

# Inhaltsverzeichnis

<b>1</b>	<b>Theoretische Grundlagen</b>	<b>2</b>
1.1	Äquivalenz . . . . .	2
1.1.1	$\alpha$ -Äquivalenz (Folie: 20_(166)) . . . . .	2
1.1.2	$\eta$ -Äquivalenz (Folie: 20_(167)) . . . . .	2
1.1.3	Church-Zahlen (Folie: 20_(179ff)) . . . . .	2
1.1.4	Rekursionsoperator Y (Folie: 20_(184)) . . . . .	3
1.2	Typinferenz . . . . .	3
<b>2</b>	<b>Parallelprogrammierung</b>	<b>4</b>
2.1	Grundlagen . . . . .	4
2.1.1	Coffman-Bedingungen (Deadlock-Bedingungen) (Folie: 54_(47)) . . . . .	4
2.1.2	Flynn's Taxonomy (Folie: 51_(13)); Grafiken von SS 14, Nr. 6 . . . . .	4
2.1.3	Beschleunigung: Amdahl's Law (Folie: 51_(17)) . . . . .	5
2.2	MPI . . . . .	5

# Kapitel 1

# Haskell

## 1.1 Allgemein

- Guards („|“ müssen weiter rechts als Funktionsname stehen)

## 1.2 Nützliche Funktionen

### 1.2.1 Listen

$(++) :: [a] \rightarrow [a] \rightarrow [a]$

Fügt zwei Listen zusammen. Falls die erste Liste nicht endlich lang ist, ist das Ergebnis die erste Liste.

Bsp.:

$$[x1, \dots, xm] ++ [y1, \dots, yn] == [x1, \dots, xm, y1, \dots, yn]$$

$$[x1, \dots, xm] ++ [y1, ..] == [x1, \dots, xm, y1, ..]$$

$head :: [a] \rightarrow a$

Gibt das erste Element einer nicht-leeren Liste zurück. Falls die Liste leer ist, wird eine Exception geworfen.

Bsp.:

$$head[1, 2, 3, 4] = 1$$

$last :: [a] \rightarrow a$

Gibt das letzte Element einer nicht-leeren, endlichen Liste zurück. Falls die Liste leer ist, wird eine Exception geworfen, ist die Liste nicht endlich terminiert die Funktion nicht.

Bsp.:

$$last[1, 2, 3] = 3$$

$(!!) :: [a] \rightarrow Int \rightarrow a$

„ $xs!!i$ “ gibt das  $i$ -te Element von Liste  $xs$  zurück. (Erstes Element hat Index 0)

Bsp.:

$$[0, 1, 2, 3]!!1 == 1$$

$tail :: [a] \rightarrow [a]$

Gibt alle Elemente einer nicht-leeren Liste außer dem ersten zurück. Falls die Liste leer ist, wird eine Exception geworfen.

Bsp.:

$$tail[1, 2, 3, 4] = [2, 3, 4]$$

$init :: [a] \rightarrow [a]$

$init\ x$  gibt die Liste aller Elemente von Liste  $x$  zurück, außer dem letzten Element von  $x$

Bsp.:

$$init[1, 2, 3, 4] == [1, 2, 3]$$

$null :: Foldable\ t \Rightarrow t\ a \rightarrow Bool$

Testet ob die Struktur (Liste) leer ist.

Bsp.:

$$null\ [1,2] == False$$

$$null\ [] == True$$

$length :: Foldable\ t \Rightarrow t\ a \rightarrow Int$

Bestimmt die Größe/Länge einer endlichen Liste, d.h. die Anzahl der Elemente.

Bsp.:

$$length\ [] == 0$$

$$length\ [1,2,3] == 3$$

$drop :: Int \rightarrow [a] \rightarrow [a]$

$drop\ n\ xs$  Gibt die Teilliste nach dem n-ten Element zurück, oder die leere Liste  $[]$ , wenn  $n > length\ xs$ :

Bsp.:

$$drop\ 6\ \text{„Hello World!“} == \text{„World!“}$$

$$drop\ 3\ [1,2,3,4,5] == [4,5]$$

$$drop\ 3\ [1,2] == []$$

$$drop\ 3\ [] == []$$

$$drop\ (-1)\ [1,2] == [1,2]$$

$$drop\ 0\ [1,2] == [1,2]$$

$takeWhile :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$

$takeWhile\ pred\ xs$  Gibt den längst-möglichen Prefix (möglicherweise leer) mit Elementen von Liste  $xs$  zurück, die das Prädikat  $pred$  erfüllen

Bsp.:

$$takeWhile\ (< 3)\ [1,2,3,4,1,2,3,4] == [1,2]$$

$$takeWhile\ (< 9)\ [1,2,3] == [1,2,3]$$

$$takeWhile\ (< 0)\ [1,2,3] == []$$

$dropWhile :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$

$dropWhile\ pred\ xs$  gibt den Suffix (möglicherweise leer) der Elemente von Liste  $xs$  zurück, ab dem ersten Element, das das Prädikat nicht erfüllt (erstes, nicht-erfüllendes Element inklusive)

$$dropWhile\ (< 3)\ [1,2,3,4,5,1,2,3] == [3,4,5,1,2,3]$$

$$dropWhile\ (< 9)\ [1,2,3] == []$$

$$dropWhile\ (< 0)\ [1,2,3] == [1,2,3]$$

$map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$

$map\ f\ xs$  wendet die Funktion „f“ auf die Elemente der Liste „xs“ an und gibt die Rückgabewerte als Liste zurück.

Bsp.:

$$map\ f\ [x1,x2,...,xn] == [f\ x1, f\ x2,..., f\ xn]$$

$$map\ f\ [x1,x2,..] == [f\ x1, f\ x2,..]$$

$reverse :: [a] \rightarrow [a]$

Gibt eine endliche Liste in umgekehrter Reihenfolge zurück.

$$reverse\ [1,2,3] == [3,2,1]$$

$foldr :: Foldable t \Rightarrow (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow t a \rightarrow b$  **oder:**

$foldr :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

$foldr\ op\ i\ xs$ : Wende Operation  $op$  auf Elemente der Liste  $xs$  an, verwende für erste Ausführung der Operation die Operanden  $i$  (links neben  $op$ , d.h. linker Operand) und  $last\ xs$  (rechts neben  $op$ , d.h. rechter Operand). Die restlichen Elemente der Liste werden rechts-geklammert verknüpft, d.h. das Ergebnis der vorherigen Operations-Ausführung wird als rechter Operand verwendet und das nächste Element der Liste als linker Operand.

**Die Elemente der Liste werden von Ende  $\rightarrow$  Anfang abgearbeitet.**

Bsp.:

$$foldr\ (+)\ 0\ [1, 2, 3, 4] == (1 + (2 + (3 + (4 + 0)))) == 10$$

Erklärung:

*Der Startwert 0 wird als rechter Operand in der innersten Klammer (ganz rechts) verwendet, der linke Operand ist das letzte Element der Liste. Das Ergebnis der innersten Klammer wird als rechter Operand für die nächste Operation verwendet, der linke Operand ist das zweit-letzte Element der Liste.*

$foldl :: Foldable t \Rightarrow (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow t b \rightarrow a$  **oder:**

$foldl :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$

$foldl\ op\ i\ xs$ : Wende Operation  $op$  auf Elemente der Liste  $xs$  an, verwende für erste Ausführung der Operation die Operanden  $i$  (links neben  $op$ , d.h. linker Operand) und  $head\ xs$  (rechts neben  $op$ , d.h. rechter Operand). Die restlichen Elemente der Liste werden links-geklammert verknüpft, d.h. das Ergebnis der vorherigen Operations-Ausführung wird als linker Operand verwendet und das nächste Element der Liste als rechter Operand.

**Die Elemente der Liste werden von Anfang  $\rightarrow$  Ende abgearbeitet.**

Bsp.:

$$foldl\ (+)\ 0\ [1, 2, 3, 4] == (((0 + 1) + 2) + 3) + 4 == 10$$

Erklärung:

*Der Startwert 0 wird als linker Operand in der innersten Klammer (ganz links) verwendet, der rechte Operand ist das erste Element der Liste. Das Ergebnis der innersten Klammer wird als linker Operand für die nächste Operation verwendet, der rechte Operand ist das zweite Element der Liste.*

$concat :: [[a]] \rightarrow [a]$

Nimmt eine Liste von Listen als Eingabe entgegen und hängt die einzelnen Listen aneinander, sodass eine große Liste entsteht. Die Reihenfolge der Elemente wird beibehalten und Elemente können mehrmals auftreten.

Bsp.:

$$concat\ [[1, 3], [2, 4, 3]] == [1, 3, 2, 4, 3]$$

$scanl :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow [b]$

(Ähnlich zu  $foldl$ )

$scanl\ op\ z\ L$ : Gibt Liste  $E$  zurück, mit

$$E_1 = z \quad // \text{erstes Element von Liste } E$$

$$\forall i \in \{2, \text{length}(xs)\} : E_i = (E_{i-1})\ op\ (L_{i-1})$$

Äquivalent:

$$last\ (scanl\ f\ z\ xs) == foldl\ f\ z\ xs$$

Bsp.:

$$scanl\ (+)\ 0\ [1, 1, 1] == [0, 0 + 1, (0 + 1) + 1, ((0 + 1) + 1)] == [0, 1, 2, 3]$$

$scanl1 :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow [a]$

Variante von  $scanl$ , ohne Startwert

Bsp.:

$$scanl1\ f\ [x1, x2, \dots] == [x1, x1\ 'f'\ x2, \dots]$$

$$scanl1\ (+)\ [1, 2, 3, 4] == scanl1\ (\backslash x\ y \rightarrow x + y)\ [1, 2, 3, 4] == [1, 3, 6, 10]$$

$scanr :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow [b]$

Analog zu *scanl*, beginnend am Anfang der Liste

Äquivalent:

$$head (scanr f z xs) == foldr f z xs$$

Bsp.:

$$scanr op i [x1, x2, x3, x4] == [i 'op' (x4)]$$

$$\begin{aligned} scanr (\backslash x y \rightarrow (x + y)/2) 2 [1, 2, 3, 4] &== [2.0, (2.0 + 4)/2, (3.0 + 3)/2, (3.0 + 2)/2, (2.5 + 1)/2] \\ &== [2.0, (6)/2, (6)/2, (5)/2, (3.5)/2] \\ &== [2.0, 3.0, 3.0, 3.5, 1.75] \end{aligned}$$

$scanr1 :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow [a]$

Variante von *scanr*, ohne Startwert

Bsp.:

$$scanr1 f [x1, x2, x3, x4] == [x4, x4 'f' x3, (x4 'f' x3) 'f' x2, ((x4 'f' x3) 'f' x2) 'f' x1]$$

$iterate :: (a \rightarrow a) \rightarrow a \rightarrow [a]$

*iterate f x* gibt die unendliche Liste zurück, die durch das wiederholte Anwenden der Funktion auf das vorherige Listenelement entsteht, das erste Listenelement der Liste ist *x*. Das heißt  $iterate f x == [x, f x, f(f x), \dots]$ .

Bsp.:

$$iterate (\backslash x \rightarrow x + 1) 0 == [0, 0 + 1, (0 + 1) + 1, ((0 + 1) + 1) + 1, \dots] == [0, 1, 2, 3, \dots]$$

$span :: (a \rightarrow Bool) \rightarrow [a] \rightarrow ([a], [a])$

*span pred xs* Liefert Tupel mit zwei Listen zurück, die erste Liste des Tupels ist der längst-mögliche Prefix (möglicherweise leer) der Liste *xs*, mit Elementen die das Prädikat *pred* erfüllen. Die zweite Liste des Tupels ist der Rest der Liste *xs*, ab dem Element, welches das Prädikat *pred* nicht mehr erfüllt. Ist die Liste *xs* leer, wird ein Tupel mit zwei leeren Listen zurückgegeben.

Äquivalent:

$$span p xs == (takeWhile p xs, dropWhile p xs)$$

Bsp.:

$$span (< 3) [1, 2, 3, 4, 1, 2, 3, 4] == ([1, 2], [3, 4, 1, 2, 3, 4])$$

$$span (< 3) [1, 2] == ([1, 2], [])$$

$$span (< 3) [3, 4] == ([], [3, 4])$$

$break :: (a \rightarrow Bool) \rightarrow [a] \rightarrow ([a], [a])$

*break pred xs* Liefert Tupel mit zwei Listen zurück, die erste Liste des Tupels ist der längst-mögliche Prefix (möglicherweise leer) der Liste *xs*, mit Elementen die das Prädikat *pred* nicht erfüllen. Die zweite Liste des Tupels ist der Rest der Liste *xs*, ab dem Element, welches das Prädikat *pred* erfüllt. Ist die Liste *xs* leer, wird ein Tupel mit zwei leeren Listen zurückgegeben.

Äquivalent:

$$break p xs == (takeWhile p xs, dropWhile p xs)$$

Bsp.:

$$break (> 3) [1, 2, 3, 4, 1, 2, 3, 4] == ([1, 2, 3], [4, 1, 2, 3, 4])$$

$$break (< 9) [1, 2, 3] == ([], [1, 2, 3])$$

$$break (> 3) [1, 2, 3] == ([1, 2, 3], [])$$

$filter :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$

Gibt einer Liste aller Elemente aus der Eingabeliste zurück, die das Eingabepredikat erfüllen. Die Reihenfolge der Elemente in der Ausgabeliste entspricht der Reihenfolge in der Eingabeliste, Elemente treten so oft auf, wie in der Eingabeliste.

Äquivalent:

$$filter\ pred\ xs == [x | x \leftarrow xs, pred\ x]$$

Bsp.:

$$filter\ (< 3)[1, 2, 3, 4, 1, 2] == [1, 2, 1, 2]$$

$zip :: [a] \rightarrow [b] \rightarrow [(a, b)]$

Verarbeitet zwei Listen zu einer Liste aus Tupeln mit zwei Elementen. Das erste Element des i-ten Tupels ist das i-te Element der ersten Eingabeliste, das zweite Element das i-te Element der zweiten Eingabeliste. Unterscheiden sich die Längen der Eingabelisten, entspricht die Länge der Ausgabeliste der kürzeren Eingabeliste. Die überschüssigen Elemente der längeren Liste werden nicht in die Ausgabeliste aufgenommen.

Bsp.:

$$zip\ [1, 2, 3]\ [4, 5, 6] == [(1, 4), (2, 5), (3, 6)]$$

$$zip\ [1, 2, 3]\ [4, 5, 6, 7, 8] == [(1, 4), (2, 5), (3, 6)] \quad // \text{Ausgabeliste gleich lang wie kürzere Eingabeliste}$$

$$zip\ []\ [4, 5, 6] == []$$

Analog:  $zip3, zip4, zip5, zip6, zip7$

Verarbeiten 3,4,5,6,7 Eingabelisten und geben eine Liste von Tupeln mit 3,4,5,6,7 Elementen zurück.

$zipWith :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$

$zipWith\ op\ xs\ ys$  verknüpft die i-ten Elemente von  $xs$  und  $ys$  mit der Operation  $op$  und gibt das Ergebnis an der i-ten Stelle in der Ausgabeliste zurück. Unterscheiden sich die Längen der Eingabelisten, entspricht die Länge der Ausgabeliste der kürzeren Eingabeliste. Die überschüssigen Elemente der längeren Elemente werden nicht verwendet.

Bsp.:

$$zipWith\ (+)\ [1, 2, 3]\ [4, 5, 6] == [5, 7, 9]$$

$$zipWith\ (+)\ [1, 2, 3]\ [4, 5, 6, 7, 8] == [5, 7, 9] \quad // \text{Ausgabeliste gleich lang wie kürzere Eingabeliste}$$

$$zipWith\ (+)\ []\ [4, 5, 6] == []$$

Analog:  $zipWith3, zipWith4, zipWith5, zipWith6, zipWith7$

Verarbeiten 3,4,5,6,7 Eingabelisten und geben eine Liste von Tupeln mit 3,4,5,6,7 Elementen zurück. Die Operation muss die entsprechende Anzahl an Parametern verwenden,  $zipWith3$  kann z.B. nicht mit der Funktion  $(+)$  verwendet werden, stattdessen muss  $(\backslash x\ y\ z \rightarrow x + y + z)$  verwendet werden.

$elem :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool$

$elem\ xs$  gibt „True“ zurück, wenn  $a$  in der Liste  $xs$  enthalten ist.

Bsp.:

$$elem\ 1[1, 2, 3, 4] == True$$

$$elem\ 1[2, 3, 4] == False$$

$sort :: Ord\ a \Rightarrow [a] \rightarrow [a]$

Sortiert Elemente vom kleinsten zum größten Element, Duplikate werden beibehalten in der Reihenfolge aus der Eingabe.

Bsp.:

$$sort[2, 3, 4, 1] == [1, 2, 3, 4]$$

$sortBy :: (a \rightarrow a \rightarrow Ordering) \rightarrow [a] \rightarrow [a]$

Sortiert Elemente vom kleinsten zum größten Element, unter Verwendung einer individuellen Vergleichsfunktion. Duplikate werden beibehalten in der Reihenfolge aus der Eingabe.

*subsequences* ::  $[a] \rightarrow [[a]]$

Gibt eine Liste mit allen Subsequenzen zurück.

Bsp.:

*subsequences* "abc" == [ "", "a", "b", "ab", "c", "ac", "bc", "abc" ]

*permutations* ::  $[a] \rightarrow [[a]]$

Gibt eine Liste mit allen Permutationen zurück.

*permutations* "abc" == [ "abc", "bac", "cba", "bca", "cab", "acb" ]

*inits* ::  $[a] \rightarrow [[a]]$

Gibt eine Liste aller Segmente der Eingabeliste zurück, das kürzeste Segment ist das erste Element der Ausgabeliste.

Bsp.:

*inits* [1, 3, 5] == [ [], [1], [1, 3], [1, 3, 5] ]

*inits* [1, 2, 3, 4, 5] == [ [], [1], [1, 2], [1, 2, 3], [1, 2, 3, 4], [1, 2, 3, 4, 5] ]

# Kapitel 2

## Theoretische Grundlagen

### 2.1 Äquivalenz

#### 2.1.1 $\alpha$ -Äquivalenz (Folie: 20\_(166))

gleicher Ausdruck / Funktion, nur andere Namen  $\Rightarrow$  durch Umbenennung Transformation von  $t_1$  zu  $t_2$  möglich

#### 2.1.2 $\eta$ -Äquivalenz (Folie: 20\_(167))

Zwei Funktionen sind gleich, falls Ergebnis gleich für alle Argumente

#### 2.1.3 Church-Zahlen (Folie: 20\_(179ff))

- Zahlen:

$$\begin{aligned}c_0 &= \lambda s. \lambda z. z \\c_1 &= \lambda s. \lambda z. s\ z \\c_2 &= \lambda s. \lambda z. s\ (s\ z) \\&\vdots \\c_n &= \lambda s. \lambda z. s^n\ z\end{aligned}$$

- Operationen

- Nachfolgerfunktion  $succ = \lambda n. \lambda s. \lambda z. s\ (n\ s\ z)$
- Addition  $plus = \lambda m. \lambda n. \lambda s. \lambda z. m\ s\ (n\ s\ z)$
- Multiplikation  $times = \lambda m. \lambda n. \lambda s. n\ (m\ s) \stackrel{\eta}{=} \lambda m. \lambda n. \lambda s. \lambda z. n\ (m\ s)\ z$
- Potenzieren  $exp = \lambda m. \lambda n. n\ m \stackrel{\eta}{=} \lambda m. \lambda n. \lambda s. \lambda z. n\ m\ s\ z$
- Subtraktion  $sub = \lambda m. \lambda n. n\ pred\ m$  (ÜB 5, Nr. 4)
  - \*  $pair = \lambda a. \lambda b. \lambda f. f\ a\ b$
  - \*  $fst = \lambda p. p(\lambda a. \lambda b. a)$
  - \*  $snd = \lambda p. p(\lambda a. \lambda b. b)$
  - \*  $next = \lambda p. pair\ (snd\ p)\ (succ\ (snd\ p))$
  - \*  $pred = \lambda n. fst\ (n\ next\ (pair\ c_0\ c_0))$
- Vergleichsoperation „lessEq  $m \leq n$ “  $lessEq = \lambda m. \lambda n. isZero\ (sub\ m\ n)$  (ÜB 6, Nr. 2)
- Vergleichsoperation „greaterEq  $m \geq n$ “  $greaterEq = \lambda m. \lambda n. isZero\ (sub\ n\ m)$
- Vergleichsoperation „eq  $m == n$ “  $eq = \lambda m. \lambda n. (\lambda a. a)\ (lessEq\ m\ n)\ (greaterEq\ m\ n)\ c_{false}$
- $isZero = \lambda n. n\ (\lambda p. c_{false})\ c_{true}$  (SS 14)

- boolesche Werte

- True  $c_{true} = \lambda t. \lambda f. t$
- False  $c_{false} = \lambda t. \lambda f. f$



### 2.1.4 Rekursionsoperator Y (Folie: 20\_(184))

$$Y = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

- Rekursionsoperator Y ist nicht typisierbar (Folie: 21\_(205))

## 2.2 Typinferenz

Bei Let-Polymorphismus angepasste Regeln von VAR und ABS (Folie: 22\_(211))

Regel	Constraint	Bsp.
$\text{CONST} \frac{c \in \text{Const}}{\Gamma \vdash c : \tau_c}$	$\text{CONST} \frac{c \in \text{Const}}{\Gamma \vdash c : \alpha_1} : \alpha_1 = \text{Typ}(c)$	$\text{CONST} \frac{1 \in \text{Const}}{\Gamma \vdash 1 : \alpha_1} : \alpha_1 = \text{Typ}(1) = \text{Int}$
$\text{VAR} \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$	$\text{VAR} \frac{(x : \alpha_1)(x) = \alpha_2}{(x : \alpha_1) \vdash x : \alpha_2} : \alpha_1 = \alpha_2$	
$\text{ABS} \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \rightarrow \tau_2}$	$\text{ABS} \frac{\Gamma, x : \alpha_2 \vdash t : \alpha_3}{\Gamma \vdash \lambda x.t : \alpha_1} : \alpha_1 = \alpha_2 \rightarrow \alpha_3$	
$\text{APP} \frac{\Gamma \vdash t_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau}$	$\text{APP} \frac{\Gamma \vdash t_1 : \alpha_2 \quad \Gamma \vdash t_2 : \alpha_3}{\Gamma \vdash t_1\ t_2 : \alpha_1} : \alpha_2 = \alpha_3 \rightarrow \alpha_1$	
$\text{LET} \frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : ta(\tau_1, \Gamma) \vdash t_2 : \tau_2}{\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : \tau_2}$	$\text{LET} \frac{\Gamma \vdash t_1 : \alpha_2 \quad \Gamma, x : ta(\alpha_2, \Gamma) \vdash t_2 : \alpha_3}{\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : \alpha_1} : \alpha_2 = \alpha_3$	

# Kapitel 3

## Parallelprogrammierung

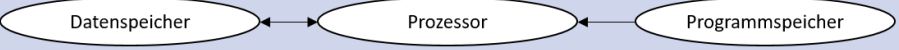
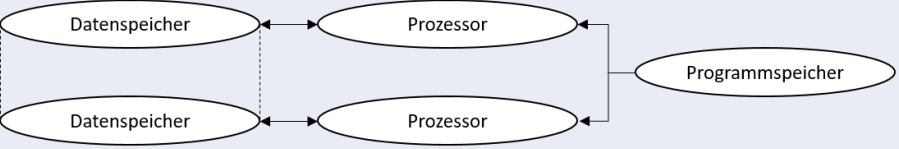
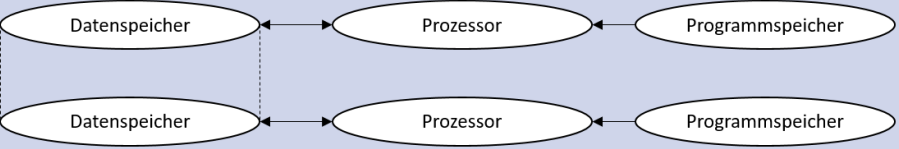
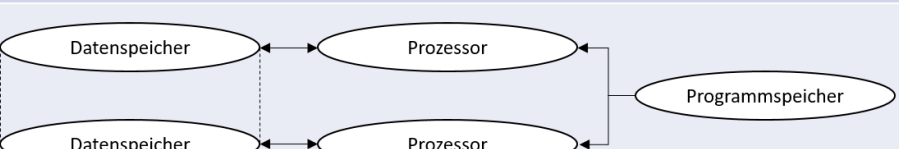
### 3.1 Grundlagen

#### 3.1.1 Coffman-Bedingungen (Deadlock-Bedingungen) (Folie: 54\_(47))

Wenn alle vier der folgenden Bedingungen zutreffen, liegt ein Deadlock vor. Deadlocks können verhindert werden, indem immer mindestens eine Bedingung nicht erfüllt ist, d.h. nicht alle auf einmal erfüllt sein können.

1. **Mutual exclusion**
  - beschränkter Zugriff auf eine Ressource
  - Ressource kann nur mit einer beschränkten Anzahl von Nutzern geteilt werden
2. **Hold and wait**: Warten auf alle benötigten Ressourcen, während die Kontrolle über bisher zugesprochene (mind. eine) Ressourcen behalten wird.
3. **No preemption**: Zugewiesene Ressourcen können nur freiwillig zurückgegeben werden, die Rückgabe kann nicht erzwungen werden.
4. **Circular Wait**: Möglichkeit von Kreisen in Ressourcen-Anfragen Graph:  
Zyklische Kette von Prozessen, die bereits Ressourcen (mind. eine) erhalten haben und gleichzeitig auf weitere Ressourcen warten, welche jeweils dem nächsten Prozess in der zirkulären Kette zugesprochen wurden.

#### 3.1.2 Flynn's Taxonomy (Folie: 51\_(13)); Grafiken von SS 14, Nr. 6

Taxonomie		Graphische Repräsentation
SISD (Single Instruction x Single Data)	von Neumann Architektur: eine Anweisungsreihenfolge auf einem Speicher ausgeführt	
SIMD (Single Instruction x Multiple Data)	Eine Anweisung auf homogene Daten angewendet (z.B. Array) z.B. Vektor Prozessoren von frühen Supercomputern	
MIMD (Multiple Instruction x Multiple Data)	versch. Prozessoren operieren auf versch. Daten z.B. aktuelle Mehrkernprozessoren	
MISD (Multiple Instruction x Single Data)	Mehrere Anweisungen gleichzeitig auf gleichen Daten ausgeführt z.B. redundante Architekturen oder Pipelines in modernen Prozessoren	

### 3.1.3 Beschleunigung: Amdahl's Law (Folie: 51\_(17))

Beschleunigung eines Algo durch die Verwendung von n Prozessoren:

$$S(n) = \frac{T_1}{T(n)} = \frac{\text{Ausführungszeit mit einem Prozessor}}{\text{Ausführungszeit mit n Prozessoren}}$$

maximale Beschleunigung durch parallele Ausführung mit n Prozessoren:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

p: parallelisierbarer prozentualer Anteil des Programms

## 3.2 MPI

- *MPI\_Comm\_rank*: Rank  $R_i$  des ausführenden Prozesses i
- *MPI\_Comm\_size*: Gesamtanzahl von Prozessen
- *MPI\_Comm\_WORLD*: Standard Communicator

Minimalbsp.: (Folie: 53\_(7))