

# Todo list

■ Grafiken korrekt? Vgl. SIMD und MISD . . . . .	14
--	----

# Inhaltsverzeichnis

0.1	Klausuren . . . . .	5
0.1.1	Haskell . . . . .	5
0.1.2	Prolog . . . . .	5
0.1.3	$\lambda$ -Kalkül . . . . .	5
0.1.4	Typinferenz . . . . .	5
0.1.5	Parallel Basics . . . . .	6
0.1.6	C . . . . .	6
0.1.7	MPI . . . . .	6
0.1.8	Compiler . . . . .	6
<b>1</b>	<b>Haskell</b>	<b>8</b>
<b>2</b>	<b>Theoretische Grundlagen</b>	<b>10</b>
2.1	Äquivalenz . . . . .	10
2.1.1	$\alpha$ -Äquivalenz (Folie: 20_(166)) . . . . .	10
2.1.2	$\eta$ -Äquivalenz (Folie: 20_(167)) . . . . .	10
2.1.3	Divergenz (Folie: 20_(182)) . . . . .	10
2.1.4	Rekursionsoperator (Folie: 20_(184)) . . . . .	10
2.1.5	Auswertungsstrategien . . . . .	10
2.1.6	Church-Zahlen (Folie: 20_(179ff)) . . . . .	10
2.1.7	Rekursionsoperator Y (Folie: 20_(184)) . . . . .	11
2.2	Typinferenz . . . . .	11
<b>3</b>	<b>Prolog</b>	<b>12</b>
3.1	Vergleich arithmetischer Ausdrücke . . . . .	12
3.2	Funktionen für Listen . . . . .	12
3.3	Sonstige . . . . .	13
<b>4</b>	<b>Parallelprogrammierung</b>	<b>14</b>
4.1	Grundlagen . . . . .	14
4.1.1	Coffman-Bedingungen (Deadlock-Bedingungen) (Folie: 54_(47)) . . . . .	14
4.1.2	Flynn's Taxonomy (Folie: 51_(13)) (Ref.: Grafiken von SS 14, Nr. 6) . . . . .	14
4.1.3	Beschleunigung: Amdahl's Law (Folie: 51_(17)) . . . . .	14
4.2	MPI . . . . .	15
4.3	Java . . . . .	16
4.3.1	Runnable . . . . .	16
4.3.2	Callable + Feature . . . . .	17
4.3.3	RecursiveAction / ForkJoinPool . . . . .	18
4.3.4	RecursiveTask / ForkJoinPool . . . . .	19
4.3.5	AkkaActor tell() . . . . .	19
4.3.6	AkkaActor ask() . . . . .	21
<b>5</b>	<b>Design by Contract</b>	<b>22</b>
5.1	JML . . . . .	22
5.1.1	Basic Syntax . . . . .	22
<b>6</b>	<b>Compiler</b>	<b>23</b>
6.1	Java-Bytecode The Java Virtual Machine Specification . . . . .	23
6.1.1	Präfixe / Suffixe . . . . .	23
6.1.2	Lesen / Schreiben von lokalen Variablen . . . . .	23
6.1.3	Lesen / Schreiben von Feldern . . . . .	24
6.1.4	Sprungbefehle . . . . .	24
6.1.5	Methodenaufrufe . . . . .	26

6.1.6	Objekterzeugung . . . . .	26
6.1.7	Arithmetische Berechnungen . . . . .	27

# Überblick

## Haskell

pythagoreisches Tripel (Laziness, Streams, unendl. Listen)	SS13
Listenverarbeitung, take, drop, split	SS13
Letztes Listenelement, inits, Lexikalische Analyse (Automat), maybe, map	SS14
Wörterbuch, Charlist	SS15
Mehrweg-Bäume	SS16
Semimagnetische Quadrate, Bäume (mit State)	SS17
Lauffängerkodierung, split, group, encode	WS12/13
Hamming-Zahlen, Streams	WS 13/14
Kombinatoren, approximation unendl. Liste	WS 13/14
Newton-Iteration, unendliche Liste, let...in,	WS 14/15
binäre Bäume), delete_min, merge	14/15
Potenzreihe, iterate	WS15/16
PowerSeries, Instanziierung vervollständigen	15/16

## Typinferenz

Standard	SS13
Natürliche Zahlen im $\lambda$ -Kalkül, Typherleitung, mgu, unifikation	SS14
Standard, mgu, let, abs, app, var	SS15
Unifikation, Typsysteme Standard	SS17
Standard, mgu, Typinferenz	SS16
Typisierung, polymorphe Typen	WS 13/14
Typinferenz, mgu, Fill-In	WS14/15
Typinferenz, mgu, Fill-In	WS15/16

## Prolog

Differenzlisten	SS13
Codeerzeugung, Abstrakte Syntaxbäume, "Taschenrechner/CPU"	SS14
Haus vom Nikolaus – Eulerpfade	SS15
nichtdeterministische endliche Automaten	SS16
Datenbank	SS16
freie Variablen Test	SS17
abvynth	WS12/13
reguläre Ausdrücke	WS 13/14
del, freie Variablen, ankreuzen, $\lambda$ als Terme	WS 14/15
Rucksack-Problem	WS 15/16

## Parallel Basics

Flynn's Taxonomie

SS14

## CBasics

C Deklarationen	SS13
C Deklarationen	SS14
C Deklarationen	WS12/13
C Deklarationen	WS13/14
Flynn's Taxonomi	WS13/14
Wissenstragen	WS 14/15
Beschleunigung berechnen	WS 14/15

## Actor Modell

Aktoren und Parallelisierbarkeit, Lastverteilung	SS17
--	------

## Java

Petri-Netze	SS17
(puffernder asynchroner Schreiber	SS16
ArrayList	SS17
Synchronisierung, Begründen anhand Code	WS 15/16
Semaphore	arg2

## Design by Contract

Selektieren verschiedener Optionen	SS17
------------------------------------	------

## Lambda

Church, next, pred	SS13
Natürliche Zahlen im $\lambda$ -Kalkül,	
$\beta$ -Reduktion	SS14
SKI, rechnen	SS15
Gleichheit auf Church-Zahlen	SS17
Zähler-Objekte im $\lambda$ -Kalkül,	
$\beta$ -Reduktion	SS16
$\lambda$ -Kalkül, Typinferenz, Church-Paare, Unifikation, Typisierung	WS12/13
Natürliche Zahlen im $\lambda$ -Kalkül,	
$\beta$ -Reduktion	WS 13/14
Grundlagen, anwendung, Reduktionen, ankreuzen	WS 14/15
church, pair, fst, snd, euclid	WS 15/16

## MPI

MPI_SUM, my_int_sum_reduce	SS13
Matrizenmultiplikation	SS14
Collective Operations, Beas, Allgather, Reduce, implementierung Gather	SS15
Erklärung von Code	SS17
C-Code in MPI umwandeln, Lückentext	SS16
Broadcast	WS 12/13
Allgather & Alltoall	WS 13/14
Fill-In	WS 14/15
MPI: Collective Operations, Ausführung, Zustände ausfüllen	WS 15/16

## Compiler

Haskell, Compiler, Rekursiver Abstieg Parser	SS13
Compiler, Grammatik, First1 und Follow1	SS14
Linksfaktorisierung SLL, abs, Syntaxbaum, Rekursiver Abstieg Parser	SS15
First1- und Follow1-Mengen, ava-Bytecode	SS17
Java-Bytecode	SS16
Haskell, Erzeugung von Java-Bytecode	WS 12/13
Syntaktische Analyse, First1 und Follow1	WS 12/13
Parseen einer einfachen Datenaustauschsprache, Parser-Prozeduren in Pseudo-Code	WS 13/14
Linksfaktorisierung SLL, Abstrakter Syntaxbaum, rekursiven Abstiegsparser	WS 14/15
Syntaktische Analyse, rekursiven Abstiegsparser	WS 15/16

## 0.1 Klausuren

### 0.1.1 Haskell

Bäume, mapTree, reduceTree / foldTree, treeSum, treeConcat, treeToList	WS 11 / 12
Hirsch-Index, Bäume, allTrees, trees_4_13	SS 12
Laufängenenkodierung, splitWhen, group, encode	WS 12 / 13 (Nr. 1)
Java-Bytecode Erzeugung	WS 12 / 13 (Nr. 8)
Laziness, Streams, pythagoreisches Tripel	SS 13 (Nr. 1)
splits, alle möglichen Teillisten	SS 13 (Nr. 4)
Hamming-Zahlen	WS 13 / 14 (Nr. 1)
Kombinatoren, unendliche Liste, Eulerverfahren, iterate, zipWith	WS 13 / 14 (Nr. 2)
last (Listen-Element), inits	SS 14 (Nr. 1)
lexikalische Analyse, Maybe, Automat, Zahlensystem	SS 14 (Nr. 2)
Newton-Iteration	WS 14 / 15 (Nr. 1)
binäre Bäume, deleteMin, merge Trees	WS 14 / 15 (Nr. 2)
splitWords (Word, NonWord), joinWords, capitalize	SS 15 (Nr. 1)

### 0.1.2 Prolog

Nichtdeterminismus, Wechselgeld, removeMoney, changeMoney	WS 11 / 12
Wolf, Ziege, Kohl	SS 12
Labyrinth	WS 12 / 13
splits, alle möglichen Teillisten (Nr. 4)	SS 13
Differenzlisten (Nr. 5)	SS 13
reguläre Ausdrücke (Nr. 5)	WS 13 / 14
Java-Bytecodeerzeugung (Nr. 5)	SS 14
del, delete, freie Variablen in Lambda-Ausdruck	WS 14 / 15 (Nr. 4)
Haus vom Nikolaus, Eulerpfade, swap (Kanten umdrehen)	SS 15 (Nr. 4)

### 0.1.3 $\lambda$ -Kalkül

Fixpunktkombinator, $\beta$ -Reduktion	WS 11 / 12
Kombinatoren, $\beta$ -Reduktion, S K I	SS 12
Church-Paare, fst, snd, map, allgemeinsten Typ	WS 12 / 13
Church-Paare, next, pred (Nr. 2)	SS 13
Church-Zahlen, isZero, pred, add (Nr. 3)	WS 13 / 14
Scott-Zahlen, isZero, pred, sub (Nr. 3)	SS 14
freie Variablen, Redex, call-by-name, Normalform, $\alpha$ - & $\eta$ -Äquivalenz	WS 14 / 15 (Nr. 3)
S K I, Normalform	SS 15 (Nr. 2)

### 0.1.4 Typinferenz

S K I, Let-Polymorphismus	WS 11 / 12
Unifikation, Prolog-Notation	SS 12
allgemeinster Typ, Typschema, let-Polymorphismus (Nr. 3)	SS 13
let-Polymorphismus, $\lambda$ -gebundene Variable $\rightarrow$ nicht polymorph (Nr. 4)	WS 13 / 14
Scott-Zahlen, allgemeinster Typ, mgu (Nr. 4)	SS 14
mgu, $\tau_{just}$ , let-Polymorphismus	WS 14 / 15 (Nr. 5)
mgu, Herleitungsbaum, let-Polymorphismus, $\lambda$ -gebundene Variable $\rightarrow$ nicht polymorph	SS 15 (Nr. 3)

### 0.1.5 Parallel Basics

Flynn's Taxonomie	WS 13 / 14 (Nr. 9), SS 14 (Nr. 6)
Vorteile, Risiken, Thread-Scheduling, Sperren, Race-Bedingung, Deadlock, blocking vs. non-blocking, synchron-Send	WS 14 / 15 (Nr. 6)
Beschleunigung berechnen, Amdahls Law	WS 14 / 15 (Nr. 6)

### 0.1.6 C

Zeiger-Arithmetik, Arrays	WS 11 / 12
Zeiger-Arithmetik, Arrays	SS 12
Precedence-Rule	WS 12 / 13
Precedence-Rule	SS 13 (Nr. 6)
	WS 13 / 14 (Nr. 6)
	SS 14 (Nr. 9)

### 0.1.7 MPI

### 0.1.8 Compiler

kollektive Operationen, Durchschnitt	WS 11 / 12 <b>Todo</b>
Matrix-Multiplikation	SS 12 <b>Todo</b> , SS 14 (Nr. 8)
Broadcast-Implementierung	WS 12 / 13 <b>Todo</b>
Reduce, Reduce-Implementierung	SS 13 (Nr. 7)
Allgather & Alltoall	WS 13 / 14 (Nr. 7)
Scatterv / Gatherv	SS 14 (Nr. 8)
Send, Recv, Finalize, Parameter-Beschreibungen	WS 14 / 15 (Nr. 9)
Gather-Implementierung	SS 15 (Nr. 5)

Syntaktische Analyse, abstrakte Syntax, rekursiver Abstiegsparser	WS 11 / 12 <b>Todo</b>
Syntaktische Analyse, abstrakte Syntax, rekursiver Abstiegsparser	SS 12 <b>Todo</b>
Haskell, Java-Bytecode Erzeugung	WS 12 / 13 <b>Todo</b>
First-& Follow-Mengen	WS 12 / 13 <b>Todo</b>
	SS 14 (Nr. 10)
Linksfaktorisierung, Haskell-Datentyp, Abstiegsparser	SS 13 (Nr. 9)
Abstiegsparser	WS 13 / 14 (Nr. 10)
Linksfaktorisierung, SLL(1), abstrakte Syntax, Klassenhierarchie, Abstiegsparser	WS 14 / 15 (Nr. 10)
Linksfaktorisierung, abstrakte Syntax, Abstiegsparser, abstrakter Syntaxbaum	SS 15 (Nr. 2)

# Kapitel 1: Haskell

- `equal:: „==“`
- `not equal:: „/=“`

*changeListAt :: (a -> a) -> Int -> [a] -> [a]*

(Ref.: SS 16, Nr. 1)

`changeListAt f i list`: Wendet auf das *i*-te Element (1. Element hat Index 0) der Liste „list“ die Funktion *f* an.

```
changeListAt f i [] = []
changeListAt f 0 (x:xs) = ((f x):xs)
changeListAt f i (x:xs) = x.(changeListAt f (i-1) xs)
```

*isPalindrom :: Eq a => [a] -> Bool*

`isPalindrom list` überprüft es sich bei der Liste „list“ um ein Palindrom handelt

```
isPalindrom xs = foldl (&&) True (zipWith (\x y -> x==y) xs (myRev1 xs))
```

*flatten :: [[a]] -> [a]*

Transform a list, possibly holding lists as elements into a ‘flat’ list by replacing each list with its elements (recursively).

```
data NestedList a = Elem a | List [NestedList a]

flatten (Elem a)      = [a]
flatten (List [])     = []
flatten (List (x:xs)) = (flatten x) ++ (flatten (List xs))
flatten (List [])     = []
```

*compress :: Eq a => [a] -> [a]*

Entferne aufeinanderfolgende Duplikate in einer Liste

```
compress []           = []
compress (x:xs) = x : (compress $ dropWhile (== x) xs)
```

*pack :: Eq a => [a] -> [[a]]*

Speichere aufeinanderfolgende Duplikate aus der Eingabeliste in Unterlisten. Falls sich Elemente aus der Eingabeliste wiederholen und nicht direkt aufeinanderfolgen, werden diese in separate Unterlisten gespeichert.

```
pack :: Eq a => [a] -> [[a]]
pack [] = []
pack (x:xs) = (x:(filter (==x) xs)):(pack $ filter (/=x) xs)
```

*duplicate :: [a] -> [a]*

Dupliziere die Elemente einer Liste

```
duplicate []          = []
duplicate (x:xs) = x x:(duplicate xs)
```

*replicate :: [a] -> Int -> [a]*

`replicate list n` repliziert die Elemente der Liste „list“ *n*-mal

```
repli xs n = concat [f x | x <- xs]
  where f x = take n (repeat x)
```



*dropEvery* ::  $[a] \rightarrow \text{Int} \rightarrow [a]$

**dropEvery** list n entfernt jedes n-te Element aus der Liste

`dropEvery [1,2,3,4] 2 == [1,3]`

```
dropEvery list count = helper list count count
  where
    helper [] _ _ = []
    helper (x:xs) count 1 = helper xs count count
    helper (x:xs) count n = x : (helper xs count (n - 1))
```

*rotate* ::  $[a] \rightarrow \text{Int} \rightarrow [a]$

**rotate** list n verschiebe die Elemente der Liste um n Stellen nach links

`rotate „abcdefgh“ 3 == „defghabc“`

`rotate „abcdefgh“ (-2) == „ghabcdef“`

```
rotate xs n = drop nn xs ++ take nn xs
  where nn = n `mod` length xs
```

# 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 Divergenz (Folie: 20\_(182))

Terme, die nicht zu einer Normalform auswerten, divergieren. Diese modellieren unendliche Ausführungen.

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

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

mit

$$f (Y f) \stackrel{\beta}{=} Y f$$

### 2.1.5 Auswertungsstrategien

- Volle  $\beta$ -Reduktion (Folie: 20\_(177)): Jeder Redex kann jederzeit reduziert werden
- Normalenreihenfolge (Folie: 20\_(177)): Immer der linkeste äußerste Redex (der Parameter zur Verfügung hat) wird reduziert
- Call-By-Name (Folie: 20\_(189)): Reduziere linkesten äußersten Redex, der nicht von einem  $\lambda$  umgeben ist  
 $\Rightarrow$  Reduziere Argumente erst, wenn benötigt
- Call-By-Value (Folie: 20\_(190)): Reduziere linkesten äußersten Redex, der nicht von einem  $\lambda$  umgeben ist und dessen Argument ein Wert (d.h. max. ausgewertet, darf auch ein Lambda-Ausdruck sein) ist  
 $\Rightarrow$  werte Argumente vor Funktionsaufruf aus, dann setze max. ausgewertete Argumente ein

### 2.1.6 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  $\text{succ} = \lambda n. \lambda s. \lambda z. s (n s z)$
  - Addition  $\text{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m s (n s z)$
  - Multiplikation  $\text{times} = \lambda m. \lambda n. \lambda s. n (m s) \stackrel{\eta}{=} \lambda m. \lambda n. \lambda s. \lambda z. n (m s) z$
  - Potenzieren  $\text{exp} = \lambda m. \lambda n. n m \stackrel{\eta}{=} \lambda m. \lambda n. \lambda s. \lambda z. n m s z$
  - Subtraktion  $\text{sub} = \lambda m. \lambda n. n \text{ pred } m$  (Ref.: ÜB 5, Nr. 4)
    - \*  $\text{pair} = \lambda a. \lambda b. \lambda f. f a b$
    - \*  $\text{fst} = \lambda p. p(\lambda a. \lambda b. a)$
    - \*  $\text{snd} = \lambda p. p(\lambda a. \lambda b. b)$
    - \*  $\text{next} = \lambda p. \text{pair} (\text{snd } p) (\text{succ} (\text{snd } p))$
    - \*  $\text{pred} = \lambda n. \text{fst} (n \text{ next } (\text{pair } c_0 c_0))$

- Vergleichsoperation „lessEq  $m \leq n$ “  $lessEq = \lambda m. \lambda n. isZero (sub\ m\ n)$  (Ref.: ÜB 6, Nr. 2)
- Vergleichsoperation „greaterEq  $m \geq n$ “  $greaterEq = \lambda m. \lambda n. isZero (sub\ n\ m)$
- Vergleichsoperation „eq  $m == n$ “  $eq = \lambda m. \lambda n. (\lambda a. a) (lessEq\ m\ n) (greaterEq\ m\ n) c_{false}$
- $isZero = \lambda n. n (\lambda p. c_{false}) c_{true}$  (Ref.: SS 14)
- boolsche Werte
  - True  $c_{true} = \lambda t. \lambda f. t$
  - False  $c_{false} = \lambda t. \lambda f. f$

### 2.1.7 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))

### Typsystem $\Gamma \vdash t : T$

$\Gamma \vdash t : \tau$  – im Typkontext  $\Gamma$  hat Term  $t$  Typ  $\tau$ .  
 $\Gamma$  ordnet freien Variablen  $x$  ihren Typ  $\Gamma(x)$  zu.

$$\begin{array}{ll} \text{CONST: } \frac{c \in \text{Const}}{\Gamma \vdash c : \tau_c} & \text{VAR: } \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \\ \text{ABS: } \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x. t : \tau_1 \rightarrow \tau_2} & \text{APP: } \frac{\Gamma \vdash t_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau} \end{array}$$

### Angepasste Regeln:

$$\begin{array}{l} \text{VAR: } \frac{\Gamma(x) = \tau' \quad \tau' \succeq \tau}{\Gamma \vdash x : \tau} \\ \text{ABS: } \frac{\Gamma, x : \tau_1 \vdash t : \tau_2 \quad \tau_1 \text{ kein Typschema}}{\Gamma \vdash \lambda x. t : \tau_1 \rightarrow \tau_2} \end{array}$$

### Let-Typregel

$$\text{LET: } \frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : ta(\tau_1, \Gamma) \vdash t_2 : \tau_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : \tau_2}$$

# Kapitel 3: Prolog

## 3.1 Vergleich arithmetischer Ausdrücke

- gleich: „:=“
- ungleich: „\=“
- kleiner: „<“
- kleiner-gleich: „=<“
- größer: „>“
- größer-gleich: „>=“

## 3.2 Funktionen für Listen

- member (Folie: 31\_(243)): Überprüfe ob Element in Liste enthalten

```
member(X,[X|T]).
member(X,[Y|T]):-member(X,T).
```

- append (Folie: 31\_(243)): Hänge eine Liste an eine andere

```
append([],L,L).
append([X|R],L,[X|T]):-append(R,L,T).
```

„Die Konkatenation von [] und L ist L. Wenn die Konkatenation von R und L die Liste T ergibt, dann ergibt die Konkatenation von [X—R] und L die Liste [X—T].“

- reverse (Folie: 31\_(245)):

```
reverse([],[]).
reverse([X|R],Y):-reverse(R,Y1), append(Y1,[X],Y).
```

effizienter:

```
reverse(X,Y):-reverse(X,[],Y).
reverse([],Y,Y).
reverse([X|R],A,Y):-reverse(R,[X|A],Y).
```

- Quicksort (Folie: 31\_(247)):

```
qsort([],[]).
qsort([X|R],Y):-split(X,R,R1,R2),
                qsort(R1,Y1),
                qsort(R2,Y2),
                append(Y1,[X|Y2],Y).
split(X,[],[],[]).
split(X,[H|T],[H|R],Y):-X>H, split(X,T,R,Y).
split(X,[H|T],R,[H|Y]):-X<=H, split(X,T,R,Y).
```

- Listenpermutation (Folie: 31\_(248)):

```
permute([],[]).
permute([X|R],P):-permute(R,P1), append(A,B,P1), append(A,[X|B],P).
```

- lösche alle Elemente X aus Liste (Ref.: Üb 8, Nr. 3) :

```
del([],_,[]).
del([X|T1],X,L2):-del(T1,X,L2).
del([Y|T1],X,[Y|T2]):-del(T1,X,T2), not(X=Y).
```

- Listenlänge (Ref.: WS 12/13, Nr. 3a)

```
length([],0).
<<<<<<< HEAD
length([_|R],NewLength):-length(R,Length), NewLength is Length +1.
```

- Alle möglichen Teillisten einer Liste (Ref.: Z-Üb 7, Nr.1.2)

```
splits(L, ([], L)).
splits([X|L], ([X|S], E)):-splits(L,(S,E)).
=====
length([_|R],NewLength):- length(R,Length), NewLength is Length +1.
```

- alle möglichen Zerlegungen (Anfang- und Endteil) einer Liste (Ref.: SS 13, Nr. 4b)

```
splits(L,([],L)).
splits([X|L],([X|S], E)):- splits(L,(S,E)).
```

- alle Teillisten einer Liste (Ref.: WS 15/16, Nr. 3a)

```
sublists([], []).
sublists([X|L], [X|L2]):-sublists(L,L2).
sublists([_|L], L2):-sublists(L,L2).
>>>>>> c54c54954143bcf252fd91a88d2b9c0b3874f131
```

- Test auf Duplikate:

```
noDuplicates([]).
noDuplicates([H|T]):-not(member(H,T)),noDuplicates(T).
```

- Entferne aufeinanderfolgende Duplikate:

```
removeDuplicates([], []).
removeDuplicates([H|[H|T]],L):- removeDuplicates([H|T],L).
removeDuplicates([H|T],[H|L]):- removeDuplicates(T,L).
```

- Entferne alle Duplikate (auch wenn doppelte Elemente nicht direkt hintereinander sind):

```
removeAllDuplicates([], []).
removeDuplicates([H|T],[H|L]):- deleteElem(T,H,L1), removeDuplicates(L1,L).
```

- letztes Element einer (nicht-leeren) Liste:

```
lastElem([X],X).
lastElem([H|T], R):-lastElem(T,R).
```

### 3.3 Sonstige

- atom(Term): True, falls Term mit einem Atom instanziiert ist (Folie: 32\_(272))
- atomic(Term): True, falls Term mit einem Atom instanziiert ist (Folie: 32\_(272))
- integer(Term): True, falls Term mit einem Integer instanziiert ist
- var(Term): True, falls Term aktuell eine freie Variable ist

# Kapitel 4: Parallelprogrammierung

## 4.1 Grundlagen

### 4.1.1 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.

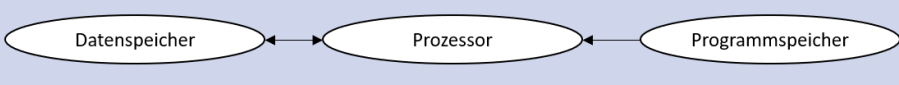
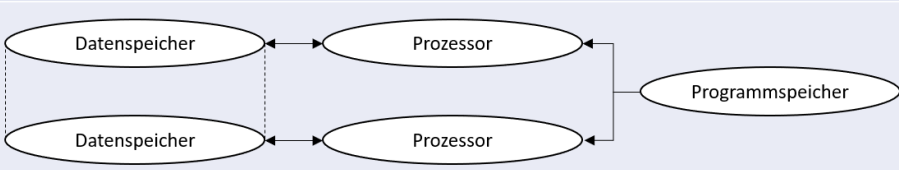
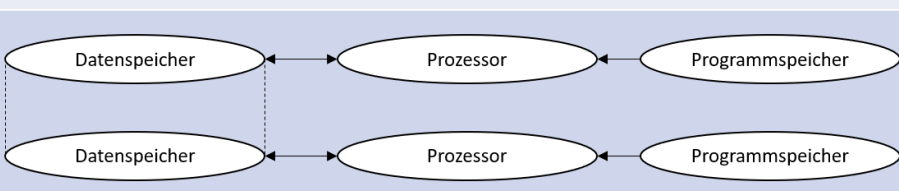
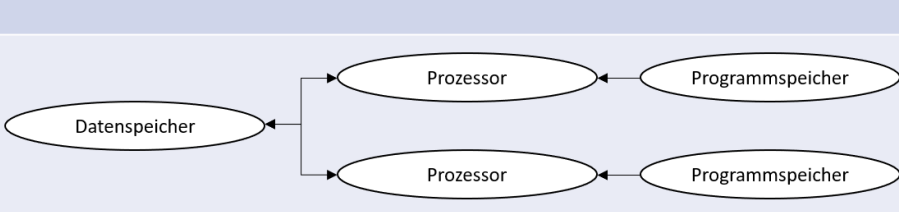
#### 3. No preemption: Zugewiesene Ressourcen können nur freiwillig zurückgegeben werden, die Rückgabe kann nicht erzwungen werden.

#### 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.

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

Grafiken korrekt? Vgl. SIMD und MISD

Taxonomie		Graphische Repräsentation
SISD (Single Instruction x Single Data)	von Neumann Architektur / Harvard 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 • Grafikprozessoren (GPUs)	
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 • Pipelines in modernen Prozessoren	

### 4.1.3 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

## 4.2 MPI

- MPI\_Comm\_WORLD: Standard Communicator
- MPI\_Init(&argc, &args): initialisiere MPI
- MPI\_Finalize(): Clean-Up nach Ausführung von MPI
- MPI\_Sendrecv\_replace: **Todo**
- MPI\_Sendrecv: **Todo**

## 4.3 Java

### 4.3.1 Runnable

```
public class DemoRunnable {
    public static void main(String[] args) {
        Thread thread0 = new Thread(new MyRunnable(0));
        Thread thread1 = new Thread(new MyRunnable(1));

        thread0.start();
        thread1.start();
    }

    public static class MyRunnable implements Runnable {
        private int id;

        public MyRunnable(int id) {
            this.id = id;
        }

        public void run() {
            System.out.println("Running: " + this.id);
        }
    }
}

/* Ausgabe:
Running: 0
Running: 1
*/
```



### 4.3.2 Callable + Feature

```

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class DemoCallableFuture {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(4);
        List<Future<String>> futures = new ArrayList<Future<String>>();

        for (int i = 0; i < 2; i++) {
            futures.add(executorService.submit(new MyCallable(i)));
        }

        for (Future<String> future : futures) {
            try {
                String result = future.get();
                System.out.println(result);
            } catch (InterruptedException e) {
                //TODO: handle exception
            } catch (ExecutionException e){
                //TODO: handle exception
            }
        }

        executorService.shutdown();
    }

    public static class MyCallable implements Callable<String> {
        private int id;

        public MyCallable(int id) {
            this.id = id;
        }

        public String call() {
            return "Running:␣" + this.id;
        }
    }
}

/* Ausgabe:
Running: 0
Running: 1
*/

```

### 4.3.3 RecursiveAction / ForkJoinPool

```

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class DemoRecursiveAction {
    public static void main(String[] args) {
        MyRecursiveAction action = new MyRecursiveAction(4, "none");
        ForkJoinPool pool = new ForkJoinPool();
        pool.invoke(action);
    }

    public static class MyRecursiveAction extends RecursiveAction {
        private static final long serialVersionUID = 1L;
        private int workload;
        private String side;

        public MyRecursiveAction(int workload, String side) {
            this.workload = workload;
            this.side = side;
        }

        @Override
        protected void compute() {
            if (this.workload > 1) {
                System.out.println("Workload_splitted_(" + this.side + "):_" + this.workload);

                int half = (int) this.workload / 2;
                MyRecursiveAction left = new MyRecursiveAction(half, "left");
                MyRecursiveAction right = new MyRecursiveAction(half, "right");

                left.fork();
                right.compute();
                left.join();
            } else {
                System.out.println("Workload_not_splitted_(" + this.side + "):_" + this.workload);
            }
        }
    }
}

/* Ausgabe:
Workload splitted (none): 4
Workload splitted (right): 2
Workload not splitted (right): 1
Workload not splitted (left): 1
Workload splitted (left): 2
Workload not splitted (right): 1
Workload not splitted (left): 1
*/

```

### 4.3.4 RecursiveTask / ForkJoinPool

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class DemoRecursiveTask {
    public static void main(String[] args) {
        MyRecursiveTask task = new MyRecursiveTask(4, "None");
        ForkJoinPool pool = new ForkJoinPool();
        Long result = pool.invoke(task);

        System.out.println("Final result: " + result);
    }

    public static class MyRecursiveTask extends RecursiveTask<Long> {
        private static final long serialVersionUID = 1L;
        private long workload;
        private String side;

        public MyRecursiveTask(long workload, String side) {
            this.workload = workload;
            this.side = side;
        }

        @Override
        protected Long compute() {
            if (this.workload > 1) {
                int half = (int) this.workload / 2;
                MyRecursiveTask left = new MyRecursiveTask(half, "Left");
                MyRecursiveTask right = new MyRecursiveTask(half, "Right");

                left.fork();
                Long rightResult = right.compute();
                Long leftResult = left.join();
                Long result = leftResult + rightResult;
                System.out.println(this.side + " result (splitted): " + result);
                return result;
            } else {
                System.out.println(this.side + " result (not splitted): " + this.workload);
                return this.workload;
            }
        }
    }
}
```

```
/* Ausgabe:
Right result (not splitted): 1
Left result (not splitted): 1
Right result (splitted): 2
Right result (not splitted): 1
Left result (not splitted): 1
Left result (splitted): 2
None result (splitted): 4
Final result: 4
*/
```

### 4.3.5 AkkaActor tell()

```
package DemoAkkaActor;
```

```

import akka.actor.*;

public class DemoAkkaActorTell {
    public static void main(String[] args) {
        ActorSystem actorSystem = ActorSystem.create("Demo");
        ActorRef zero = actorSystem.actorOf(Props.create(MyActor.class, 0));
        ActorRef one = actorSystem.actorOf(Props.create(MyActor.class, 1));

        zero.tell("Hello", one);
        one.tell("Hello", zero);
        actorSystem.terminate();
    }

    public static class MyActor extends AbstractActor {
        private int id;

        public MyActor(int id) {
            this.id = id;
        }

        @Override
        public Receive createReceive() {
            return receiveBuilder()
                .match(String.class, this::handleStringMessage)
                .matchAny(message -> unhandled(message))
                .build();
        }

        private void handleStringMessage(String message) {
            System.out.println("Id:␣" + this.id + "␣message:␣" + message);
            getSender().tell("Hi", ActorRef.noSender());
        }
    }
}

/* Ausgabe
Id: 1 message: Hello
Id: 0 message: Hello
Id: 0 message: Hi
Id: 1 message: Hi
*/

```

## 4.3.6 AkkaActor ask()

```

package DemoAkkaActor;

import akka.actor.*;
import akka.pattern.Patterns;
import akka.util.Timeout;
import scala.concurrent.Await;
import scala.concurrent.Future;

import java.util.concurrent.TimeUnit;

public class DemoAkkaActorAsk{
    public static void main(String[] args) {
        ActorSystem actorSystem = ActorSystem.create("Demo");
        ActorRef actor = actorSystem.actorOf(Props.create(MyActor.class));

        Integer[] values = new Integer[] {1,2};
        Timeout timeout = new Timeout(1, TimeUnit.SECONDS);
        Future<Object> future = Patterns.ask(actor, values, timeout);

        try {
            int result = (Integer) Await.result(future, timeout.duration());
            System.out.println("Result:␣" +result);
        } catch (Exception e) {
            e.printStackTrace();
        }

        actorSystem.terminate();
    }

    public static class MyActor extends AbstractActor {
        private int id;

        @Override
        public Receive createReceive() {
            return receiveBuilder()
                .match(Integer[].class, this::handleIntegerMessage)
                .matchAny(message -> unhandled(message))
                .build();
        }

        private void handleIntegerMessage(Integer[] message) {
            int sum = 0;
            for (int val : message){
                sum+=val;
            }
            getSender().tell(sum, getSelf());
        }
    }
}

/* Ausgabe:
Result: 3
*/

```

# Kapitel 5: Design by Contract

## 5.1 JML

### 5.1.1 Basic Syntax

Syntax	Bedeutung
$a ==> b$	a impliziert b
$a <==>$	a und b äquivalent
$a <!=>$	a und b <b>nicht</b> äquivalent ( $a \nleftrightarrow b$ )
<code>\result</code>	Ergebnis der Methode
<code>\old(E)</code>	Wert von E, bevor die Methode ausgeführt wurde
<code>(\forall \text{forall declaration; range-expression; body-expression})</code>	<code>(\forall \text{int } i; 0 \leq i \ \&amp;\&amp; \ i &lt; \text{size}; \text{\old(elements[i])} == \text{elements[i]})</code> für alle i zwischen 0 und „size“ gilt: <code>\old(elements[i]) == elements[i]</code>
<code>(\exists \text{exists declaration; range-expression; body-expression})</code>	<code>(\exists \text{int } i; 0 \leq i \ \&amp;\&amp; \ i &lt; \text{size}; \text{\old(elements[i])} == \text{elements[i]})</code> es gibt ein i zwischen 0 und „size“, für das gilt: <code>\old(elements[i]) == elements[i]</code>

# Kapitel 6: Compiler

## 6.1 Java-Bytecode The Java Virtual Machine Specification

Class-Datei disassembeln: javap

### 6.1.1 Präfixe / Suffixe

Präfix / Suffix	Operand Typ
i	integer
l	long
s	short
b	byte
c	character
f	float
d	double
a	reference

### 6.1.2 Lesen / Schreiben von lokalen Variablen

Befehl	Parameter	Beschreibung	Beispiel
iconst_x	$x \in \{0, 1, 2, 3, 4, 5, m1\}$	lädt die int-Konstante x m1 steht für Konstante „-1“	<b>iconst_1</b> lädt den int-Wert „1“ auf den Stack
<b>TYPE</b> load_x	x: Index der lokalen Variable $x \in \{0, 1, 2, 3\}$ (x gehört zum Befehl, es gibt pro x ein Befehl; d.h. x ist keine Variable)	lädt den Wert der Variable mit Typ „TYPE“ mit Index x auf den Stack	<b>iload_2</b> lade Wert von der Variable mit Index 2 und dem Typ „Integer“
<b>TYPE</b> store_x	x: Index der Variable $x \in \{0, 1, 2, 3\}$ (x gehört zum Befehl, es gibt pro x ein Befehl; d.h. x ist keine Variable)	speichert den obersten Wert auf dem Stack mit Typ „TYPE“ in Variable mit Index x	<b>istore_2</b> speichere den obersten Wert auf dem Stack vom Typ „Integer“ in Variable 2
<b>TYPE</b> load x	x: Index der lokalen Variable $0 \leq x \leq 255$ (x mit 1 Byte darstellbar)	lädt den Wert der Variable mit Typ „TYPE“ mit Index x auf den Stack	<b>iload 7</b> lade Wert von der Variable mit Index 7 und dem Typ „Integer“
<b>TYPE</b> store x	x: Index der lokalen Variable $0 \leq x \leq 255$ (x mit 1 Byte darstellbar)	speichert den obersten Wert auf dem Stack mit Typ „TYPE“ in Variable mit Index x	<b>istore 7</b> speichere den obersten Wert auf dem Stack vom Typ „Integer“ in Variable 7
bipush const	const: konstanter Wert, der auf den Stack geladen werden soll	Lade den geg. konstanten Wert auf den Stack	bipush 10 Lade den Wert 10 auf den Stack
ldc	Todo	Beschreibung	Beispiel

### Vergleich „iload x“ vs. „iload x“

- „iload x“ (mit Unterstrich):  $x \in \{0, 1, 2, 3\}$  ist ein (einziger) Befehl, ohne Parameter. „x“ ist schon im Opcode enthalten. Der Befehl besteht aus 1 Byte.
- „iload x“ (ohne Unterstrich):  $0 \leq x \leq 255$  ist ein Befehl, mit Parameter „x“. x ist nicht im Opcode enthalten. „iload x“ funktioniert mit allen Zahlen x, die in ein Byte passen.

Da die Befehle mit Unterstrich Platz sparen, werden sie von realen Compilern bevorzugt; vorausgesetzt x ist klein genug.

### 6.1.3 Lesen / Schreiben von Feldern

Befehl	Parameter	Beschreibung	Beispiel
putfield	Todo	Beschreibung	Beispiel
getfield	Todo	Beschreibung	Beispiel
putstatic	Todo	Beschreibung	Beispiel
getstatic	Todo	Beschreibung	Beispiel

### 6.1.4 Sprungbefehle

Befehl	Parameter	Beschreibung	Beispiel
ifle LABEL	LABEL: Label, zu dem gesprungen werden soll, falls die Bedingung erfüllt ist	Wenn der oberste Wert auf dem Stack <b>kleiner oder gleich 0</b> ist, dann springe zu dem geg. Label	ifle then springe zu Label „then“, (Folie: 73_(411))
ifge LABEL	LABEL: analog zu „ifle“	Wenn der oberste Wert auf dem Stack <b>größer oder gleich 0</b> ist, dann springe zu dem geg. Label	ifge then analog zu „ifle“
ifeq LABEL	LABEL: analog zu „ifle“	Wenn der oberste Wert auf dem Stack <b>gleich 0</b> ist, dann springe zu dem geg. Label	ifeq then analog zu „ifle“
ifne LABEL	LABEL: analog zu „ifle“	Wenn der oberste Wert auf dem Stack <b>nicht gleich 0</b> ist, dann springe zu dem geg. Label	ifne then analog zu „ifle“
ifgt LABEL	LABEL: analog zu „ifle“	Wenn der oberste Wert auf dem Stack <b>größer 0</b> ist, dann springe zu dem geg. Label	ifgt then analog zu „ifle“
iflt LABEL	LABEL: analog zu „ifle“	Wenn der oberste Wert auf dem Stack <b>kleiner 0</b> ist, dann springe zu dem geg. Label	iflt then analog zu „ifle“
ifnull LABEL	LABEL: analog zu „ifle“	Wenn der oberste Wert (/Referenz) auf dem Stack <b>NULL ist</b> , dann springe zu dem geg. Label	ifnull then analog zu „ifle“
ifnonnull LABEL	LABEL: analog zu „ifle“	Wenn der oberste Wert (/Referenz) auf dem Stack <b>nicht NULL ist</b> , dann springe zu dem geg. Label	ifnonnull then analog zu „ifle“



Befehl	Parameter	Beschreibung	Beispiel
if_icmpeq LABEL	LABEL: analog zu „ifle“	Wenn die beiden obersten Integer-Werte auf dem Stack <b>gleich</b> sind, springe zu dem geg. Label	if_icmpeq then springe zu then (Folie: 73_(426))
if_icmpne LABEL	LABEL: analog zu „ifle“	Wenn die beiden obersten Integer-Werte auf dem Stack <b>nicht gleich</b> sind, springe zu dem geg. Label	if_icmpne then analog zu „if_icmpeq“
if_icmpge LABEL	LABEL: analog zu „ifle“	Wenn der oberste Integer-Wert auf dem Stack <b>gleich oder größer</b> als der zweite ist, springe zu dem geg. Label	if_icmpge then analog zu „if_icmpeq“
if_icmpgt LABEL	LABEL: analog zu „ifle“	Wenn der oberste Integer-Wert auf dem Stack <b>größer</b> als der zweite ist, springe zu dem geg. Label	if_icmpgt then analog zu „if_icmpeq“
if_icmple LABEL	LABEL: analog zu „ifle“	Wenn der oberste Integer-Wert auf dem Stack <b>kleiner oder gleich</b> als der zweite ist, springe zu dem geg. Label	if_icmple then analog zu „if_icmpeq“
if_icmplt LABEL	LABEL: analog zu „ifle“	Wenn der oberste Integer-Wert auf dem Stack <b>kleiner</b> als der zweite ist, springe zu dem geg. Label	if_icmplt then analog zu „if_icmpeq“
goto LABEL	LABEL: analog zu „ifle“	Springe bedingungslos zu dem geg. Label	goto done springe zu Label „done“ (Folie: 73_(411))

### 6.1.5 Methodenaufrufe

Tabelle 6.1: Quelle

Befehl	Parameter	Beschreibung	Beispiel
invokevirtual #INDEX	INDEX: Index der aufzurufenden Methode	Rufe die <b>nicht statische, public oder protected</b> Methode mit dem geg. Index auf. Die Parameter der Methode werden automatisch in den Operandenstack geladen, <b>beginnend bei Variablen-Index 1</b> (Variablen-Index 0: this-Variable / Referenz)	invokevirtual #2 Rufe die Methode mit Index 2 auf (Folie: 73_(419))
invokestatic #INDEX	INDEX: analog zu „invokevirtual“	Rufe die <b>statische</b> Methode mit dem geg. Index auf. Die Parameter der Methode werden automatisch in den Operandenstack geladen, <b>beginnend bei Variablen-Index 0</b> (statische Methode, somit keine „this“-Variable)	invokestatic #2 analog zu „invokevirtual“
invokespecial #INDEX	INDEX: analog zu „invokevirtual“	Rufe die <b>private, Superklassen- oder Konstruktor-</b> Methode mit dem geg. Index auf. Die Parameter der Methode werden automatisch in den Operandenstack geladen, <b>beginnend bei Variablen-Index 1</b> (Variablen-Index 0: this-Variable / Referenz)	invokespecial #2 analog zu „invokevirtual“
invokeinterface #INDEX	INDEX: analog zu „invokevirtual“	Rufe eine <b>Interface</b> Methode auf, wobei Implementierung des aufrufenden Objektes verwendet wird	invokeinterface #2 analog zu „invokevirtual“
invokedynamic #INDEX	INDEX: analog zu „invokevirtual“	<b>Todo</b>	invokedynamic #2 analog zu „invokevirtual“

### 6.1.6 Objekterzeugung

Befehl	Parameter	Beschreibung	Beispiel
newarray TYPE	TYPE: Typ der Array-Werte	Erstelle ein Array mit Werten des geg. Typ und der Größe des obersten Stack-Wertes.	newarray int Erstelle ein int-Array

### 6.1.7 Arithmetische Berechnungen

Befehl	Parameter	Beschreibung	Beispiel
<b>TYPE</b> mul	-	multipliziert zwei Werte vom Typ „TYPE“ und lädt das Ergebnis als obersten Wert auf den Stack	<b>imul</b>
<b>TYPE</b> div	-	dividiert zwei Werte vom Typ „TYPE“ und lädt das Ergebnis als obersten Wert auf den Stack	<b>idiv</b>
<b>TYPE</b> add	-	addiert zwei Werte vom Typ „TYPE“ und lädt das Ergebnis als obersten Wert auf den Stack	<b>iadd</b>
<b>TYPE</b> sub	-	subtrahiert zwei Werte vom Typ „TYPE“ und lädt das Ergebnis als obersten Wert auf den Stack	<b>iadd</b>
<b>TYPE</b> neg	-	negiert einen Wert vom Typ „TYPE“ und lädt das Ergebnis als obersten Wert auf den Stack	<b>ineg</b>
iinc index, const	index: Index der zu inkrementierenden Variable const: Wert um den die Variable inkrementiert werden soll	Inkrementiere die Variable mit dem geg. Index um den geg. konstanten Wert	iinc 1, 1 inkrementiere die Variable mit Index 1 um 1 iinc 1, -1 inkrementiere die Variable mit Index 1 um (-1)