Informatik II

Felix Friedrich

Vorlesung am D-BAUG der ETH Zürich

FS 2014

0. ORGANISATORISCHES

W i l l k o m m e n

zur Vorlesung Informatik II !

am D-BAUG der ETH Zürich.

Ort und Zeit:

Montag 12:45– 14:30, HPH G2.

Pause 13:30 – 13:40, leichte Verschiebung möglich

Vorlesungshomepage

http://informatik2.baug.ethz.ch

Team

Dozent	Felix Friedrich
Chefassistent	Lars Widmer
Assistenten	Renzo Roth Nico Previtali Robin Guldener Urs Müller Oskar Triebe Severin Wischmann Raffaele Lauro Fabian Schewetofski Chahat Bhatia Sandro Huber Temmy Bounedjar Simon Lösing Fabian Stutz Lei Zhong

Übungen

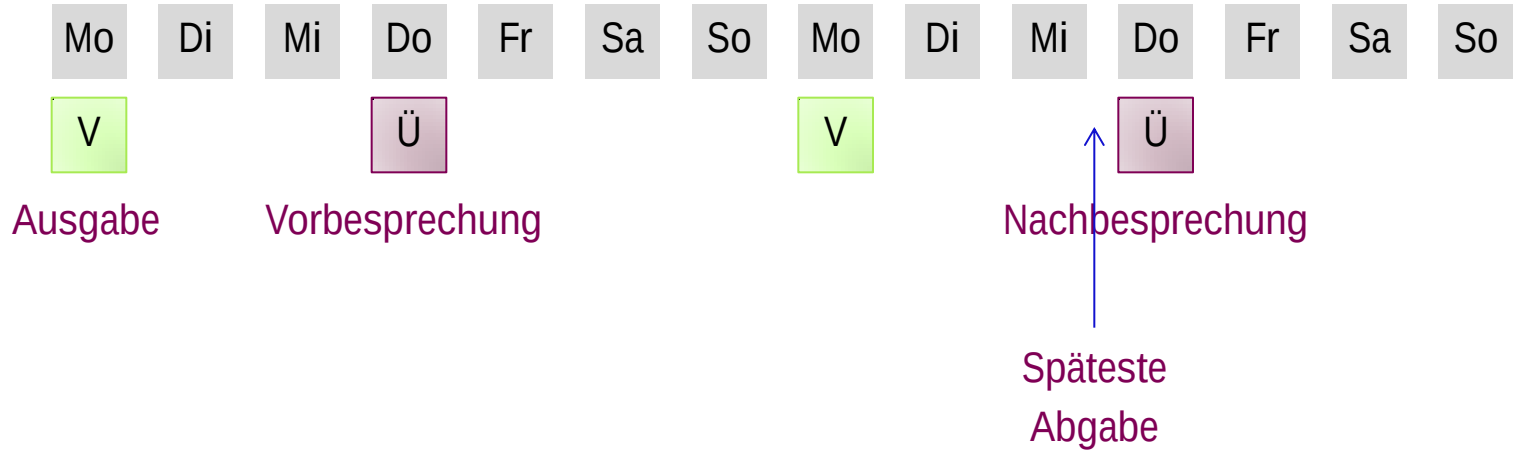
Donnerstag	12:00 – 13:00	Oskar Triebe	HCI J 8	
	13:00 – 14:00	Renzo Roth	HCI D 4	
		Nico Previtali	HCI D 6	
		Robin Guldener	HCI F 2	
		Urs Müller	HCI F 8	
		Severin Wischmann	HIT F 13	
		Raffaele Lauro	HIT F 31.1	
		Fabian Stutz	HIT F 31.2	
		Chahat Bhatia	HIT J 53	englisch
	14:00 – 15:00	Sandro Huber	HIT F 13	
		Temmy Bounedjar	HIT F 31.1	
		Simon Lösing	HIT F 31.2	
	15:00 – 16:00	Fabian Schewetofski	HCI D 6	
		Lei Zhong	HCI F 2	englisch

Einschreiben in die Übungsgruppen

- Gruppeneinteilung selbstständig via Webseite
- E-Mail mit Link nur nach Belegung dieser Vorlesung in myStudies
- Genau eine E-Mail mit personalisiertem Link
- Link behalten !
- Die gezeigten Räume und Termine sind abhängig vom Studiengang.
- Weitere Einschränkungen bei der Auswahl.

Ablauf

- Übungsblattausgabe zur Vorlesung.
- Vorbesprechung der Übung am Donnerstag darauf
- Abgabe der Übung spätestens eine Woche später (Mittwoch 23:59). Abgabe per email an den Assistenten
- Nachbesprechung der Übung am Donnerstag darauf. Korrektur der Abgaben innerhalb einer Woche nach Nachbesprechung.



Zu den Übungen

- Ab HS 2013 für Prüfungszulassung kein Testat mehr erforderlich.
- Bearbeitung der wöchentlichen Übungsserien ist freiwillig, wird aber dringend empfohlen!

Tipps

- Üben Sie!
- Wenn Sie zu zweit an Übungen arbeiten, stellen Sie Gleichverteilung des aktiven Parts sicher.
- Lassen Sie sich nicht frustrieren. Schlafen Sie eine Nacht, wenn Sie nicht vorwärtskommen.
- Holen Sie sich Hilfe, wenn Sie nicht vorwärtskommen.
- Üben Sie!

Computerarbeitsplätze

- In den Computerräumen des D BAUG ist die nötige Software (Java + Eclipse) installiert, genauso auch im Hauptgebäude im Zentrum.
- Falls Sie auf Ihrem eigenen Rechner arbeiten möchten: Installationsanweisung für Java / Eclipse auf dem ersten Übungsblatt
- Bringen Sie Ihren Computer doch bitte zur ersten Übungsstunde mit.

Literatur

- Robert Sedgewick, Kevin Wayne, Einführung in die Programmierung mit Java. Pearson, 2011
  - Englisch: Robert Sedgewick, Kevin Wayne, Introduction to Programming in Java: An Interdisciplinary Approach, Addison-Wesley, 2008
- Christian Ullenboo, Java ist auch eine Insel, http://openbook.galileocomputing.de/javainsel/
- Guido Krüger, Heiko Hansen, Handbuch der Java-Programmierung Standard Edition Version 7, Addison-Wesley, 2011, http://www.javabuch.de
- Thomas Ottmann, Peter Widmayer, Algorithmen und Datenstrukturen, Springer
  pdf derzeit auch als download bei Springer

Relevantes für die Prüfung*

Prüfungsstoff für die Endprüfung schliesst ein

- Vorlesungsinhalt und
- Übungsinhalte.

Folien, die mit einem Stern (*) markiert sind, sind nicht unmittelbar prüfungsrelevant.

Prüfung ist (Hand-)schriftlich. Zugelassene Hilfsmittel: selbstverfasste Zusammenfassung auf maximal 8 A4-Seiten (respektive 4 Blatt doppelseitig)

Übung lösen können ≈ Prüfung lösen können

Prüfung Vorgehen

Prüfung gemeinsam mit Informatik I

Informatik I	2 Stunden	Basisprüfung Inf I + II
Pause	30 Minuten	
Informatik II	2 Stunden	Basisprüfung Inf I + II + Semesterkurs Inf II

In Ihrem und unserem Interesse ...

- Bitte melden sie *frühzeitig*, wenn Sie Probleme sehen, wenn Ihnen
  - die Vorlesung zu schnell, zu langsam, zu schwierig, zu einfach, zu .... ist
  - die Übungen nicht machbar sind ...
  - Sie sich nicht gut betreut fühlen ...
- kurz: wenn Ihnen irgendetwas auf dem Herzen liegt

Überblick, die Turing-Maschine, von Pascal zu Java, Rekursion, Exceptions, Terminierung und Korrektheit, das Halteproblem, Fallstudie Altägyptische Multiplikation, Endrekursion und Iteration, Invarianten, Effizienz

1. EINFÜHRUNG

Ziel der Vorlesung

Hauptziel: Problemlösen mit dem Computer
auch als Vorbereitung für Ihr weiteres Studium

- Hilfsmittel
  - Objektorientierte Programmiersprache (Java)
  - Tools wie Matlab und Datenbanken
- Methodik
  - *Kernthemen:* Objektorientiertes Programmieren, Datenstrukturen, Algorithmen und Komplexität, Datenbanken
  - *Fallstudien:* Interessante Probleme aus der Informatik und angrenzenden Gebieten

Warum Java?

- Sehr weit verbreitete moderne Sprache, funktioniert auf sehr vielen Systemen
  - Portabilität durch Zwischensprache
- Verbietet einige typische Fehler
- Gute Datenbankanbindung

Ziel der heutigen Vorlesung

- Überblick über die Vorlesung, Einführung
- Wiederholung / Auffrischung einiger Begrifflichkeiten
  - Algorithmus, Rekursion, Endrekursion und Iteration, Korrektheit / Terminierung, Invarianten, Effizienz
- Pascal → Java
  Prozedurales Programmieren mit Java

Schauen wir uns zuerst noch einmal die universelle Maschine an, mit der wir es zu tun haben.

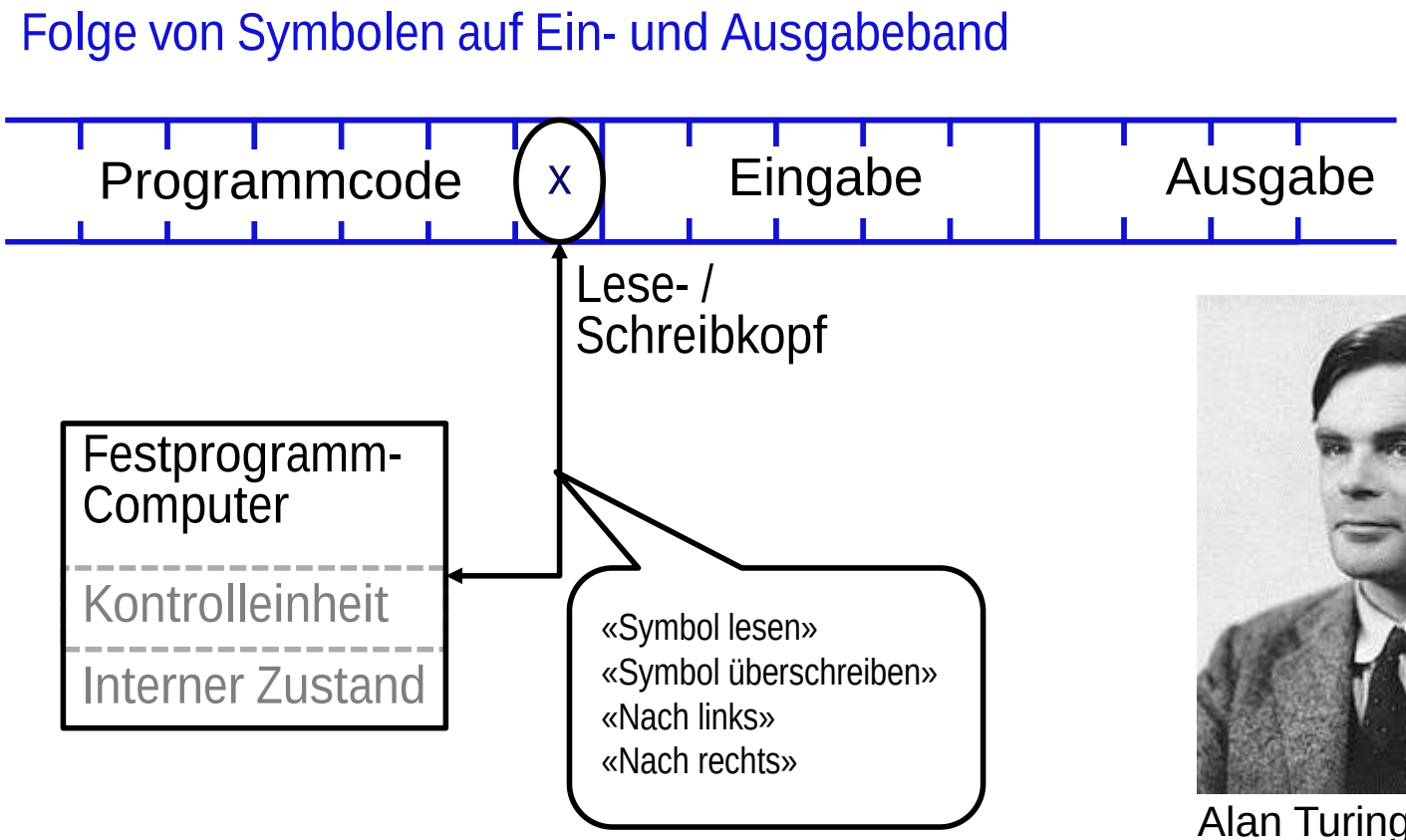
Programmieren

- Mit Hilfe einer *Programmiersprache* wird dem Computer eine Folge von Verarbeitungs-anweisungen erteilt, damit er genau das macht, was wir wollen.
- Die Folge von Verarbeitungsanweisungen ist das *Computerprogramm*.

http://en.wikipedia.org/wiki/Harvard_Computers
Harvard computers, ca. 1890
Menschliche Berufsrechner

Der universelle Computer*

Eine geniale Idee:
Universelle Turing Maschine (Alan Turing, 1936)



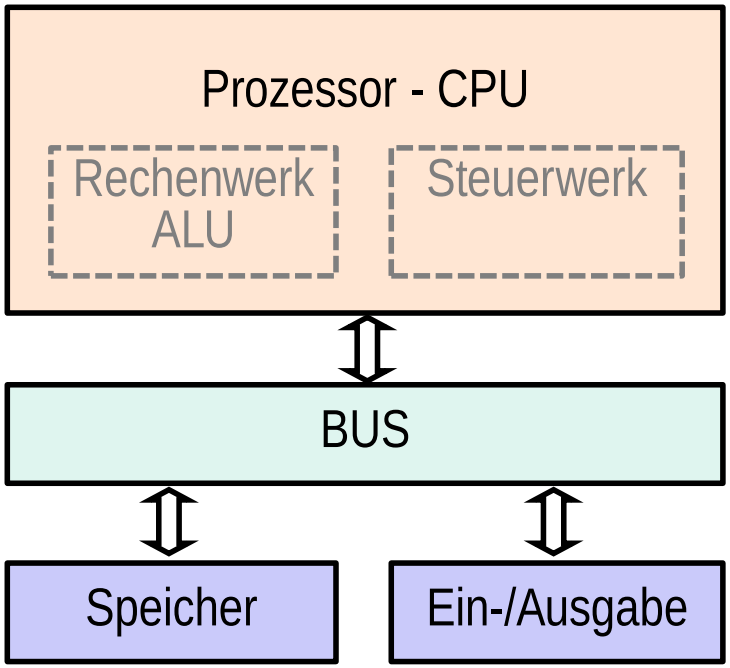
http://en.wikipedia.org/wiki/Alan_Turing

Alan Turing

Realisierungen*

Z1 – Konrad Zuse (1938)
ENIAC – Von Neumann (1945)

Von Neumann Architektur



/http://www.hs.uni-hamburg.de/DE/GNT/hh/biogr/zuse.htm

Konrad Zuse

http://commons.wikimedia.org/wiki/File:John_von_Neumann.jpg

John von Neumann

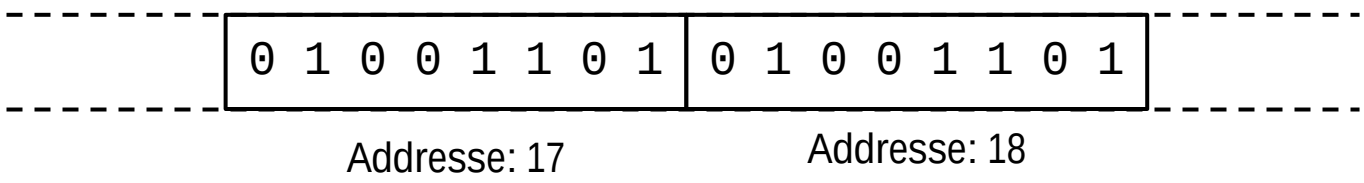
Erinnerung: Universelles Rechenmodell

Zutaten der *Von Neumann Architektur*

- Hauptspeicher (RAM) für Programme *und* Daten
- Prozessor (CPU) zur Verarbeitung der Programme und Daten
- I/O Komponenten zur Kommunikation mit der Aussenwelt

Erinnerung: Hauptspeicher

- Folge von Bits aus {0,1}.
- Programmzustand: Werte aller Bits.
- Zusammenfassung von Bits zu Speicherzellen (oft: 8 Bits = 1 Byte).
- Jede Speicherzelle hat eine Adresse.
- Random Access: Zugriffszeit auf Speicherzelle (nahezu) unabhängig von ihrer Adresse.



Erinnerung: Prozessor

Der Prozessor

- führt Programminstruktionen in Maschinensprache aus,
- hat eigenen "schnellen" Speicher (Register), darunter auch den Programmzähler (PC),
- kann vom Hauptspeicher lesen und in ihn schreiben und
- beherrscht eine Menge einfachster Operationen (z.B. Addieren zweier Registerinhalte).

Geschwindigkeit

In der mittleren Zeit, die der Schall von mir zu Ihnen unterwegs ist...

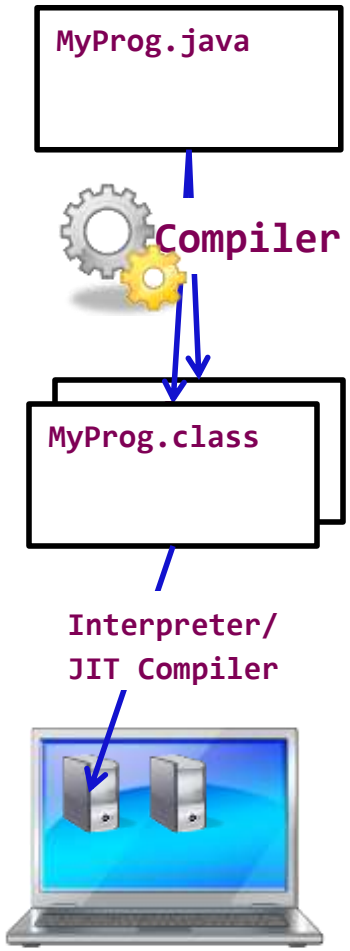
30 m

≙ mehr als 100.000.000 Instruktionen

arbeitet ein heutiger Desktop-PC mehr als 100 Millionen Instruktionen ab.

Java

- Basiert auf einer *virtuellen Maschine*, (mit Von-Neumann Architektur)
  - Programmcode wird in Zwischencode übersetzt,
  - Zwischencode läuft in einer "simulierten" Rechnerumgebung. Einfachste Form: Interpretation des Zwischencodes durch einen sog. Interpreter
  - Optimierung: JIT (Just-In-Time) Kompilation von häufig benutztem Code. Partielle Transition
    virtuelle Maschine → physikalische Maschine
- Folgerung und erklärtes Ziel von Java
  - Portabilität (Write Once Run Anywhere)



Java (Inf II) vs. Pascal (Inf I)

Andere Sprachfamilien → *Syntaktische* Unterschiede

	Pascal	Java
Gross-/Kleinschreibung	nicht *case-sensitive*	*case sensitive*
Deklarationsanordnung	`i,j,k: INTEGER;`	`int i,j,k;`
Blöcke	`BEGIN` *statements* `END;`	`{` statements `}`
Zuweisungen	`i := i + j`	`i = i + j`
Vergleich	`i = j` `i <> j`	`i == j` `i != j`
Logische Operatoren	`AND, OR, NOT`	`&&, ||, !`
Integer-Operatoren	`MOD, DIV`	`%, /`

Java (Inf II) vs. Pascal (Inf I)

	Pascal	Java

Bedingte Ausführung	`IF` *expression* `THEN`
	  statement
	`ELSE`
	  statement
		`if (`*expression*`)`
		  statement
		`else`
		  statement

For-Schleifen	`FOR i := 1 TO 10 DO`
	  statement
		`for (int i=1; i<= 10; i++)`
		  statement

While-Schleifen	`WHILE i<10 DO`
	  statement
		`while (i<10)`
		  statement

Repeat-Schleifen	`REPEAT` *statements* `UNTIL i = 10`	`do` *statement* `while (i != 10)`
Rückgabe	*functionName* `:= x`	`return x;`
Konstanten	`CONSTANT max = 100`	`final int MAX = 100;`
Arrays	`X: ARRAY [0..9] OF INTEGER;`	`int[] A = new int[10];` oder `int A[] = new int[10];`

Java (Inf II) vs. Pascal (Inf I)

- Funktionen / Prozeduren

```
FUNCTION f(x: INTEGER): INTEGER
BEGIN
   ...
   f := 47
END;

PROCEDURE p(x: INTEGER)
BEGIN
   ...
END;
```

- Grundtypen

```
BYTE, WORD, CARDINAL, QWORD

SHORTINT, SMALLINT (INTEGER),
LONGINT, INT64

SINGLE (REAL), DOUBLE

CHAR

BOOLEAN
```

- (Statische) Methoden

```
static int f(int x){

   ...
   return 47;

};

static void p(int x)
{
   ...
}
```

- Grundtypen

```
-- (keine vorzeichenlosen Typen)

byte, short,
int, long

float, double

char

boolean
```

Java (Inf II) vs. Pascal (Inf I)

- Datensätze (Records)

```
R = RECORD
   a: INTEGER;
   b: BOOLEAN;
END
```

- Records beinhalten nur Daten.
- Prozeduren können auf Records via Referenzparameter (VAR) operieren.
  Erinnerung: Pass-by-Reference vs. Pass-by-Value

- Klassen

```
class R {
   int a;
   boolean b;
}
```

- Klassen beinhalten Daten und Methoden zum Operieren auf den Daten.
- Es gibt keine Referenz-parameter. Klassen sind hinter den Kulissen jedoch als Zeiger realisiert.
  Später mehr dazu.

Wo kann ich das nachlesen?

- Verschiedene Bücher (z.B. die eingangs angegebenen)
- Tutorials, z.B. http://docs.oracle.com/javase/tutorial/
- Java Language Specification http://docs.oracle.com/javase/specs/jls/se7/html/
- Pascal vs. Java http://www.cs.cmu.edu/~cburch/pgss97/javapascal.html
- Vergleich Pascal / C / Java http://www.cs.cornell.edu/Courses/cs409/2000SP/Java/java-compare.html

Links auf der Homepage der Vorlesung

Erstes Java Programm

```
public class Hello {

  public static void main(String[] args)
  {
    System.out.println("Hello World.");
  }
}
```

Programme sind als Klassen organisiert.

Diese spezielle *Methode* signalisiert, dass diese Klasse wie ein Programm in Pascal aufgerufen werden kann

"die Methode println in der Variablen *out* in der Klasse *System*"

Aufruf: `java Hello`

Klassen

- Ein Java-Programm besteht aus mindestens einer *Klasse*.
- Klassen *kapseln* Daten und Code. Wir gehen darauf später noch genauer ein.
- Ein Java-Programm, das eigenständig lauffähig ist, hat darüber hinaus eine Klasse mit `main`-Funktion. Diese spielt die Rolle des Programmrumpfes bei Pascal

```
public class Test{

        // potentiell weiterer Code und Daten

        public static void main(String[] args) {
                ...
        }

}
```

- Eine *Instanz* einer Klasse heisst *Objekt.* Auch das wird im Detail noch erklärt.

Prozedurale Programmierung mit Java

- Pascal → Java
  - Programm → *Klasse* mit public static void main
  - Prozedur → static *Methode* in dieser Klasse
  - Globale Variablen → static Variablen in dieser Klasse
  - Deklarationen, Ausdrücke und Anweisungen: s. Tabelle oben
  - Records → Klassen (Achtung: Referenzsemantik)
  - Arrays → Arrays (Achtung: Referenzsemantik)
  - Var-Parameter → *es gibt kein Pass-by-Reference! Die Referenzsemantik der Klassen erlaubt aber ein gemeinsames Operieren auf denselben Daten.*

Algorithmus: Altägyptische Multiplikation

- Beispiel: $11 \cdot 9 = 9 \cdot 11$

11	9
~~22~~	~~4~~
~~44~~	~~2~~
88	1
99	

9	11
18	5
~~36~~	~~2~~
72	1
99	

- Verdoppeln (linke Spalte), ganzzahliges Halbieren (rechte Spalte)
- Zeilen streichen, bei denen rechts eine gerade Zahl steht
- Übrige Zahlen der linken Spalte aufaddieren

Vorteile der altägyptischen Multiplikationsmethode

- Kurze Beschreibung
- Einfach in der Anwendung
- Effizient für Computer im Dualsystem
  - Verdoppeln: "left shift"

$$9 = 01001_b \rightarrow 10010_b = 18$$

  - Halbieren: "right shift"

$$9 = 01001_b \rightarrow 00100_b = 4$$

  - Algorithmus wird daher in der ALU (arithmetic logic unit) einiger CPUs verwendet

Fragen

- Funktioniert das immer?
  - z.B. für negative Zahlen?
  - Wenn nicht, wann?
- Warum funktioniert es für alle natürlichen Zahlen?
  - Wie beweist man die Korrektheit?
- Besser als die "Schulmethode"?
  - Was heisst "gut"?
  - Lässt sich Güte linear anordnen?
- Wie schreibt man das Verfahren unmissverständlich auf?
  - Notation / Sprache ?

Zur Erinnerung: Verhalten eines Programmes

Zur Compilationszeit:

- vom Compiler akzeptiertes Programm (syntaktisch und partiell semantisch korrekt)
- Compiler Fehler

Zur Laufzeit:

- korrektes Resultat
- inkorrektes Resultat
- Programmabsturz
- Programm terminiert nicht

Terminierung: Halteproblem*

Unentscheidbarkeit des Halteproblems*

Es gibt kein Programm, das für jedes Programm *P* und jede Eingabe *I* korrekt feststellen kann, ob das Programm *P* bei Eingabe von I terminiert.

Das heisst, die Korrektheit von Programmen kann *im Allgemeinen nicht* automatisch überprüft werden.

*Alan Turing, 1936. Theoretische Fragestellungen dieser Art waren für Alan Turing die Hauptmotivation für die Konstruktion seiner Rechenmaschine

Wie funktioniert der Algorithmus?

Beobachtung:

$a \cdot b = 2a \cdot \frac{b}{2}$, wenn b gerade

Somit

$$a \cdot b = \begin{cases} 2a \cdot \frac{b}{2} & \text{falls } b \text{ gerade} \\ a + 2a \cdot \frac{b-1}{2} & \text{falls } b \text{ ungerade} \end{cases}$$

Was nützt uns das?

- Wenn Verdoppeln und Halbieren triviale Operationen sind, dann kann Multiplikation zweier Zahlen auf ein einfacheres Problem zurückgeführt werden.

Terminierung

- Wir müssen noch festlegen, wann die Problemreduktion am Ende ist
  - Eine endlose Reduktion ist schliesslich nicht praktikabel
  - Wir hören auf, wenn das Problem trivial geworden ist
  - Hier: wenn $b = 1$ ist

Wir schreiben also: $a \cdot b = \begin{cases} a & \text{falls } b = 1 \\ 2a \cdot \frac{b}{2} & \text{falls } b \text{ gerade} \\ a + 2a \cdot \frac{b-1}{2} & \text{falls } b \text{ ungerade} \end{cases}$

- Terminierung klar, denn: b/2 und (b-1)/2 sind kleiner als b. Somit fällt b in jedem Rekursionsschritt
- Voraussetzung: b > 0 !

Multiplikationsalgorithms, rekursiv

Ein Multiplikationsalgorithmus lässt sich also wie folgt rekursiv definieren ($b > 0$)

$$f(a,b) = \begin{cases} a & \text{falls } b = 1 \\ f(2a, b/2) & \text{falls } b \text{ gerade} \\ a + f(2a, (b-1)/2) & sonst \end{cases}$$

In Java

```
static int f(int a, int b){
    if (b == 1) return a;
    else if (b%2 == 0) return f(2*a, b/2);
    else return a + f(2*a, (b-1)/2);
}
```

Das ganze Programm

```
class Mult {
   public static void main(String args[]) {
      int i = 5, j = 9;
      System.out.println(i + " x " + j + " = " + f(i, j));
   }

   // pre: a, b > 0
   // post: returns a*b
   static int f(int a, int b) {
      System.out.println(a + " " + b);
      if (b == 1)
         return a;
      if (b % 2 == 0)
         return f(a + a, b / 2);
      else
         return a + f(a + a, (b-1) / 2);
   }
}
```

Korrektheit

Wir wollen zeigen, dass die Funktion

$$f(a,b) = \begin{cases} a & \text{falls } b = 1 \\ f(2a, b/2) & \text{falls } b \text{ gerade} \\ a + f(2a, (b-1)/2) & sonst \end{cases}$$

für alle $a, b \in \mathbb{N}^+$ das Produkt von a und b berechnet.

Wir schränken bewusst den Definitionsbereich der Funktion ein (Preconditions ← Informatik I).

Damit sollte auch der Korrektheitsbeweis für die Java-Funktion geführt sein, oder?

- Woher wissen wir, dass (wann) das Programm der definierten Funktion genau entspricht?

Korrektheit: Beweis per Induktion

- Mathematischer Beweis: Induktion über $b \in \mathbb{N}^+$
- Induktionsanfang: $b = 1 \Rightarrow f(a,b) = a = a \cdot 1$ ✓
- Induktionsschritt: $b \to b+1 \; (b > 0, a > 0)$
  - Induktionsannahme: $f(a,b') = a \cdot b' \;\; \forall\, 0 < b' \leq b$
  - Zu zeigen: $f(a,b+1) = a \cdot (b+1)$

Wir rechnen nach

$$f(a,b+1) = \begin{cases} a + f\left(2a, \frac{b}{2}\right) = a + a \cdot b & b \; gerade \\ f\left(2a, \frac{b+1}{2}\right) = a \cdot (b+1) & b \; ungerade \end{cases}$$ ✓

Nach Induktionsannahme stimmt das, da immer $\frac{b+1}{2} \leq b$

Experimente

- Was passiert für negative Eingaben von b?
- Was passiert für b=0?

Ausgabe von f(1,0):

```
1 0
2 0
4 0
8 0
16 0
32 0
64 0
128 0
256 0
512 0
1024 0
....
```

```
...
8388608 0
16777216 0
33554432 0
67108864 0
134217728 0
268435456 0
536870912 0
1073741824 0
-2147483648 0
0 0
....
```

```
...
0 0
0 0
....
```

Exception in thread "main" java.lang.StackOverflowError

Stack-Überlauf

Arithmetischer Überlauf

Ausnahmen (Exceptions)

Ausnahmen sind Fehlerereignisse

- Werden oft vom System ausgelöst
- Können aber auch explizit im Programm ausgelöst werden
- Können abgefangen und behandelt werden

Strukturierung in Java durch "try" und "catch":

```
try {
     // Hier stehen Anweisungen, bei denen
     // eine Fehlerbedingung eintreten kann
}
catch (Fehlertyp1) {
     // "Behandlung" dieses Fehlertyps;
}
catch (Fehlertyp2) {
     // "Behandlung" dieses Fehlertyps;
}
```

Beispiel: Abfangen einer Exception

```
class MultTest {

     public static void main(String args[]) {
          for (int a = -5; a < 5; ++a)
               for (int b=-5; b<5; ++b)
                    test(a,b);
          System.out.println("done.");

     }
     static int f(int a, int b) { ... }

     static void test(int a, int b)
     {
          try
          {
               if (a*b !=  f(a,b))
                    System.out.println("error for a=" + a +", b=" + b);
          }
          catch(StackOverflowError e)
          {
               System.out.println("a stack overflow occured for a="+a+", b="+b);
          }
     }
}
```

Bemerkung

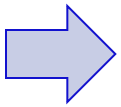
Programmierung mit Exceptions erlaubt Unterbrechung des normalen Kontrollflusses.

Der gezielte Einsatz mit benutzerdefinierten Exceptions ist manchmal praktisch, aber auch umstritten.

Iterative Version?

- Die Rekursion lässt sich *endrekursiv* schreiben, d.h. so dass nur ein rekursiver Aufruf der Funktion am Ende der Funktion stattfindet:

```
static int f(int a, int b) {
    if (b == 1)
        return a;
    if (b % 2 == 0)
        return f(2*a, b/2);
    return a + f(2*a, (b-1)/2);
}
```

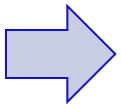


```
static int ft(int a, int b) {
    int z = 0;
    if (b == 1)
        return a;
    if (b % 2 != 0)
        {
            --b;
            z = a;
        }
    return z + ft(2*a, b / 2);
}
```

- Vorteil: endrekursive Funktionen lassen sich leicht iterativ schreiben

Endrekursion → Iteration

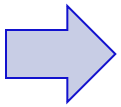
```
static int ft(int a, int b) {
    int z = 0;
    if (b == 1)
        return a;
    if (b % 2 != 0)
        {
            --b;
            z = a;
        }
    return z + ft(2*a, b / 2);
}
```



```
static int fi(int a, int b) {
    int res = 0;
    while (b != 1) {
        int z = 0;
        if (b % 2 != 0)
            {
                --b;
                z = a;
            }
        a *= 2;
        b /= 2;
        res += z;
    }
    res += a;
    return res;
}
```

Vereinfachen

```
static int fi(int a, int b) {
    int res = 0;
    while (b != 1) {
        int z = 0;
        if (b % 2 != 0)
            {
                --b;
                z = a;
            }
        a *= 2;
        b /= 2;
        res += z;
    }
    res += a;
    return res;
}
```



```
static int fi(int a, int b) {
    int res = 0;
    while (b > 0) {
        if (b % 2 != 0)
            {
                --b;
                res += a;
            }
        a *= 2;
        b /= 2;
    }
    return res;
}
```

Invarianten !

```
static int fi(int a, int b) {
    int res = 0;
    while (b > 0) {
        if (b % 2 != 0)
            {
                --b;
                res += a;
            }
        a *= 2;
        b /= 2;
    }
    return res;
}
```

Sei $x = a \cdot b$

Hier gilt offensichtlich $x = res + a \cdot b$

Wenn hier $x = res + a \cdot b$ gilt ...

... dann gilt es auch hier

Wenn hier $x = res + a \cdot b$ gilt ...

... dann gilt es auch hier

Hier gilt offensichtlich $b = 0$,
also auch $x = res + a \cdot b = res$

$\Rightarrow res + a \cdot b$ ist eine **Invariante** !

Invarianten als Beweistechnik

Der Ausdruck $res + a \cdot b$ ist eine sogenannte Invariante

Zwar ändern sich die Werte von a, b, res im Programm, aber $res + a \cdot b$ bleibt „im Wesentlichen“ unverändert:

- „Kurzzeitig“ wird die Invariante durch eine Anweisung zerstört, aber dann gleich darauf wieder repariert
- Betrachtet man solche Aktionsfolgen als „atomar“, bleibt der Wert tatsächlich invariant
- Insbesondere erhält die Schleife die Invariante („Schleifeninvariante“), sie wirkt dort wie der Induktionsschritt bei der vollständigen Induktion

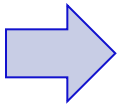
Damit ist klar: Die Funktion liefert das Produkt von a und b

Invarianten sind offenbar mächtige Beweishilfsmittel!

Weiteres Kürzen des Codes

- Mit etwas Wissen über die Integerarithmetik...

```
static int fi(int a, int b) {
   int res = 0;
   while (b > 0) {
      if (b % 2 != 0)
         {
            --b;
            res += a;
         }
      a *= 2;
      b /= 2;
   }
   return res;
}
```



```
static int fi(int a, int b) {
   int res = 0;
   while (b > 0) {
      res += a * (b%2);
      a *= 2;
      b /= 2;
   }
   return res;
}
```

Dieser Code enthält allerdings eine echte Multiplikation. Wir verwenden ihn im Folgenden für die Analyse des Algorithmus.

Schlussfolgerung

- Altägyptische Multiplikation entspricht der "Schulmethode", zur Basis 2!

```
static int fi(int a, int b) {
    int res = 0;
    while (b > 0) {
        res += a * (b%2);
        a *= 2;
        b /= 2;
    }
    return res;
}
```

			a		x					b	
1	0	0	1		x		1	0	1	1	
							1	0	0	1	(9)
						1	0	0	1		(18)
						1	1	0	1	1	
				1	0	0	1				(72)
				1	1	0	0	0	1	1	(99)

Effizienz des Multiplikationsalgorithmus

- Frage: Wie lange dauert eine Multiplikation von a und b?
- Als Mass für die Effizienz wählen wir die Gesamtzahl der elementaren Operationen.
  - Verdoppeln, Halbieren, Test auf "gerade", Addition
  - Im rekursiven Code: maximal 6 Operationen pro Aufruf
- Wesentliches Kriterium:
  - Anzahl rekursiver Aufrufe
    bzw.
  - Anzahl Schleifendurchläufe
    (im iterativen Fall)

```
static int f(int a, int b) {
   if (b == 1)
      return a;
   if (b % 2 == 0)
      return f(2*a, b/2);
   return a + f(2*a, b/2);
}
```

- $\frac{b}{2^x} \leq 1 \Rightarrow x \geq \log_2 b$

  Also nicht mehr als $6 \log_2 b$ elementare Operationen