffekte und Fehler

(Auch: Illustration von Feldern, Feldoperationen und Bits)

```
// 111, 100 und 000: mittleres Element auf 0, sonst 1
void step110(int[] line) {
    for(int i=1; i<line.length-1; i++) {
        int code = 0;
        if(line[i-1]==1) code += 4;
        if(line[i]==1) code += 2;
        if(line[i+1]==1) code += 1;
        if(code==7 || code==4 || code==0)
            line[i] = 0; // ???
        else
            line[i] = 1;
    }
}
```

(Auch: Binärcode)

©2010

TU Kaiserslautern

579

## Seiteneffekte und Fehler (2)

```
void step110(int[] line) {
    int[] result = new int[line.length];
    for(int i=1; i<line.length-1; i++) {
        int code = 0;
        if(line[i-1]==1) code += 4;
        if(line[i]==1) code += 2;
        if(line[i+1]==1) code += 1;
        if(code==7 || code==4 || code==0)
            result[i] = 0;
        else
            result[i] = 1;
    }
    for(int i=0; i<line.length; i++) {
        line[i] = result[i];
    }
}
```

©2010

TU Kaiserslautern

580

## Seiteneffekte und Fehler (3)

Fehler in step110 ist Beispiel für Probleme von Seiteneffekten: wir haben unerwartet den Kontext geändert, in dem die Funktion läuft. Wir

vermeiden das in imperativen Sprachen durch Kopieren. In Haskell wird die Kopie automatisch durchgeführt. In Java können wir etwas effizienteres machen:

©2010

TU Kaiserslautern

581

## Seiteneffekte und Fehler (4)

```
// ohne Hilfsfelder, ohne Kopieren
void step110(int[] line) {
    int last, last2;
    for(int i=1; i<line.length-1; i++) {
        int code = 0;
        last2 = last;
        if(line[i-1]==1) code += 4;
        if(line[i]==1) code += 2;
        if(line[i+1]==1) code += 1;
        if(code==7 || code==4 || code==0)
            last = 0;
        else
            last = 1;
        if(i>1) line[i-1] = last2;
    }
    line[line.length-2] = last2;
}
```

©2010

TU Kaiserslautern

582

## Seiteneffekte und Fehler (5)

- Was ist intuitiver: funktional or prozedural?
- Warum ist die Frage wichtig?
- Helfen math. Eigenschaften, wie *referential transparency*?
- Zustände in der Welt: Objekte, Personen, ...

©2010

TU Kaiserslautern

583

## Determinismus

Die meisten prozeduralen Konstrukte sind *deterministisch*: gleiche Daten + gleiches Program → gleiche Ausgabe + gleiche Zustände  
Dies ist nicht unbedingt notwendig.

©2010

TU Kaiserslautern

584

## Determinismus (2)

- nicht-deterministische Ausgabe / nicht-determinierter Algorithmus: andere Ergebnisse
- nicht-deterministischer Algorithmus: andere Zustandsfolge
- nicht-deterministische Programme können (müssen aber nicht) unterschiedliche Ergebnisse liefern

©2010

TU Kaiserslautern

585

## Determinismus (3)

Bedeutung:

- wähle Zufallszahlen im Algorithmus: randomisierte Algorithmen
- Parallelität → nicht-deterministisches Verhalten
- nicht-deterministische Algorithmen → Parallelisierung

## Determinismus (4)

```
int divide(int x, int y) {  
    for(;;) {  
        int z = choose_between(1, x+1);  
        if(x==z*y) return z;  
    }  
}
```

Randomisierte Algorithmen:

- Las Vegas – immer richtig aber manchmal langsam
- Monte Carlo – schnell aber manchmal falsch  
(Welcher ist dies? Warum x+1?)