

# Übung 1: Einführung

<http://people.f4.htw-berlin.de/~hebold/htw/pka/exercises/einf%C3%BChrung.pdf>

Gegeben:  $f_1(x) = x^2 - 2$ ,  $f_2(x) = x - 5$ ,  $f_3(x) = e^{x-2}$ ,  $f_4(x) = \ln(x + 5)$ ,  
 $f_5(x) = 5$

Bestimmen Sie die Ausdrücke der folgenden Funktionen:

- a)  $f_1 \circ f_2$
- b)  $f_1 \circ f_2 \circ f_3$
- c)  $(f_1 + f_4) \circ f_2$
- d)  $f_3 \circ (f_1 + f_4) \circ f_2$
- e)  $f_4 \circ f_4^{-1}$
- f)  $f_4^{-1} \circ f_4$
- g)  $f_2 \circ f_3 \circ f_3^{-1} \circ f_2^{-1}$
- h)  $f_1 \circ f_5 \circ f_5^{-1} \circ f_1^{-1}$

Bilden Umkehrfunktion:

$$y = \ln(x + 5) \Leftrightarrow e^y = e^{\ln(x+5)} \Leftrightarrow e^y = x + 5 \Leftrightarrow e^y - 5 = x \Rightarrow y = e^x - 5$$

Lösung:

- a)  $(x - 5)^2 - 2$
- b)  $(e^{x-2} - 5)^2 - 2$
- c)  $(x - 5)^2 - 2 + \ln(x)$
- d)  $e^{(x-5)^2 - 2 + \ln(x) - 2}$
- e)  $id$  (da  $f_4^{-1}$  existiert)
- f)  $id$  (da  $f_4^{-1}$  existiert)
- g)  $id$  (da  $f_2^{-1}, f_3^{-1}$  existiert)
- h)  $\nexists (f_5^{-1} \text{ existiert nicht})$

Schreiben Sie in Java jeweils ein Programm, das die Funktionen

a)  $x + y$

b)  $x * y$

c)  $x^y$

als int-Werten rekursiv berechnet

```
public static int add(int x, int y) {  
    if (y == 0) { return x; }  
    return 1 + add(x, y-1);  
}
```

```
public static int multiply(int x, int y) {  
    if (y == 0) { return 0; }  
    return x + multiply(x, y-1);  
}
```

```
public static int pow(int x, int y) {  
    if (y == 0) { return 1; }  
    return x * pow(x, y-1);  
}
```

Schreiben Sie in Java Programme zu Berechnung der Fakultät mit dem Datentyp BigInteger.

---

$$n! = 1 * 2 * \dots * (n - 1) * n = n * (n - 1)!$$

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

## Collatz-Problem Definition:

$f : \mathbb{N} \rightarrow \mathbb{N}$  mit  $f(1) = 1$ ,  $n$  gerade:  $f(n) = n/2$ ,  $n$  ungerade:  $f(n) = 3n + 1$

$F : \mathbb{N} \rightarrow \mathbb{N}$  mit  $F(1) = f(1) = 1$ ,  $F(n) = F(f(n))$

Collatz-Problem: Ist  $F$  für jedes  $n \in \mathbb{N}$  definiert, d.h.  $\forall n \in \mathbb{N} \exists F(n) \in \mathbb{N}$ ?

```
public static int f(int x) {
    if (x == 1) { return 1; }
    else if (x % 2 == 0) { return x / 2; }
    return 3 * x + 1;
}
public static int F(int x) {
    if (x == 1) return 1;
    return F(f(x));
}
public static int Flength(int x, int c) {
    if (x == 1) return c;
    return Flength(f(x), c + 1);
}
public static int Flength(int x) {return Flength(x, 1);}
```

Relationen, mit  $R \subseteq M \times M$

Reflexivität:  $(x, x) \in R$

Symmetrie:  $(x, y) \in R \Rightarrow (y, x) \in R$

Antisymmetrie:  $(x, y) \in R, (y, x) \in R \Rightarrow x = y$

Asymmetrie:  $(x, y) \in R \Rightarrow (y, x) \notin R$

Transitivität:  $(x, y) \in R, (y, z) \in R \Rightarrow (x, z) \in R$

Funktion: bijektiv = surjektiv (rechtstotal, isOnto) +  
injektiv (linkseindeutig, isOneOne)

TODO: Bild Bijektivität

Mengen:  $A = \{3, 4\}$ ,  $B = \{\{3, 4\}\}$

Welche Behauptung stimmt?

- a)  $A = B$
  - b)  $A \subseteq B$
  - c)  $A \subsetneq B$
  - d)  $|A| = |B|$
- 

- a) Nein, da unterschiedlich mächtig. siehe 4.)
- b) Nein, da gelten muss:  $\forall x \in A : x \in B$ , aber hier:  $3, 4 \notin B$
- c) A keine echte Teilmenge von B, da gelten muss:  
 $A \subset B \wedge A \neq B \Rightarrow a.) \wedge \neg b.)$  , a.) ist falsch
- d) Nein, da  $2 \neq 1$

Das kartesische Produkt zweier Mengen  $A$  und  $B$  ist wie folgt definiert:

$$A \times B = \{(x, y) : x \in A \wedge y \in B\}$$

Prüfen Sie, ob das kartesische Produkt assoziativ ist, d.h. ob für Mengen  $X, Y, Z$  gilt:  $(X \times Y) \times Z = X \times (Y \times Z)$

---

Nein, da die Tupel  $((x, y), z)$  und  $(x, (y, z))$  unterschiedlich sind, also eine andere Struktur haben.



Welcher der folgenden Ausdrücke ist korrekt?

- a)  $0 \in \emptyset$
- b)  $0 = \emptyset$
- c)  $0 \subseteq \emptyset$
- d)  $\{0\} \subseteq \emptyset$

- 
- a) Nein, die leere Menge hat keine Elemente
  - b) Nein, Zahlen sind keine Menge
  - c) Nein, siehe b.)
  - d) Nein, siehe a.)

Sei  $X$  eine Menge endlicher Größe und  $2^X$  die Potenzmenge von  $X$ .  
Welches Ergebnis liefern:

- a)  $|2^X \cup X|$
- b)  $|2^X| \cup |X|$

---

Beispiel: Potenzmenge  $X = \{a, b\} \Rightarrow 2^X = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$

- a)  $2^X \cup X = \{\emptyset, \{a\}, \{b\}, \{a, b\}, a, b\} \Rightarrow |2^X \cup X| = 2^{|X|} + |X|$
- b)  $\nrightarrow$  Vereinigung von Zahlen, nicht Mengen

## Übung 2: Konzepte Abstraktion

[http://people.f4.htw-berlin.de/~hebold/htw/pka/exercises/  
konzepte-Abstraktion.pdf](http://people.f4.htw-berlin.de/~hebold/htw/pka/exercises/konzepte-Abstraktion.pdf)

Knappsack:

Bildet von unterschiedlichen Summen eine Menge von  $n$  Summanden.  
(vgl. Merkle-Hellman Verfahren)

Beispiel: Die Folge der Werte  $(3, 4, 5)$  liefert die Summen  
 $0, 3, 4, 5, 7, 8, 9, 12$ .

---

Bilde Potenzmenge der Eingabe und addiere Elemente dieser einzelnen Mengen.

Zum Beispiel:  $2^{\{3,4,5\}} = \{\emptyset, \{3\}, \{4\}, \{5\}, \{3, 4\}, \{4, 5\}, \{3, 5\}, \{3, 4, 5\}\}$

Nennen Sie mindestens 3 Gründe für Abstraktion.

---

- a) **Wiederverwendbarkeit** von allgemeinen Problemlösungen
- b) **Klassifizieren** von Problemen, erkennen der Struktur
- c) Allgemeine Lösung zu detaillierten Problemen ( $\approx$  **Kompression**)
- d) **Vereinfachung**, reduzieren auf gemeinsame Eigenschaft

## Inwiefern wird beim Programmieren ganz generell abstrahiert?

---

- ▶ OOP (A)  $\Leftrightarrow$  reale Welt (K)
- ▶ Programmcode (A)  $\Leftrightarrow$  Problembeispiele (K)
- ▶ Programme (A)  $\Leftrightarrow$  Prozesse / Programmlauf (K)  
(vgl. Debuggen - ständiger Kontextwechsel)

Legende: Konkretisierung (K), Abstraktion (A)

Inwiefern wird bei der **strukturierten** Programmierung abstrahiert?

---

if, for, while-Konstrukte ersetzen Sprungbefehle

Beispiel:

```
while(c > 0) {  
    c--;  
}
```

wird zu

```
start:  
if(c <= 0) goto end;  
    c--;  
    goto start;  
  
end: ...
```

Inwiefern wird bei der **prozeduralen** Programmierung abstrahiert?

---

- ▶ parametrisierte Prozeduren ersetzen alle Werte-Kombinationen beim Aufruf

Beispiel:  $f(x) \hat{=} f(1), f(2), \dots f(n)$

- ▶ Prozeduren ersetzen konkrete Implementierungen ( $\approx$  **Blackbox**)

Beispiel: `sort(array[] field)` sortiert ohne, dass bekannt ist wie.



Inwiefern wird bei der **modularen** Programmierung abstrahiert?

---

- ▶ Kein konkreten Zustände, d.h. nur statische (vgl. `static`) Werte ( $\approx$  **Zustandslos**)
- ▶ Implementierung unbekannt, Referenzierung über Name ( $\approx$  **Blackbox**)

Beispiel: `import java.io`  $\rightarrow$  `import java.nio` bei gleicher API

## Inwiefern wird bei der **objekt-orientierten** Programmierung abstrahiert?

- Vererbung, generischen Datentypen

Beispiel: A extends B

(A Konkretisierung von B ↔ B Generalisierung/Abstraktion von A)

- Polymorphismus, dynamisches Binden

Beispiel:

```
class B {  
    void f() { out.println("B"); }  
}
```

```
class A extends B {  
    void f() { out.println("A"); }  
}
```

```
B testA = new A();  
testA.f(); // "A"  
B testB = new B();  
testB.f(); // "B"
```

## Inwiefern wird bei der **funktionalen** Programmierung abstrahiert?

- ▶ Nur Funktionen und Rückgabewerte, keine Unterscheidung zwischen Daten(typen) und Objekten  
Beispiel:

```
function addiere(x,y) {  
  if (typeof y === "undefined" ) {  
    return function (y) { return x + y; }  
  }  
  return x + y;  
}  
addiere(2,4); // 6  
var addiere_zu_drei = addiere(3)  
addiere_zu_drei(5) // 8
```

- ▶ Überladen von Funktionen  
Beispiel: `summe(a,b,c)` (A) und `summe(a,summe(b,c))` (K)

## Inwiefern wird bei der Programmierung abstrakter Datentypen abstrahiert?

---

- Lists, Arrays als Abstraktion  
vgl. Gruppe (Math.), generische Datentypen  
Beispiele:

```
static Object getFirst(ArrayList b) {  
    return ...  
}
```

```
ArrayList a = new ArrayList<Integer>();  
...  
<?> c = getFirst(a);
```

Beim Abstraktionskonzept wird auf verschiedenen konkreten Objekten mit einem Namen referiert, wobei die Besonderheiten unberücksichtigt bleiben - von diesen wird abstrahiert.

Bei der Konkretisierung wird umgekehrt einem Namen ein bestimmtes konkretes Objekt zugeordnet - der Name wird gebunden.

Wann erfolgt im Rahmen der Programmierung die Konkretisierung, d.h. die Bindung eines Namens?

---

Die Konkretisierung erfolgt zur **Laufzeit**, d.h. bei der Zuweisung wird der Typ und Name konkret festgelegt.

Beispiel:

```
// Abstraktion: Instanziierung  
List x = new ArrayList();  
// Konkretisierung: Zuweisung, Casting  
void doSth(List x) {ArrayList y=(ArrayList)x;}
```

Imperative Programmierung:

Von was wird durch einen Variablennamen abstrahiert?

---

Vom konkreten Wert hinter dem Variablennamen, da nur die Referenz auf den Wert benutzt wird.

Beispiel: `int x = 3; x = 7;`

Imperative Programmierung:  
Von was wird durch Pointer abstrahiert?

---

Von der Speicheradresse

Beispiel:

```
int *myPointer = 3 //Wert  
myPointer = 1234 // Speicheradresse
```

Imperative Programmierung:

Von was wird durch eine Initialisierung `int i=42` bzw. Zuweisung abstrahiert?

---

- Speicherreservierung + Zuweisung und Belegung
- Darstellung

Beispiel: Big Endian / Little Endian (dt.: Byte-Reihenfolge)



## Imperative Programmierung:

Von was wird durch

- if-Abfrage
- for-Schleife `for(int i=0;i<a;i++) block`
- while-Schleife

abstrahiert?

---

if-, for- und while-Konstrukte ersetzen verschiedene Goto-Anweisungen und Labels

```
while(c>0) {  
    c--;  
}
```

wird zu

```
start:  
if(c <= 0) goto end;  
    c--;  
    goto start;  
  
end: ...
```

Imperative Programmierung:

Von was wird durch eine

- a) Prozedur (void)
  - b) Funktion (non-void)
- abstrahiert?

---

a) Implementierung

b) Implementierung und Rückgabewert von der Funktion

Imperative Programmierung:

Von was wird in C und C++ und Java durch den abstrakten Datentyp Array abstrahiert?

---

Von den Speicherbereichen, relativer Zugriff

C und C++: Speicherbereiche sind zusammenhängend

Beispiel:

```
int arr [] = {1,2,3};  
arr[0]
```

Java: Speicherbereiche sind nicht zwingend zusammenhängend

Beispiel:

```
new int [] {1,2,3}[0]
```

Imperative Programmierung:

Funktionen werden in C, C++ und Java durch Aufrufe zur Laufzeit konkretisiert. Signatur und Methode sind die Abstraktionen. Zusätzlich bietet C++ Funktionen mit default Parametern. Was bedeuten diese für Abstraktion und Konkretisierung?

---

Default Parameter ermöglichen es, dass die überladene Methode nicht konkret die ursprüngliche Methode aufrufen muss. Es kann abstrakt die selbe Signatur mit den default Parametern verwendet werden.

Imperative Programmierung:

Von was wird in C++ durch eine inline-Funktion abstrahiert?

---

Wie Makros als Textersetzung ohne Stack / Heap, jedoch wie Funktion mit Auswertung. Konkrete Ersetzung des Wertes ohne Aufbau Stack / Heap.

Abgrenzung: Makro  $\Leftrightarrow$  Inline-Funktion

Haben beide keine Stack / Heap, d.h. beide arbeiten mit Textersetzung.

Inline-Funktionen sehen aber aus wie Funktionen. Problem bei

Texterersetzung mit Mehrfachaufruf, Beispiel:

```
Max(x,y) = {x > y ? x : y}
```

```
Max(x++, y++)
```

Objektorientierte Programmierung:

Von was wird in Java durch eine Referenz `Type ref` abstrahiert?

---

Type: Für alle Objekt-Typen und deren Ableitung von `Type`  
`ref` steht für ein konkretes Objekt vom Typ `Type` oder einer Subklasse davon.

## Objektorientierte Programmierung:

Die meisten objektorientierter Sprachen verfügen über primitive Datentypen wie z.B. `int`. Warum haben diese primitiven Datentypen aus der Sicht des Abstraktionskonzepts einen Sonderstatus?

---

- ▶ Call-by-Value: Passen direkt in Speicher ohne Referenzzugriff
- ▶ keine Kapselung notwendig/möglich
- ▶ Vererbung nicht möglich, da keine Klasse.

Objektorientierte Programmierung:

Was wird durch die ausschließliche Verwendung von Klassen, Objekten und Referenzen, d.h. durch die Streichung der primitiven Datentypen, im Sinne des Abstraktionskonzepts erreicht?

---

- ▶ Kontinuität (Einheitlichkeit)
- ▶ Gemeinsamer Oberdatentyp, von dem alles erbt → ein einziger Vererbungsbaum



Objektorientierte Programmierung:

C++ und Java kennen die Möglichkeit des **overriding** (überschreiben).  
Inwiefern handelt es sich um eine Abstraktion? D.h. von welchen  
konkrete Elementen wird abstrahiert?

---

Der Funktionsname ist die Abstraktion von der konkreten  
Implementierung der Objekt-Methode, da sich durch das überschreiben  
die Implementierung je nach Konkretisierung ändert.

Objektorientierte Programmierung:

Mehrfachvererbung ist in Java bei Klassen nicht zugelassen.

- a) Nennen Sie eine Begründung im Rahmen des Abstraktionsprinzips.
  - b) Wieso ist Mehrfachvererbung bei Interfaces zugelassen?
  - c) Wie löst C++ die genannten Probleme?
- 

- a) Overriding von Methoden ist durch die Einfachvererbung eindeutig
- b) Es steckt keine konkrete Implementierung dahinter
- c) Durch die Reihenfolge der Vererbung (erst beste Funktion wird genutzt)

Objektorientierte Programmierung:  
Inwiefern handelt es sich bei der Definition von Superclasses  
(Oberklassen) um eine Abstraktion?

---

- ▶ Verallgemeinerung (Generalisierung / Abstraktion) der konkreten Klasse, mit allgemeineren und weniger Eigenschaften
- ▶ Allgemeingültige Oberklasse für alle Unterklassen

Funktionale Programmierung:

Inline-Funktionen sind Teil der meisten funktionalen Sprachen.

Beschreiben Sie im Rahmen des Abstraktionskonzepts das Problem mit rekursiv definierten (inline-) Funktionen.

---

Eine rekursive Definition ist bei Inline-Funktionen (vgl. Makros) nicht möglich, da die Textersetzung unendlich ausgeführt wird.

Beispiel:

```
inline int add(a, b) {  
    if(b == 0) return 1; // ignoriert,  
    return add(a, b-1) + 1; // da hier nur ↪  
        Textersetzung  
}
```

## Funktionale Programmierung:

Inwiefern kann man sagen, dass in rein funktionalen Sprachen auf einer höheren Stufe der Abstraktion programmiert wird?

---

- ▶ Alles ist eine Funktion: Werte, Klassen und das Programm selbst. Es gibt keine Unterscheidung
- ▶ Funktionen können (zur Laufzeit) erzeugt und verändert werden.

Beispiel: Lambda-Ausdrücke

Funktionale Programmierung:

In rein funktionalen Sprachen sind Funktionen als Parameter und Rückgabewerte von Funktionen zugelassen.

- a) Inwiefern wird dadurch eine höhere Stufe der Abstraktion erreicht, als Paradigmen bei Sprachen, die dieses Feature nicht haben?
  - b) Ist es möglich, durch diese Erweiterung Probleme zu lösen, die in imperativen Sprachen nicht gelöst werden können?
- 

- a) Abstraktion der Kontrollstruktur: dynamischer Kontrollfluss (Bsp: Callbacks), Funktionen können zur Laufzeit konstruiert/modifiziert werden.
- b) Nein. wie bei imperativen Sprachen. Gegenbsp.: Ackermann ist imperativ nicht lösbar, sondern nur rekursiv. TODO: Frage

# Übung 3: Paradigmen

[http://people.f4.htw-berlin.de/~hebold/htw/pka/exercises/  
konzepte-Paradigmen.pdf](http://people.f4.htw-berlin.de/~hebold/htw/pka/exercises/konzepte-Paradigmen.pdf)

Das von-Neumann-Rechnerkonzept (auch von-Neumann-Architektur) zählt zur archetypischen Realisierung des imperativen Programmierparadigmas. Warum?

---

Imperative Konzept  $\hat{=}$  Befehlsorientiert  
Fetch, Execute-Zyklus



Die Turing-Maschine realisiert ebenfalls das imperativen Programmierparadigma. Warum?

---

Jeder Zustand verknüpft über Befehle, vgl. Überföhrungsfunktion

Wieso wird vom von-Neumann-Rechner**konzept** aber von der Turing-**Maschine** gesprochen?

Konzept: Abstraktion

Maschine: Konkrete Idee (auch wenn so nicht realisierbar, durch das unendlich lange Band)

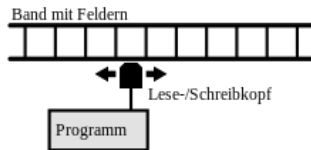
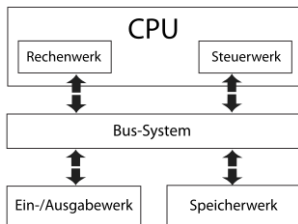


Abbildung : Von-Neumann, Turing-Maschine

Im Zusammenhang mit dem Neumann-Rechnerkonzept ist die Rede vom von-Neumann-Flaschenhals, wenn Nachteile des Konzepts genannt werden.

- a) Was ist darunter zu verstehen?
  - b) Gibt es eine vergleichbare Problematik für die Turing-Maschine?
- 

- a) Alle Befehle / Daten müssen durch den Bus
- b) Schreib-/Lesekopf kann pro Zeiteinheit entweder schreiben oder lesen

Nennen Sie wenigstens einen konzeptionellen Unterschied zwischen von-Neumann-Rechnerkonzept und Turing-Maschine.

---

Von TM ausgehend:

- Daten und Programme liegen **nicht** im selben Speicher
- keine Nummerierung/Adressierung eines Feldes auf dem Band
- keine Sprungadressen
- kann nur 1 Feld gehen pro Befehl

Setzt die Turing-Maschine das von-Neumann-Rechnerkonzept um?

---

**Nein**, weil

- Bei TM: Daten  $\neq$  Programme
- TM hat keine Sprungadresse

oder **Ja** mit Einschränkungen (vgl. Nein)

- Arbeiten befehlsorientiert
- Arbeiten deterministisch

Wie könnte das Paradigma der strukturierten Programmierung in das von-Neumann-Rechnerkonzept integriert werden?

---

Überwachen bzw. Regeln der Sprunganweisungen.  
D.h. begrenzter Bereich (Scope) z.B. bei if-Anweisungen

Wieso verletzt das Konzept der lokalen static-Variablen in C das Paradigma der funktionalen Programmierung?

---

Funktionsausgabe nur abhängig von Eingabe. D.h. bei gleicher Eingabe gleiche Ausgabe (Idempotent, Deterministisch).

```
int f(int i) {  
    // Ausführung bei Objekt-Init,  
    // nicht bei Methodenaufruf  
    static int x = 0;  
    x++;  
    return x+i;  
}
```

## Wieso verletzen Pointer in C das Paradigma der funktionalen Programmierung?

Paradigma der f. Programmierung: Funktionsausgabe nur abhängig von Eingabe. D.h. bei gleicher Eingabe gleiche Ausgabe.

```
int f(int *i) {  
    // Veraendern der Speicheradresse und  
    // somit der Eingabe  
    *i = 1234;  
    ...  
}
```



In Java gibt es mit dem Collection-Framework eine Reihe von sogenannten Container-Klassen. Welches objektorientierte Programmierparadigma verletzen Objekte z.B. der Klassen ArrayList oder Vector?

---

Es werden Referenzen gespeichert. D.h. die Datenkapselung ist verletzt. Werte müssten unveränderlich (immutable) sein.

```
class Dummy{int value;}  
...  
Dummy example = new Dummy()  
ArrayList<Dummy> list = new ArrayList<Dummy>()  
list.add(example);  
  
// Zugriff auf value ueber 2 Wege:  
example.value = 1;  
list.get(0).value = 2;
```

Wie müsste das Funktionskonzept in C beschränkt bzw. erweitert werden, damit es nicht zu Verletzungen des Paradigmas der funktionalen Programmierung kommen kann?

---

Zustandslosigkeit durch:

- kein static und Pointer
- keine Systemaufrufe

Das funktionale Programmierparadigma, das die referentielle Transparenz der Variablen fordert, wird in C durch die Zuweisung verletzt.

- a) Wie müssten die Regeln für die Verwendung der Zuweisung geändert werden, damit die Zuweisung in das funktionale Konzept passt?
- b) Welcher Art von Anweisung entspräche die veränderte Zuweisung dann?

- 
- a) Unveränderliche Variablen, Einmal-Zuweisung, bzw. nur Init.
  - b) Konstanteninitialisierung

Die referentielle Transparenz sorgt dafür, dass Programme in rein funktionalen Sprachen problemlos nebenläufig abgearbeitet werden können.

- a) Erklären Sie den Zusammenhang an einem Beispiel.
  - b) Erklären Sie an einem Beispiel den Zusammenhang fehlender referentieller Transparenz und Problemen bei nebenläufig ausgeführten Programmen.
- 
- a) Werte unveränderlich (immutable), d.h. kein Zustand und stark begrenzter Gültigkeitsbereich. Funktionen sind Thread-Safe, da Parameter nicht von außen geändert werden können.  
Beispiel: Immutable-Klassen sind automatisch thread-Safe, parallelisierbar und skalierbar.
  - b) Objekt-Parameter können während Thread-Unterbrechung über eine Referenz außerhalb der eigentlich Funktion verändert werden und haben somit einen anderen Zustand. Kein Determinismus, Race-Condition möglich.

Objektorientierte Sprachen kennen sogenannte inline-Funktionen.

- a) Wieso sind inline-Funktionen in objektorientierten Programmiersprachen implementiert?
  - b) Sind inline-Funktionen in rein funktionalen Programmiersprachen sinnvoll?
  - c) Welches Problem ergibt sich aus inline-Funktionen im Rahmen einer rein funktionalen Sprache?
- 

- a) Performancevorteile, da weniger Overhead ohne Heap / Stack.
- b) Ja, erhöhen Geschwindigkeit
- c) Rekursionen können nicht aufgelöst werden. Daher nur als Zusatz, nicht als Ersatz für normale Funktionen.

Angenommen in C würden innerhalb von parametrisierten Makros Zuweisungen nicht mehr zugelassen sein.  
Inwiefern verletzen Makros dann trotzdem weiterhin Paradigmen der funktionalen Programmierung?

---

Variablen sind außerhalb vom Makro gültig, d.h vor oder nach dem Aufruf / Ersetzung.

Durch die Textersetzung können Variablen(namen) genutzt werden, die erst im nachfolgenden Programm definiert werden. Dadurch ist Idempotenz / Determinismus verletzt.

Beispiel:

```
#define printX print x
int x = 100;
printX(); // Ausgabe: 100
```

## Übung 4: Primitiv rekursive Funktionen

[http://people.f4.htw-berlin.de/~hebold/htw/pka/exercises/  
algorithmen-rekursiveFunktionen.pdf](http://people.f4.htw-berlin.de/~hebold/htw/pka/exercises/algorithmen-rekursiveFunktionen.pdf)

Formulieren Sie die folgenden Ausdrücke mit Hilfe von  $F = +, \cdot$  und  $N, O, P_i^m, S^{n+1}, R$ :

a)  $a \cdot b + c$

d)  $4 \cdot (3 + x)$

e)  $(a + b) \cdot (a + b)$

h)  $(a + b) \cdot (c + d)$

---

a)  $S(+, S(\cdot, P_1^3, P_2^3), P_3^3)(a, b, c)$

d)  $S(\cdot, P_1^3, S(+, P_2^3, P_3^3))(4, 3, x)$

e)  $S(\cdot, +, +)(a, b)$

h)  $S(\cdot, S(+, P_1^4, P_2^4), S(+, P_3^4, P_4^4))(a, b, c, d)$



Formulieren Sie eine primitiv rekursive Funktion, die

- a) die arithmetische Differenz bestimmt.
  - b) die das Vorzeichen prüft und 0 bei 0 und 1 bei Werten  $> 0$  liefert.
  - c) das Maximum von zwei Zahlen liefert.
- 

a)

$$\begin{aligned} pre &= S(R(O, P_2^3), P_1^1, O) \\ \dot{-} &= S(R(P_1^1, pre \circ P_1^3), P_2^2, P_1^2)(a, b) \end{aligned}$$

b)  $sign = S(R(O, N \circ O \circ P_2^3), P_1^1, O)$

c)

$$\begin{aligned} first &= S(\cdot, S(gt, P_1^2, P_2^2), P_1^2) \\ sec &= S(\cdot, S(gt, P_2^2, P_1^2), P_2^2) \\ eq &= S(\cdot, S(eq, P_1^2, P_2^2), P_1^2) \\ max &= S(add, S(add, first, sec), eq) \end{aligned}$$

Formulieren Sie eine primitiv rekursive Funktion, die

- a) den ganzzahligen Rest der Werte  $x$  und  $y$  bestimmt.
  - b) die Teilbarkeit einer Zahl  $x$  hinsichtlich  $y$  prüft.
- 

a)

$$\text{mod}(a, b) = x < y ? x : \text{mod}(x - y, y)$$

$$\text{mod}(a, b) = x \cdot \text{gt}(y, x) + \text{mod}(x - y, y) \cdot \text{gte}(x, y)$$

$$R_g = S(+, S(\cdot, x, \text{gt}(y, x)), S(\cdot, S(P_1^4, S(\div, x, y), y), \text{gt}(x, y)))$$

- b)  $\text{neg} = S(R(N \circ O, O \circ P_2^3), P_1^1, P_1^1)$  mit  $\text{neg} \circ \text{sign} \circ \text{mod}(x, y)$

0 und N sind Funktionen  $\mathbb{N} \rightarrow \mathbb{N}$ ,  $P, S^{n+1}$  und R dagegen Funktionen auf Funktionen, dh. Operatoren.  $S^{n+1}$  ist wie folgt definiert:

- a) Wieso kann man sagen, dass  $S^{n+1}$  für mehrere Operatoren steht?
  - b) Wieviele verschiedene Operatoren ergeben sich aus  $S^{n+1}$ ?
  - c) Was bedeutet es für die Programmierung, dass es mehrere Operatoren  $S^{n+1}$  gibt?
- 

- a) Die beliebigen f's bilden ein Funktionsschema
- b)  $n$
- c) Overloading muss möglich sein für unterschiedliche Stelligkeiten  $n$

Der Ausdruck  $S^{n+1}(f, g_1, \dots, g_n)$  abstrahiert von der Anzahl  $n$  möglicher Funktionen und der Stelligkeit von  $f$ .

a) Begründen Sie:

- i) Es handelt sich also eigentlich um  $n$  verschiedene Operatoren.
- ii) Die Stelligkeit von  $f$  ist  $n$ .
- iii) Die Stelligkeit der Funktionen  $g_1, \dots, g_n$  ist  $m$ .

b) Was bedeuten i) – iii) für die Implementierung (Programmierung) von  $S^{n+1}$ ?

---

a) Definition:  $S^{n+1}(f, g_1, \dots, g_n)(x_1, \dots, x_m) = f(g_1(x_1, \dots, x_m), \dots, g_n(x_1, \dots, x_m))$

- i) Ja, da  $g_i$   $n$ -mal aufgerufen wird.
- ii) Ja, da die Parameter  $g_1, \dots, g_n$  sind.
- iii) Ja, da die Parameter von  $g_i$   $x_1, \dots, x_m$  ist.

b) i) Variable Parameterliste muss in der Sprache vorhanden sein.  
ii) Variable Parameterliste muss in der Sprache vorhanden sein.  
iii) Variable Parameterliste muss in der Sprache vorhanden sein.

Die Stelligkeit  $m$  der Funktionen  $g_1, \dots, g_n$  spielt für den Operator  $S^{n+1}(f, g_1, \dots, g_n)$  im Rahmen des Konzepts der rekursiven Funktionen und bei den Fragestellungen der Berechenbarkeit keine Rolle.

- a) Was ist mit dieser Aussage gemeint?
  - b) Was bedeutet das für die Programmierung von  $S^{n+1}$  ?
- 

- a)
- b)

Die Stelligkeit  $n$  der Funktion  $f$  spielt für den Operator  $S^{n+1}(f, g_1, \dots, g_n)$  im Rahmen des Konzepts der rekursiven Funktionen und bei den Fragestellungen der Berechenbarkeit keine Rolle.

- a) Was ist mit dieser Aussage gemeint?
  - b) Was bedeutet das für die Programmierung von  $S^{n+1}$  ?
- 

- a)
- b)

Die Mehrstelligkeit der rekursiven Funktionen führt nicht aus der Menge der rekursiven Funktionen hinaus. Es spielt also keine Rolle, ob man  $S$ ,  $R$  und  $\mu$  auf  $n$ -stellige oder ein-stellige Funktionen anwendet. Mit anderen Worten: Alle  $n$ -stelligen rekursiven Funktionen können durch einstellige rekursive Funktionen dargestellt werden.

a) Beschreiben Sie die Umsetzung.

---

a) Durch das Currying wird eine mehrstellige Funktion  $f(x_1, x_2)$  in mehrere einstellige Funktionen, die hintereinander ausgeführt werden, umgeformt:  $f(x_1)(x_2)$

Sie wollen eine primitiv rekursive Funktion definieren, die die Funktion

$$f(x) = \begin{cases} 1 & \text{wenn } x = 1 \\ \frac{x}{2} & \text{wenn } x \text{ gerade} \\ 3 \cdot x + 1 & \text{wenn } x \text{ ungerade} \end{cases}$$

berechnet, d.h. die Formulierung erfolgt ausschließlich mit den Funktionen bzw. Funktionssymbolen  $N, 0, P_i^m, S, R$ .

a) Warum scheitern Sie?

a) ??



Was bedeutet die Aussage: „Alle berechenbaren Funktionen können mit Hilfe der Funktionen  $0$ ,  $N$ ,  $P_i^m$ ,  $S^{n+1}$ ,  $R$  und  $\mu$  formuliert werden“ für die Programmierung?

---

Alle berechenbare Probleme können mit primitiv rekursiven Funktionen umgesetzt werden.

Was unterscheidet die folgenden Aufzählfunktionen  $\mathbb{N}^2 \rightarrow \mathbb{N}$  (Paare natürlicher Zahlen auf natürliche Zahlen) voneinander:

- ▶  $c(x, y) = \frac{(x+y)(x+y+1)}{2}$
- ▶  $p(x, y) = 2^x \cdot 3^y$

# Übung 5: Induktion

`http://people.f4.htw-berlin.de/~hebold/htw/pka/exercises/  
algorithmen-Induktion.pdf`

Was bedeutet es, wenn die Verankerung bei  $n=a$ , also z.B.  $n=5$  bewiesen wird, aber nicht für kleinere Werte?

---

Bewiesen erst ab  $n=5$  und aufwärts, bzw. Voraussetzung erst ab dann beweisbar anwendbar.

Kann man aus der Allgemeingültigkeit von  $\varphi$  schließen, dass  $\varphi(0)$  und  $\varphi(n) \Rightarrow \varphi(n^+)$  gelten?

---

Durch 0 und den Nachfolger schließt man auf die Allgemeingültigkeit, daher ist dieser Satz falsch... hier sind Prämisse und Konklusion vertauscht.

Angenommen  $\varphi$  wird für 0 bewiesen, ist für ein  $n = a > 0$  ungültig und  $\varphi(n) \Rightarrow \varphi(n + 1)$  kann wiederum gezeigt werden. Was besagt das für die Induktion?

---

Darf die bewiesene Verankerung im Induktionsschritt verwendet werden?

---

Induktionsschritt ist die Verallgemeinerung, die Verankerung wird mit konkreten Werten angewendet. Daraus folgt: Nein, darf man nicht.

Das Schema der vollständigen Induktion lautet:

Gilt für eine Menge  $A$

a)  $0 \in A$

b)  $x \in A \Rightarrow x^+ \in A$

$\models \mathbb{N} \subseteq A$

oder

Gilt für ein auf  $\mathbb{N}$  definiertes Prädikat  $\varphi$

a)  $\varphi(0)$

b)  $\varphi(x) \Rightarrow \varphi(x^+)$

$\models \forall x(\varphi(x))$

Geben Sie wenigstens 3 Möglichkeiten der Verallgemeinerung (Abstrahierung) an.

- a) beliebige Startwerte, statt 0
- b) mehrere Startwerte/Verankerungen
- c) andere Nachfolgerfunktion
- d)  $\mathbb{N}$  oder Menge  $A$  kann ersetzt werden
- e) Prädikat  $\varphi$  kann ersetzt werden



## Übung 5: $\lambda$ -Kalkül

[http://people.f4.htw-berlin.de/~hebold/htw/pka/exercises/  
algorithmen-lambdaCalculus.pdf](http://people.f4.htw-berlin.de/~hebold/htw/pka/exercises/algorithmen-lambdaCalculus.pdf)

Das  $\lambda$ -Kalkül unterscheidet zwei Arten von Ausdrücken: Auswertungen und Abstraktionen. Benennen Sie für jeden der Ausdrücke dessen Art und dann innerhalb des Ausdrucks gebundene Variablen und Rumpf bzw. Funktionsargument und Funktion.

a)  $\lambda a.(a \ \lambda b.(b \ a))$

b)  $\lambda x.\lambda y.\lambda z.((z \ x) \ (z \ y))$

---

Auswertung: in Klammern, hat Argumente

Abstraktion: hat *keine* Argumente,  $\sim$  Funktion  
Achtung: Nicht verwechseln mit Rumpf, der auch in Klammern stehen kann.

a) Abstraktion, da nicht in Klammern

Funktion:  $\lambda a.f$

Rumpf:  $(a \ \lambda b.(b \ a))$

Gebundene Variablen:  $a$  (was ist mit  $b$ ?)

b) wie 1)

Funktion:  $\lambda x.\lambda y.\lambda z.f$

Rumpf:  $((z \ x) \ (z \ y))$

Gebundene Variablen:  $x, y, z$

Das  $\lambda$ -Kalkül unterscheidet zwei Arten von Variablen: gebundene und freie. Benennen Sie für jeden der folgenden Ausdrücke diese.

a)  $\lambda x. \lambda y. (\lambda x. y \quad \lambda y. x)$

b)  $\lambda x. (x \quad (\lambda y. (\lambda x. x \quad y) \quad x))$

---

Alle Variablen sind gebunden.

Werten Sie folgende  $\lambda$ -Ausdrücke aus:

a)  $((\lambda x. \lambda y. (y \ x) \ \lambda p. \lambda q. p) \ \lambda i. i)$

b)  $((((\lambda x. \lambda y. \lambda z. ((x \ y) \ z) \ \lambda f. \lambda a. (f \ a)) \ \lambda i. i) \ \lambda j. j)$

Der Ausdruck wird als Wert in dem/der **Symbol/Variable** ersetzt, festgelegt durch  $\lambda$ **Symbol**. Beachte die Klammern um den jeweilige Funktion, die erst die Auswertung erlaubt.

a)  $((\lambda x. \lambda y. (y \ x) \ \underline{\lambda p. \lambda q. p}) \ \lambda i. i) \Rightarrow (\lambda y. (y \ \lambda p. \lambda q. p) \ \lambda i. i)$   
 $(\lambda y. (y \ \lambda p. \lambda q. p) \ \underline{\lambda i. i}) \Rightarrow (\lambda i. i \ \lambda p. \lambda q. p)$   
 $(\lambda i. i \ \underline{\lambda p. \lambda q. p}) \Rightarrow \lambda p. \lambda q. p$

b)  $((((\lambda x. \lambda y. \lambda z. ((x \ y) \ z) \ \underline{\lambda f. \lambda a. (f \ a)}) \ \lambda i. i) \ \lambda j. j) \Rightarrow$   
 $((\lambda y. \lambda z. ((\lambda f. \lambda a. (f \ a) \ y) \ z) \ \underline{\lambda i. i}) \ \lambda j. j)$   
 $((\lambda y. \lambda z. ((\lambda f. \lambda a. (f \ a) \ y) \ z) \ \underline{\lambda i. i}) \ \lambda j. j) \Rightarrow$   
 $(\lambda z. ((\lambda f. \lambda a. (f \ a) \ \lambda i. i) \ z) \ \underline{\lambda j. j})$   
 $(\lambda z. ((\lambda f. \lambda a. (f \ a) \ \lambda i. i) \ z) \ \underline{\lambda j. j}) \Rightarrow$   
 $((\lambda f. \lambda a. (f \ a) \ \lambda i. i) \ \lambda j. j)$   
 $((\lambda f. \lambda a. (f \ a) \ \underline{\lambda i. i}) \ \lambda j. j) \Rightarrow (\lambda a. (\lambda i. i \ a) \ \lambda j. j)$   
 $(\lambda a. (\lambda i. i \ a) \ \underline{\lambda j. j}) \Rightarrow (\lambda i. i \ \lambda j. j)$   
 $(\lambda i. i \ \underline{\lambda j. j}) \Rightarrow \lambda j. j$

Gegeben sind folgende  $\lambda$ -Ausdrücke:

- $\text{def } id = \lambda x.x$
- $\text{def } apply = \lambda f.\lambda x.(f \ x)$

Zeigen Sie, dass  $id = (apply \ (apply \ id))$

Hinweis: Literale aus verschiedenen eingesetzten  $\lambda$ -Ausdrücken sind nicht identisch trotz gleichem Namens.

$$(apply \ (apply \ id)) \Rightarrow (apply \ (\lambda f.\lambda x.(f \ x) \ id))$$

$$(apply \ (\lambda f.\lambda x.(f \ x) \ id)) \Rightarrow (apply \ \lambda x.(id \ x))$$

$$(apply \ \lambda x0.(\lambda x1.x1 \ x0)) \Rightarrow (apply \ \lambda x0.x0)$$

$$(apply \ \lambda x0.x0) \Rightarrow (\lambda f.\lambda x2.(f \ x2) \ \lambda x0.x0)$$

$$(\lambda f.\lambda x2.(f \ x2) \ \lambda x0.x0) \Rightarrow \lambda x2.(\lambda x0.x0 \ x2)$$

$$\lambda x2.(\lambda x0.x0 \ x2) \Rightarrow \lambda x2.x2 \Leftrightarrow \lambda x.x$$

Gegeben sind folgende  $\lambda$ -Ausdrücke:

- $\text{def } apply = \lambda f. \lambda x. (f \ x)$
- $\text{def } pair = \lambda x. \lambda y. \lambda z. ((z \ x) \ y)$

Zeigen Sie, dass  $apply = \lambda x. \lambda y. (((pair \ x) \ y) \ id)$

---

Gegeben sind folgende  $\lambda$ -Ausdrücke:

- $\text{def } id = \lambda x.x$
- $\text{def } apply = \lambda f.\lambda x.(f \ x)$
- $\text{def } pair = \lambda x.\lambda y.\lambda z.((z \ x) \ y)$
- $\text{def } self = \lambda x.(x \ x)$
- $\text{def } second = \lambda x.\lambda y.y$

Zeigen Sie, dass  $id = (self \ (self \ second))$

Wieso wird das abstrakteste  $\lambda$ -Kalkül als **typfrei** bezeichnet?

---

Arbeit nur auf Symbolen, reine Textersetzung



Bei  $\lambda$ -Kalkül-Ausdrücken wird von Auswertung und Abstraktion gesprochen. Erklären Sie an einem Beispiel, in wiefern bei  $\lambda$ -Kalkül-Ausdrücken abstrahiert und konkretisiert wird.

---

Mit welchen Programmierkonzept aus C ist das typfreie  $\lambda$ -Kalkül vergleichbar?

- a) Makros
- b) Templates
- c) Funktionen
- d) Pointern

---

Makros, da diese keine Textersetzung durchführen.

Im Zusammenhang mit der Auswertung von  $\lambda$ -Ausdrücken kann es zu Namenskonflikten kommen, die mit Hilfe der sogenannten  $\alpha$ -Konvertierung gelöst werden. Erklären Sie an einem Beispiel die Problematik. Wie wird das Problem konkret gelöst?

Die  $\alpha$ -Konvertierung ist das Umbenennen der Elemente. Bei  $(\lambda x.(x \ x) \ \lambda x.(x \ x))$  kommt es zu einem Konflikt. Nach der Konvertierung:  $(\lambda x.(x \ x) \ \lambda f.(f \ f))$

Die Auswertung von Ausdrücken wird als  $\beta$ -Reduktion bezeichnet.  
Welches Problem tritt hier auf? (Beispiel!)

---

Es können Konflikte auftreten, die durch die  $\alpha$ -Konvertierung gelöst werden müssen.

Beispiel:  $(\lambda x.(x \ x) \ \lambda x.(x \ x))$

Die einzige Möglichkeit im  $\lambda$ -Kalkül eine Wiederholung zu formulieren basiert auf dem sogenannten Fixpunktsatz. Was ist damit gemeint? Wieso lösen Fixpunkte das Problem der Wiederholung?

---

Tafelanachrieb?

## Übung 7: $\lambda$ -Kalkül - Fortsetzung

[http://people.f4.htw-berlin.de/~hebold/htw/pka/exercises/  
algorithmen-lambdaCalculus\\_cont.pdf](http://people.f4.htw-berlin.de/~hebold/htw/pka/exercises/algorithmen-lambdaCalculus_cont.pdf)

Definieren Sie im  $\lambda$ -Kalkül die

- a) die Null ( $=0$ )
  - b) die Nachfolgefunktion  $\text{succ}$
  - c) eine beliebige Zahl
- 

- a)  $\lceil 0 \rceil = \lambda x. x$
- b)  $\lceil n + 1 \rceil = \lambda x. ((x \perp) \quad \lceil n \rceil)$
- c)  $\lceil n \rceil$  = beliebig häufig anwenden von 2.) ?

Definieren Sie im  $\lambda$ -Kalkül die primitiv rekursiven Funktionen:

a) 0

b)  $N$

c)  $P_n^m$

d)  $S^{n+1}$



Wie lautet der Fixpunkt von:

- a) not
- b) succ

---

Info: <https://de.wikipedia.org/wiki/Fixpunkt-Kombinator>

Definieren Sie im  $\lambda$ -Kalkül das primitiv rekursive Funktionsschema **R**

Beschreiben Sie den Unterschied von  $=$  (Gleichheit) und  $\equiv$  (Identität) an einem Beispiel.

---

Gleichheit: Äquivalenz; kann auch Behauptung sein und soll sich logisch ergeben.

Identität: Definition linker Seite durch rechte Seite, vgl.

$A := B$ ,  $A =_{def} B$  oder hier  $A \equiv B$  für Definition von  $B$  zu  $A$ .

Bei der Definition der Substitution werden Funktionen auf zwei Arten miteinander verknüpft. Nennen Sie die beiden Arten. Beschreiben Sie den Unterschied an einem Beispiel.

---

Komposition, Mehrstelligkeit (Currying) Currying =  
$$(\dots (f(a_1)a_2) \dots a_n) = f(a_1, a_2, \dots, a_n)$$

Funktionen können durch Komposition oder Currying verknüpft werden.

- a) Beschreiben Sie den Unterschied an einem Beispiel.
- b) Welche der beiden Verknüpfungsarten entspricht einer Funktionsdefinition in Java?

---

Java: immer mehrstellige Funktionen, Gegenteil von funktionalen Sprachen

# Übung 7: Komplexitätsklassen

[http://people.f4.htw-berlin.de/~hebold/htw/pka/exercises/  
komplexit%C3%A4t.pdf](http://people.f4.htw-berlin.de/~hebold/htw/pka/exercises/komplexit%C3%A4t.pdf)

Die Komplexitätsklasse **P** wird üblicherweise als Entscheidungsproblem definiert.

- a) Formulieren Sie die entsprechende Definition.
  - b) Formulieren Sie **P** als Suchproblem.
-

Die Komplexitätsklasse **NP** wird üblicherweise als Entscheidungsproblem definiert.

- a) Formulieren Sie die entsprechende Definition.
  - b) Formulieren Sie **NP** als Suchproblem.
-



Die Zeitkomplexität eines Algorithmus wird in Abhängigkeit von der Länge der Eingabe auf der Turing-Maschine gemessen und nicht in Abhängigkeit vom Wert der Eingabe. Dabei bleibt die entsprechende Funktion  $time_F$ , die die Zeitkomplexität von Algorithmus F beschreibt, auf  $\{0,1\}^*$  definiert, dh.

$time_F : \{0,1\}^* \rightarrow \mathbb{N}_0 : x \mapsto \max\{steps_F(y) : |x| = |y|\}$  mit  $steps_F(y)$  als Funktion, die die Rechenschritte zu y liefert.

- a) Nennen Sie zwei Gründe für diesen Wechsel des Maßes. Mit  $time_F$  wird die folgende Relation definiert:  $x \sim y := time_F(x) = time_F(y)$
- b) Beschreiben Sie verbal, was die Relation besagt.
- c) Prüfen Sie, ob die Relation die drei klassischen Eigenschaften einer Relation erfüllt, dh. prüfen Sie, ob  $\sim$  eine Äquivalenzrelation ist.

Was besagen die Aussagen:

a)  $P = NP$

b)  $P \subset NP$

c)  $NP \subset P$

in der Definition mit **Turing-Maschinen** und **Beweissystemen**

---

Lösungsfunktion und Algorithmus sind etwas anderes.

- a) Erklären Sie den Unterschied an einem Beispiel.
  - b) Wieso beziehen sich die Begriffe **P** und **NP** auf Algorithmen und nicht auf Funktionen ?
- 

- a) Primzahl und Verfahren zur Berechnung, ob eine Primzahl ist oder nicht.
  - b) Wieso beziehen sich die Begriffe **P** und **NP** auf Algorithmen und nicht auf Funktionen ?
-

In welche der beiden folgenden Komplexitätsklassen gehört ein Programm, das aus einer gegebenen endlichen Menge alle Teilmengen erzeugt?

a) **P**

b) **NP**

Bestimmen Sie die Ableitungen der folgenden Funktionen:

a)  $f(x) = x * \ln x$

b)  $f(x) = \ln(\ln x)$

c)  $f(x) = \log_n x$

d)  $f(x) = \lg x$

e)  $f(x) = x^x$

f)  $f(x) = e^{\ln x}$

Lösung:

a)  $f'(x) = \ln x$  mit P.

b)  $f'(x) = \frac{1}{x * \ln x}$  mit K. + P.

c)  $f'(x) = \frac{1}{x * \ln n}$  mit P.

d)  $f'(x) = \frac{1}{x * \ln 10}$

e)  $f'(x) = \ln x * x^x$  mit K. + P.

f)  $f'(x) = 1 (= 1 * x^{1-1})$

Legende: Anwendung der Kettenregel (K), Produktregel (P)

Hinweise:

- ▶  $\lg = \log_{10}$ ,  $\ln = \log_e$ ,
- ▶  $e^{\ln(x)} = x$ , da aus  $a^b = c$  und  $b = \log_a c$  folgt  $a^{b=\log_a c} = c$
- ▶  $\log_a b = \frac{\ln b}{\ln a}$
- ▶  $\frac{a}{b} = a * b^{-1}$

Finden Sie zu jeder der gegebenen Funktionen  $f$  eine Funktion  $g$ , so dass  $f \in \mathbf{O}(g)$  gilt:

a)  $f(n) = f(n-1) + 10$  a)

b)  $f(n) = f(n-1) + n$  b)

c)  $f(n) = 2 * f(n-1)$  c)

d)  $f(n) = f(n/2) + 10$  d)

e)  $f(n) = f(n/2) + n$  e)

f)  $f(n) = 2 * f(n/2) + n$  f)

g)  $f(n) = 3 * f(n/2)$  g)

h)  $f(n) = 2 * f(n/2) + \mathbf{O}(n^2)$  h)

Nennen Sie jeweils zwei Beispiele für Probleme, die von:

- a) konstanter
- b) logarithmischer
- c) linearer
- d) quadratischer
- e) polynomieller
- f) exponentieller
- g) faktorieller

Wachstumsordnung sind.

Beschreiben Sie verbal oder auch formal die Bedeutung der Ausdrücke:

- a)  $f \in \mathbf{O}(n^2)$
- b)  $\mathbf{O}(f) = \mathbf{O}(n^2)$
- c)  $\mathbf{O}(f) \in \mathbf{O}(n^2)$
- d)  $n^2 \in \mathbf{O}(e^x)$
- e)  $f(x) = 1 + x^2 + \mathbf{O}(\log x)$
- f)  $\mathbf{O}(f) \subseteq \mathbf{O}(n^2)$

Hinweis: Falls ein Ausdruck sinnlos ist, sollten Sie das vermerken.

- 
- a)  $f$  ist Element der Menge aller Funktionen mit quadratischem Wachstum
  - b) Wachstumsklasse  $f$  ist äquivalent zur Wachstumsklasse  $n^2$
  - c)  $\nexists \mathbf{O}(n^2)$  enthält keine Mengen
  - d) existiert ein  $c$  für:  $n^2 \leq c * e^x$  ?
  - e) Formal:  $=$  macht keinen Sinn,  $\in$  wäre hier richtig
  - f) Teilmenge kann es sein, vgl. c)



Die Symbole  $\mathbf{O}$ ,  $\Omega$ ,  $\Theta$ ,  $o$  und  $\omega$  beschreiben Relationen zwischen Funktionen.

- a) Formulieren Sie die Aussagen, die erfüllt sein müssten, damit die genannten Relationen reflexiv, symmetrisch und transitiv sind.  
(Hinweis: Es geht ausdrücklich nicht darum, eine gültige Aussage zu formulieren, sondern nur um die Formalisierung.)
- b) Überprüfen Sie die Gültigkeit der Aussagen aus a).
- c) Handelt es sich bei den genannten Relationen jeweils um eine:  
Äquivalenzrelation, Quasiordnung, Ordnung und/oder Halbordnung