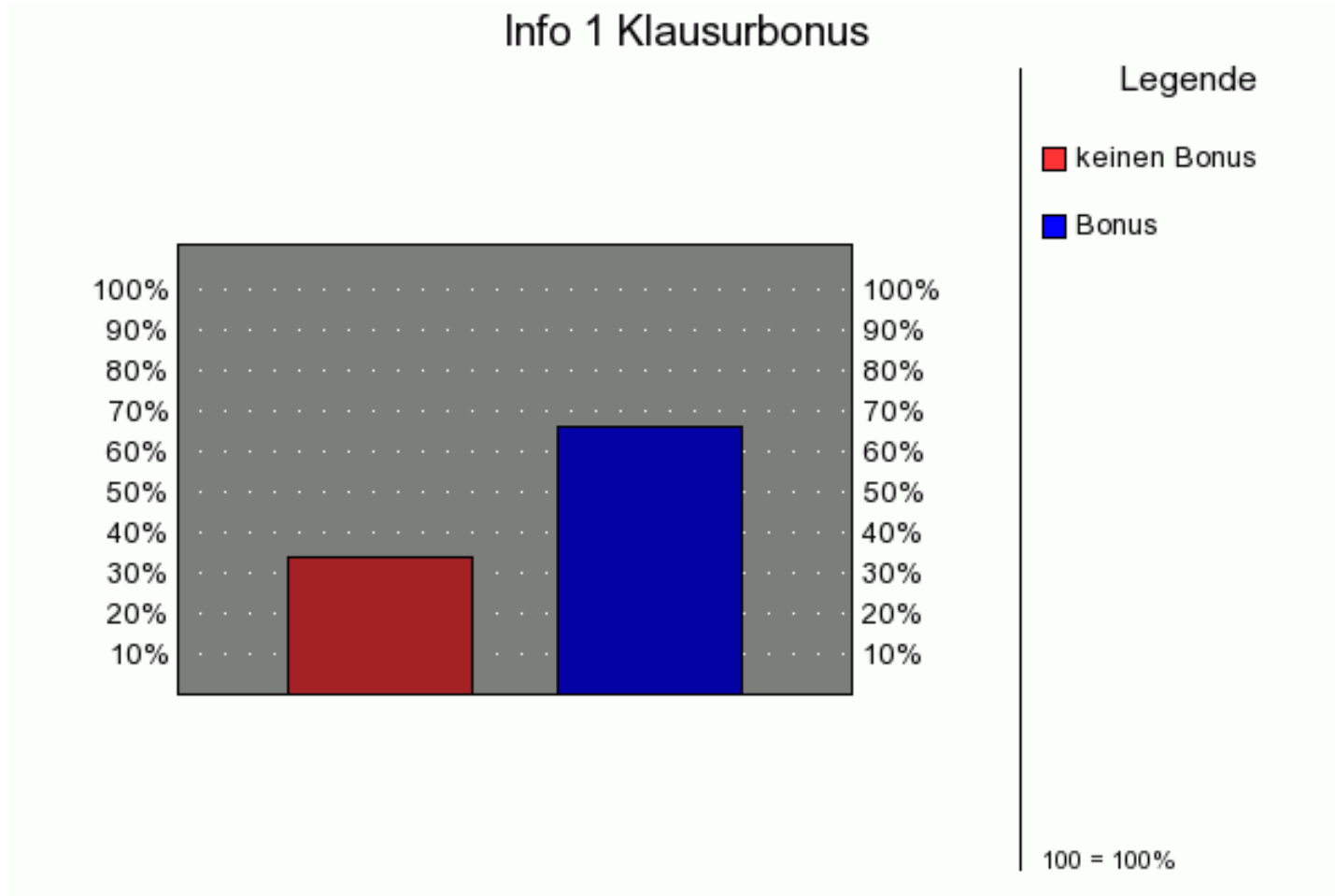
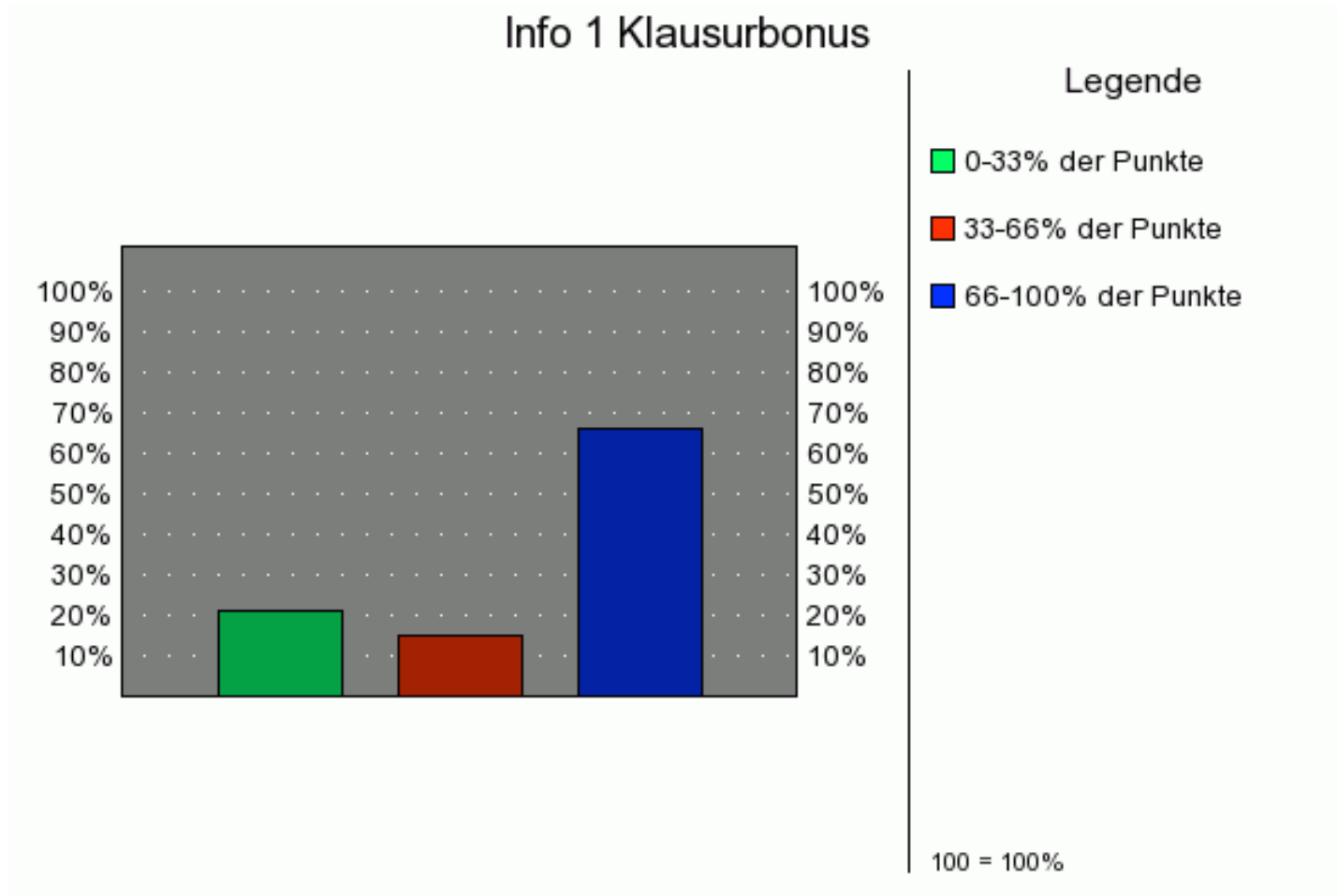


Übungen zu Informatik I Wintersemester 03/04

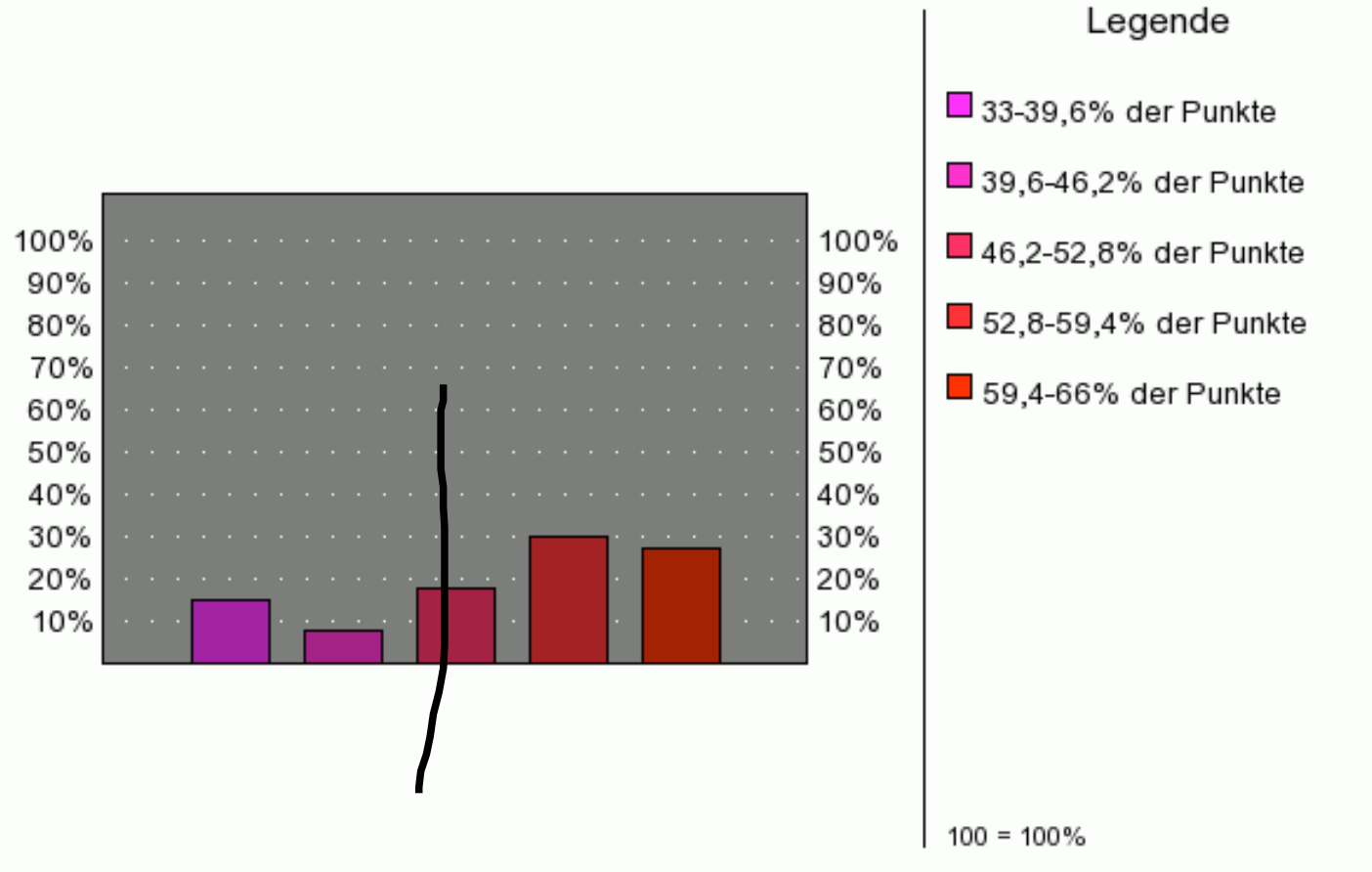
Übungsleiter: Dipl.-Inform. Tom Gelhausen







Info 1 Klausurbonus 33-66%



- Wem noch Punkte bis zur 2/3 Marke im ersten Teil fehlen, kann dies im zweiten Teil ausgleichen
- Es gab auf den Übungsblättern 0-9
 - 418 Theoriepunkte $\times 2/3 = 278$ Theoriepunkte
 - 154 Praxispunkte $\times 2/3 = 102$ Praxispunkte
- Beispiel:
 - Student Schorsch Schlaumeier hat 285,5 Theoriepunkte, aber nur 99 Praxispunkte \Rightarrow Ihm fehlen 3 Praxispunkte.
 - Wenn Schorsch Schlaumeier auf den Übungsblättern 10-15 nicht nur 2/3 der Theoriepunkte, sondern auch mindestens (2/3 der Praxispunkte +3) erreicht, bekommt er den Klausurbonus trotzdem.



- Auswertung nächste Woche (zus. mit der Auswertung der Umfrage)
- Aufgabe 2 (Markov Algorithmus)
 - Pfeile sind wichtig! „→“ Ableitungsschritt „⇒“ logische Folgerung
- Aufgabe 3 (Haskell)
 - Hinweise beachten! („Verwenden Sie für folgende Aufgaben keine Funktionen aus der ‚prelude.hs‘. Hierunter fällt auch ++!“)
 - ¼ der Gesamtpunktzahl (bestmöglich Note nur noch 2,0)
 - zusammen mit A6 (Prolog) über 40% der Punkte ➡ 3,0 bestenfalls!
- Aufgabe 5.1 (PL)
 - Skolemform von $(\exists x \exists y: P(x, y)) \rightarrow (\forall x \forall y: Q(x, y)) \wedge \exists y: R(y, y)$
 - Nur gültige Regeln verwenden
 - Alle gültigen Regeln dürfen verwendet werden (Kommutativgesetz)



- „Entwerfen Sie eine Prolog-Klausel `isHalbOrdnung`, die überprüft, ob eine gegebene Relation `rel` eine Halbordnung darstellt.“
- Klar: Halbordnung ist reflexiv, transitiv und antisymmetrisch.
- Aber: wie `reflexiv`, `transitiv` und `antisymmetrisch` definieren?
- So? `reflexiv :- rel(e1,e1), rel(e2,e2), rel(e3,e3), ...`
- Eigentlich lieber der Bauart: `reflexiv' :- rel(X,X).`
- Problem: Prolog prüft nur, ob ein `X` existiert, für das das gilt (\exists)
- Ziel: wir wollen aber eine Aussage über alle `X` treffen (\forall)



▪ Erinnerung PL:

$$\begin{aligned} \bullet \neg(\exists x:Q(x)) &= \forall x:(\neg Q(x)) && \triangleq && \forall x:(P(x)) = \neg(\exists x:\neg P(x)) \\ \bullet \neg(\forall x:Q(x)) &= \exists x:(\neg Q(x)) && && \swarrow \text{wenn } P(x) := \neg Q(x) \end{aligned}$$

▪ Mit Prolog können wir nur die Existenz prüfen

$$\text{ex} :- p(X) . \quad \triangleq \quad \exists x:P(x)$$

▪ Eine Aussage, die für alle x gilt, können wir nur erhalten, wenn wir erfolglos versuchen ein x zu finden, für das die negierte Aussage nicht gilt (1. Zeile)

- Wir wollen prüfen, ob eine Aussage P für alle x aus unserer Grundmenge U gilt: $\forall x \in U: P(x)$

- Die negierte Aussage lautet: $\exists x \in U: \neg P(x)$

- genauer: $\exists x: (x \in U) \wedge \neg P(x)$

- Damit: $\text{exGegenteil} :- \text{elem}(X), \text{not}(p(X)) .$

- und: $\text{fa} :- \text{not}(\text{exGegenteil}) .$



▪ Betrachte reflexiv:

• Def: $\forall x \in U: (xpx)$

• Gegenteil: $\exists x \in U: \neg(xpx)$

also `irreflexiv` :- `elem(X), not(rel(X,X)).`

und `reflexiv` :- `not(irreflexiv).`

▪ Betrachte transitiv:

• Def: $\forall x,y,z \in U: ((xpy \wedge ypz) \Rightarrow xpz)$

• Gegenteil: $\exists x,y,z \in U: ((xpy \wedge ypz) \not\Rightarrow xpz)$

• Umgeformt: $\exists x,y,z \in U: ((\underline{xpy} \wedge \underline{ypz}) \wedge \underline{\neg(xpz)})$

damit `intransitiv` :- `elem(X), elem(Y), elem(Z),`
`rel(X,Y), rel(Y,Z), not(rel(X,Z)).`

und `transitiv` :- `not(intransitiv).`



▪ Betrachte `antisymmetrisch`:

- Def: $\forall x,y \in U: ((xpy \wedge ypx) \Rightarrow x=y)$
- Gegenteil: $\exists x,y \in U: ((xpy \wedge ypx) \not\Rightarrow x=y)$
- Umgeformt: $\exists x,y \in U: ((xpy \wedge ypx) \wedge x \neq y)$
- Achtung: $\not\equiv \forall x,y \in M: xpy \Rightarrow ypx$!!! (\neg antisymmetrisch \neq symmetrisch)

also `aantisymmetrisch` :- `elem(X), elem(Y),
rel(X,Y), rel(Y,X), not(X=Y).`

und `antisymmetrisch` :- `not(aantisymmetrisch).`



▪ Und zusammengefasst:

`isHalbOrdnung` :- `reflexiv, transitiv, antisymmetrisch.`

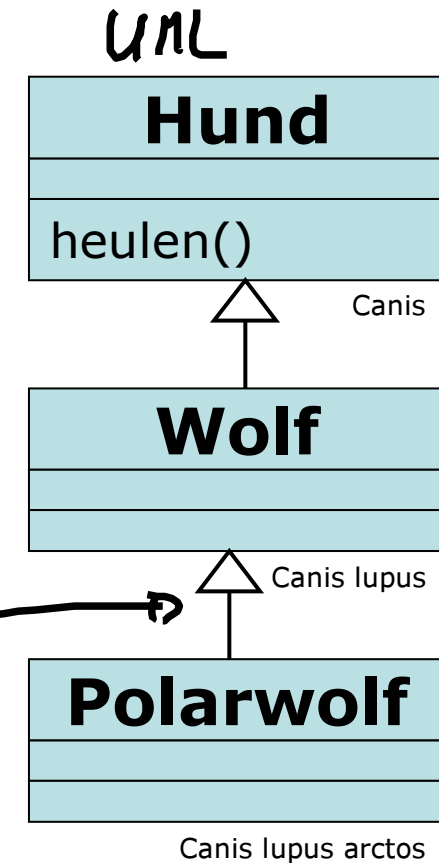


- Analogie Taxonomie: Der schwedische Naturforscher Carl von Linné (1707-1778) entwickelte die moderne Klassifikation der Lebewesen

Kategorie	Beispiel: Gartenhummel	
Reich	→	<i>Animalia</i> (Tiere)
Stamm	→	<i>Arthropoda</i> (Gliedertiere)
Klasse	→	<i>Hexapoda</i> (Insekten)
Ordnung	→	<i>Hymenoptera</i> (Hautflügler)
Familie	→	<i>Apidae</i> (Bienen)
Gattung	→	<i>Bombus</i> (Hummel)
Art	→	<i>B. hortorum</i> (Gartenhummel)

Feld-, Wald-, und Wiesenhummel



- Wozu?
 - **Signaturvererbung**: Es gibt bei den Untertypen die gleiche Funktion
 - **Implementierungsvererbung**: Diese Funktion funktioniert genau gleich
- Substitutionsprinzip: Verwende ein B als ob es ein A wäre...
 - B erbt von A
 - A ist allgemeiner als B
 - B ist spezieller als A
 - B hat (mindestens) alle Eigenschaften, die A hat
- Funktionen/Methoden mit gleichem Namen
 - **Überschreiben** (nur im Vererbungsfall): eine Methode, die bei einer Oberklasse A definiert wird, in einer Unterklasse B neu (spezieller) implementieren
 - **Überladen**: eine Funktion (meist Operator), die für mehrere Datentypen definiert ist
 - $5 \quad \underline{+} \quad 3 \quad = \quad 8$
 - „Hallo“ + „Welt“ = „HalloWelt“



- Bedeutung
 - Alle Objekte eines bestimmten Typs fasst man formal in einer **Klasse** zusammen
 - Ein konkretes Objekt (oft auch „Ausprägung“ genannt) einer Klasse nennt man **Instanz**
 - Klasse bezeichnet die grundsätzliche Idee/das Konzept des Gegenstandes und existiert unabhängig davon, ob eine Ausprägung existiert, oder nicht
 - Eine Instanz gehört immer zu einer bestimmten Klasse

- **Achtung:** In Haskell können nur Datentypen Instanzen sein, die Werttupel bezeichnet man nicht als Instanzen!!!
- Instanzen in Java, C++, C#, Eiffel und anderen objektorientierten Sprachen bezeichnen die Werttupel haben somit einen ganz anderen Charakter.
- Das Prinzip Klasse ↔ Instanz funktioniert aber überall gleich!



- Typklassen definieren eine abstrakte Schnittstelle

```
class Person a where  
  name      :: a -> String ← Signaturverbund  
  adresse   :: a -> String
```

- Vererbung fasst Schnittstellen zusammen

```
class (Person a) => NatuerlichePerson a where  
  vorname   :: a -> String  
  geboren   :: a -> Datum  
  vorname a = "Derderdenhebelnichtzieht" ← Impl.-Vererbung
```

- Konkrete Klassen erklären sich konform zur Schnittstelle

```
data Mensch = MachMensch String String Datum String  
instance Person Mensch where  
  name      (MachMensch x _ _ _) = x  
  adresse   (MachMensch _ _ _ x) = x  
instance NatuerlichePerson Mensch where  
  vorname   (MachMensch _ x _ _) = x  
  geboren   (MachMensch _ _ x _) = x
```



```
Main> name (MachMensch "Gelhausen" "Tom" "im Mai" "AGV")
"Gelhausen" :: [Char]
Main> vorname (MachMensch "Gelhausen" "Tom" "im Mai" "AGV")
"Tom" :: [Char]
Main> :type MachMensch "Gelhausen" "Tom" "im Mai" "AGV"
MachMensch "Gelhausen" "Tom" "im Mai" "AGV" :: Mensch
Main> :info Mensch
-- type constructor
data Mensch

-- constructors:
MachMensch :: String -> String -> Datum -> String -> Mensch

-- instances:
instance Person Mensch
instance NatuerlichePerson Mensch

Main>
```



- Mehrere Untertypen sind möglich

```
class (Person a) => JurPerson a where
  gruendung      :: a -> Datum
```

```
class (JurPerson a) => GbRPerson a where
  gesellschafter :: a -> Int -> Person
```

```
data GbR = MachGbR [Person] String Datum
```

Gesellschafter

Adresse

Gründung

Aber dummerweise geht das so nicht! ☹

```
Reading file "C:\...\test.hs":
```

```
ERROR "C:\...\test.hs":27 - Undefined type constructor "Person"
```

```
Prelude>
```

- Warum?

- Haskell ist **statisch typgebunden**: Der Typ eines Ausdrucks lässt sich ohne Programmausführung ermitteln → Kap. 5.4



▪ Typsynonyme (äquivalent zum alten Typ)

- `type Datum = String`
- `type Liste = []`

▪ „Algebraische“ Typdeklarationen

- `data Mensch = MachMenschen String Datum`
- `data Stapel t = MachStapel | Lege (Stapel t) t deriving Eq`
- `data Eq a => Menge a = Leer | MachMenge a (Menge a)`

▪ Typklassen: Klassendefinitionen

- `class Person a where`
`name :: a -> String`

▪ Typklassen: Instanzdefinitionen

- `instance Person Mensch where`
`name (MachMenschen x _ _ _) = x`

▪ Typklassen: Abgeleitete Instanzen („deriving“ ⇒ automatisch erzeugt!)

- `instance Eq a => Eq (Stapel a) where`
`MachStapel == MachStapel = True`
`Lege (u) v == Lege (x) y = v==y && u==x`
`_ == _ = False`



- Die lineare Suche ebenso wie viele andere elementare Algorithmen lässt sich auch mit dem Funktional `until` ausdrücken:

```
search w xs = (until p tail xs) /= []  
              where p xs = ([] == xs) || (w == head xs)
```

```
[ until p f x | p x P(x) = x  
  | otherwise = until p f (f x)
```

- `until p f x` wendet `f` wiederholt auf `x`, `(f x)`, `(f (f x))`, ... an, bis die Bedingung `p x` wahr wird

- in imperativen Sprachen entspricht dies der `while`-Schleife mit negierter Bedingung:

```
while not (p x) do x := f x
```

- `f` heißt daher der **Schleifenrumpf**, `p x` die **Abbruchbedingung**
- Gilt initial `p x`, so ist die Schleife leer (`f` null mal anwenden)
- Da `f` nicht nur auf `x`, sondern auch auf `(f x)`, `(f (f x))`, ... angewandt wird, muß die Vorbedingung von `f` (Zusicherung über `x`) gleich der Nachbedingung (Zusicherung über `f x`) sein. Diese Zusicherung heißt **Schleifeninvariante** der **until-Schleife**.



```
ggT a b = fst (until p gg' (a,b))
      where p (x,y)    = y==0
            gg' (x,y) = (y, x `mod` y)
```

- Korrektheit: siehe Vorlesung.
- Schleifenrumpf: gg'
- Abbruchbedingung: $p \text{ } gg' (a,b)$
- Schleifeninvariante:
 - Vorbedingung: $x=a$ und $y=b \Rightarrow \underline{ggT \ x \ y = ggT \ a \ b}$
 - Nachbedingung:
 - Sei $(m,n) :=$ Ergebnis von " $gg' (x,y)$ "
 - Dann gilt: $m=y$ und $n=(x \bmod y)$
 - Damit: $ggT \ m \ n$ $= ggT \ y \ (x \bmod y) = ggT \ y \ x = ggT \ x \ y = ggT \ a \ b$



```
bk n k | k==0      = 1
        | k==n     = 1
        | otherwise = (bk (n-1) (k-1)) + (bk (n-1) k)
```

- Vorbedingung:
 - n und k müssen positive, ganze Zahlen sein
 - n muß größergleich k sein
- Nachbedingung:
 - $bk\ n\ 0 = bk\ n\ n = 1$
 - $bk\ n\ k = bk\ (n-1)\ (k-1) + bk\ (n-1)\ k$
- Ist die Vorbedingung in der Nachbedingung enthalten?
 - Fall 1 (k=0): Abbruch der Iteration ✓
 - Fall 2 (k=n): Abbruch der Iteration ✓
 - Fall 3 (n>k>0):
 - 1. Aufruf: $(n-1) \geq (k-1) \geq 0$
 - 2. Aufruf: $(n-1) \geq k > 0$



```

binom n k | k==0           = 1
           | k==n         = 1
           | otherwise     = (binom (n-1) (k-1)) + (binom (n-1) k)
    
```

Korrektheit:

- **Partielle Korrektheit:** wenn ein Programm für *alle* Eingaben, die den Vorbedingungen genügen, ein korrektes Ergebnis liefert (falls es terminiert)
- Programm ist partiell korrekt: (Vergleich mit math. Definition)

$$\binom{n}{0} = \binom{n}{n} = 1$$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$



$$\begin{aligned} b_k \ n \ k \quad | \quad k==0 &= 1 \\ &| \quad k==n &= 1 \\ &| \quad \text{otherwise} &= (b_k \ (n-1) \ (k-1)) + (b_k \ (n-1) \ k) \end{aligned}$$

Korrektheit:

- **Totale Korrektheit:** es kann nachgewiesen werden, daß das Programm für alle Eingaben, die den Vorbed. genügen, terminiert.

☞ Definiere eine Funktion $\tau: \text{ParamTyp1} \times \dots \times \text{ParamTypN} \mapsto \mathbb{Z}$, zeige:

- τ ist streng monoton fallend
 - τ besitzt untere Schranke
- } wenn man in jedem Rekursionsschritt die aktuellen Aufrufparameter einsetzt

- Terminierungsfunktion z.B. $\tau(n,k) = n+k$
 - Streng monoton fallend, untere Schranke: 0
 - also ist das Programm total korrekt



- Sowie die Musterlösungen finden Sie unter

<http://www.infoeins.de/uebungsblaetter.php>

