

# Programmierparadigmen Cheat Sheet

Linus Schöb (Vorlage von Darius Schefer, Max Schik)

## Contents

|                        |   |
|------------------------|---|
| Haskell                | 1 |
| Lambda Calculus        | 3 |
| Typen                  | 4 |
| Prolog                 | 4 |
| Unifikation            | 6 |
| Parallelprogrammierung | 7 |
| Java                   | 8 |
| Compiler               | 9 |

## Haskell

Referenzielle Transparenz: Im gleichen Gültigkeitsbereich bedeuten gleiche Ausdrücke stets das gleiche. Zwei verschiedene Ausdrücke, die zum gleichen Wert auswerten, können stets durch den anderen ersetzt werden, ohne die Bedeutung des Programms zu verändern.

### General Haskell stuff

```
-- type definitions are right associative
foo :: a -> b -> c -> d
foo :: (a -> (b -> (c -> d)))
-- function applications are left associative
foo a b c d
((((foo a) b) c) d)
```

```
-- pattern matching can use constructors and constants
head (x:x2:xs) = x
only [x] = x
first (Pair a b) = a
first (a, b) = a
response "hello" = "world"
```

```
-- alias for pattern matching
foo l@(x:xs) = l == (x:xs) -- returns true
```

```
-- guards
foo x y
| x > y = "bigger"
| x < y = "smaller"
| x == y = "equal"
| otherwise = "love"
```

```
-- case of does pattern matching
foo x = case x of
    [] -> "salad"
    [1] -> "apple"
    (420:1) -> "pear"
```

```
-- list comprehension
[foo x | x <- [1..420], x `mod` 2 == 0]

[0..5] == [0,1,2,3,4,5]
```

```
-- combine two functions
f :: a -> b
g :: b -> c
```

```
h :: a -> c
h = f . g
```

```
-- type alias
type Car = (String,Int)
```

```
-- data types
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
            deriving (Show)
```

```
-- defines interface
class Eq t where
    (==) :: t -> t -> Bool
    (/=) :: t -> t -> Bool
```

```

-- default implementation
x /= y = not $ x == y

class Coll c where
  contains :: (Ord t) =>
    (c t) -> t -> Bool

-- extends interface
class (Show t) => B t where

foo :: (B t) -> String

-- implement interface
instance Eq Bool where
  True == True = True
  False == False = True
  True == False = False
  False == True = False

```

## Idioms

```

-- backtracking
backtrack :: Conf -> [Conf]
backtrack conf
  | solution conf = [conf]
  | otherwise = concat (map backtrack (filter legal (successors conf)))

solutions = backtrack initial

-- accumulator
-- linear recursion = only one recursive branch per call
-- end recursion = linear recursion + nothing to do with the result after recursive call
-- end recursion makes things memory efficient
fak n = fakAcc n 1
  where fakAcc n acc = if (n==0) then acc else fakAcc (n-1) (n*acc)
-- end of where is determined by indentation!

```

## Important functions

See extra Cheat Sheet: <https://github.com/rudymatela/concise-cheat-sheets>

`foldr` can handle infinite lists (streams) if combinator does sometimes not depend on right rest. `foldl` cannot. Result is a list => probably want to use `foldr`

Additional functions (maybe handwrite on other cheatsheet):

```

-- in a list of type [(key, value)] returns first element where key matches given value
lookup :: Eq a => a -> [(a, b)] -> Maybe b
-- applies function until the predicate is true
until :: (a -> Bool) -> (a -> a) -> a -> a
-- returns true if the predicate is true for at least one element
any :: Foldable t => (a -> Bool) -> t a -> Bool
-- return true if the predicate is true for all elements
all :: Foldable t => (a -> Bool) -> t a -> Bool
-- reverse list
reverse = foldl (flip (:)) []

```

# Lambda Calculus

## General stuff

- Function application is left associative  $\lambda x. f \ x \ y = \lambda x. ((f \ x) \ y)$
- untyped lambda calculus is turing complete

## Primitive Operations

### Let

- let  $x = t_1$  in  $t_2$  wird zu  $(\lambda x. t_2) t_1$

## Church Numbers

- $c_0 = \lambda s. \lambda z. z$
- $c_1 = \lambda s. \lambda z. s \ z$
- $c_2 = \lambda s. \lambda z. s \ (s \ z)$
- $c_3 = \lambda s. \lambda z. s \ (s \ (s \ z))$
- etc...
- **Successor Function**
  - $succ \ c_2 = c_3$
  - $succ = \lambda n. \lambda s. \lambda z. s \ (n \ s \ z)$
- **Arithmetic Operations**
  - $plus = \lambda m. \lambda n. \lambda s. \lambda z. m \ s \ (n \ s \ z)$
  - $times = \lambda m. \lambda n. \lambda s. n \ (m \ s)$
  - $exp = \lambda m. \lambda n. n \ m$
- $isZero = \lambda n. n \ (\lambda x. c_{false}) \ c_{true}$

## Boolean Values

- $c_{true} = \lambda t. \lambda f. t$
- $c_{false} = \lambda t. \lambda f. f$
- $not = \lambda a. a \ c_{false} \ c_{true}$
- $and = \lambda a. \lambda b. a \ b \ a$
- $or = \lambda a. \lambda b. a \ a \ b$
- $xor = \lambda a. \lambda b. a \ (not \ b) \ b$
- $if = \lambda a. \lambda then. \lambda else. a \ then \ else$

## Equivalences

### $\alpha$ -equivalence (Renaming)

Two terms  $t_1$  and  $t_2$  are  $\alpha$ -equivalent  $t_1 \stackrel{\alpha}{=} t_2$  if  $t_1$  and  $t_2$  can be transformed into each other just by consistent (no collision) renaming of the bound variables. Example:  $\lambda x. x \stackrel{\alpha}{=} \lambda y. y$

### $\eta$ -equivalence (End Parameters)

Two terms  $\lambda x. f \ x$  and  $f$  are  $\eta$ -equivalent  $\lambda x. f \ x \stackrel{\eta}{=} f$  if  $x$  is not a free variable of  $f$ . Example:  $\lambda x. f \ a \ b \ x \stackrel{\eta}{=} f \ a \ b$

## Reductions

### $\beta$ -reduction

A  $\lambda$ -term of the shape  $(\lambda x. t_1) \ t_2$  is called a Redex. The  $\beta$ -reduction is the evaluation of a function application on a redex. (Don't forget to add parenthesis!)

$$(\lambda x. t_1) \ t_2 \Rightarrow t_1 [x \mapsto t_2]$$

A term that can no longer be reduced is called **Normal Form**. The Normal Form is unique. Terms that don't get reduced to Normal Form diverge (grow infinitely large). Example:  $(\lambda x. x \ x) \ (\lambda x. x \ x)$

**Full  $\beta$ -Reduction:** Every Redex can be reduced at any time.

**Normal Order:** The leftmost outer redex gets reduced.

**Call by Name (CBN):** Reduce the leftmost outer Redex *if* not surrounded by a lambda. Example:

$$\begin{aligned} & (\lambda y. (\lambda x. y \ (\lambda z. z) \ x)) \ ((\lambda x. x) \ (\lambda y. y)) \\ \Rightarrow & (\lambda x. ((\lambda x. x) \ (\lambda y. y)) \ (\lambda z. z) \ x) \neq \end{aligned}$$

**Call by Value (CBV):** Reduce the leftmost Redex *that is* not surrounded by a lambda and whose argument is a value. A value is a term that can not be further reduced. Example:

$$\begin{aligned} & (\lambda y. (\lambda x. y \ (\lambda z. z) \ x)) \ ((\lambda x. x) \ (\lambda y. y)) \\ \Rightarrow & (\lambda y. (\lambda x. y \ (\lambda z. z) \ x)) \ (\lambda y. y) \\ \Rightarrow & (\lambda x. (\lambda y. y) \ (\lambda z. z) \ x) \neq \end{aligned}$$

Call by Name and Call by Value may not reduce to the Normal Form! Call by Name terminates more often than Call by Value.

## Church-Rosser

The untyped  $\lambda$  is confluent: If  $t \stackrel{*}{\Rightarrow} t_1$  and  $t \stackrel{*}{\Rightarrow} t_2$  then there exists a  $t'$  with  $t_1 \stackrel{*}{\Rightarrow} t'$  and  $t_2 \stackrel{*}{\Rightarrow} t'$ .

## Recursion

Rekursive Funktion = Fixpunkt des Funktional

$Y = \lambda f. (\lambda x. f \ (x \ x)) \ (\lambda x. f \ (x \ x))$  is called the recursion operator.  $Y \ G$  is the fixpoint of  $G$ .

# Typen

## Regelsysteme

- Term  $\psi$  herleitbar: “ $\vdash \psi$ ”
- Frege'scher Schlussstrich: aus dem über dem Strich kann man das unter dem Strich herleiten
- Prädikatenlogik erster Stufe:

$$\begin{array}{c} \Delta I \frac{\vdash \psi \quad \vdash \varphi}{\vdash \psi \wedge \varphi} \quad \Delta E_1 \frac{\vdash \psi \wedge \varphi}{\vdash \psi} \quad \Delta E_2 \frac{\vdash \psi \wedge \varphi}{\vdash \varphi} \\ \\ \forall E \frac{\vdash \forall x. \varphi}{\vdash \varphi[x \mapsto \psi]} \quad \forall I \frac{\vdash \varphi[x \mapsto c] \quad \text{Variable } c \text{ kommt nicht in } \varphi \text{ vor}}{\vdash \forall x. \varphi} \\ \\ MP \frac{\vdash \psi \Rightarrow \varphi \quad \vdash \psi}{\vdash \varphi} \quad LEM \frac{}{\vdash \varphi \vee \neg \varphi} \end{array}$$

- Beweiskontext:  $\Gamma \vdash \phi$ 
  - $\phi$  unter Annahme von  $\Gamma$  herleitbar
  - Erleichtert Herleitung von  $\phi \Rightarrow \psi$
  - Assumption Introduction  $\frac{}{\Gamma, \phi \vdash \phi}$

## Typsysteme

- Einfache Typisierung
  - $\vdash (\lambda x. 2) : bool \rightarrow int$
  - $\vdash (\lambda x. 2) : int \rightarrow int$
  - $\vdash (\lambda f. 2) : (int \rightarrow int) \rightarrow int$
- Polymorphe Typen
  - $\vdash (\lambda x. 2) : \alpha \rightarrow int$  ( $\alpha$  ist implizit allquantifiziert)
- Nicht alle sicheren Programme sind typisierbar
  - Typisierbare  $\lambda$ -Terme (ohne **define**) haben Normalform  $\Rightarrow$  terminieren  $\Rightarrow$  sind nicht turing-mächtig
  - Typsystem nicht vollständig bzgl.  $\beta$ -Reduktion
    - \* insb. Selbstapplikation im Allgemeinen nicht typisierbar
    - \* damit auch nicht Y-Kombinator

## Regeln

- “ $\Gamma \vdash t : \tau$ ”: im Typkontext  $\Gamma$  hat Term  $t$  den Typ  $\tau$
- $\Gamma$  ordnet freien Variablen  $x$  ihren Typ  $\Gamma(x)$  zu

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

## Typschema

- “ $\forall \alpha_1. \dots \forall \alpha_n. \tau$ ” heißt *Typschema* (Kürzel  $\phi$ )
  - Es bindet freie Typvariablen  $\alpha_1, \dots, \alpha_n$  in  $\tau$
- VAR-Regel muss angepasst werden:

$$VAR \frac{\Gamma(x) = \phi \quad \phi \succeq \tau}{\Gamma \vdash x : \tau}$$

- Neu:

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

- $ta(\tau, \Gamma)$ : Typabstraktion
  - Alle freien Typvariablen von  $\tau$  quantifiziert, die nicht frei in Typannahmen von  $\Gamma$ , also alle “wirklich unbekannten” (auch global)

# Prolog

## Generelles Zeug

Prolog ist nicht vollständig da die nächste Regel deterministisch gewählt wird, daher können Endlosschleifen entstehen und keine Lösung gefunden werden obwohl sie existiert.

Kleingeschriebene Wörter sind Atome. Großbuchstaben sind Variablen. `_` ist Platzhalter-Variable.

Prädikat heißt deterministisch gdw. stets es auf höchstens eine Weise erfüllt werden kann.

*% Prolog erfüllt Teilziele von links nach rechts*

`foo(X) :- subgoal1(X), subgoal2(X), subgoal3(X).`

*% != Cut = alles links (inklusive Prädikat links von :-) ist nicht reerfüllbar.*

*% Arten von Cuts:*

*% - Blauer Cut: beeinflusst weder Programmlaufzeit, noch -verhalten*

*% - Grüner Cut: beeinflusst Laufzeit, aber nicht Verhalten*

*% - Roter Cut: beeinflusst das Programmverhalten (häufig: letzten Wächter unnötig machen)*

*% Faustregel: Cut kommt, wenn wir sicher im richtigen Zweig sind, Ergebnisse danach*

`foo(X, Y) :- operation_where_we_only_want_the_first_result(X, Z), !, Y = Z.`

*% Idiom: generate and test*

`foo(X, Y) :- generator(X, Y), tester(Y).`

*% z.B.:*

```

sqrt(X,Y) :- nat(Y),
            Y2 is Y*Y, Y3 is (Y+1)*(Y+1),
            Y2 =< X, X < Y3.
% Früher testen => effizienter

% Listen mit Cons:
[1,2,3] = [1|[2|[3|[]]]].
[1,2,3|[4,5,6,7]] = [1,2,3,4,5,6,7].

% === Arithmetik
% erstmal nur Terme:
2 - 1 \= 1.
% Auswerten mit "is":
N1 is N - 1.
% Arithmetische Vergleiche:
=:=, \=, <=<, >,>=

```

## Wichtige Funktionen

Built-In:

```

% member(X, L): X ist in Liste L (alle Richtungen)
member(X, [X|R]).
member(X, [Y|R]) :- member(X,R).

% append(A, B, C): C = A ++ B (alle Richtungen)
append([], L, L).
append([X|R], L, [X|T]) :- append(R, L, T).

% reverse(L, R): R ist Liste L rückwärts (alle Richtungen)
reverse([], []).
reverse([X|R], Y) :- reverse(R, Y1), append(Y1, [X], Y).

% N ist Länger der Liste L (alle Richtungen)
length(L, N).

% Prüft, ob Prädikat X erfüllbar ist (NICHT: findet Instanziierung, sodass X nicht erfüllt ist)
not(X) :- call(X),!,fail.
not(X).

% Prüft, dass nicht gleich (alle Richtungen)
dif(X, Y) :- when(?(X,Y), X \== Y)

% Meta
atom(X). % Prüft ob X ein Atom ist
integer(X). % Prüft ob X eine Zahl ist
atomic(X). % Prüft ob X ein Atom oder eine Zahl ist

```

Weiter:

```

% Prüft ob Permutation voneinander. Iteriert durch alle Permutationen bei Reerfüllung (alle Richtungen)
permute([], []).
permute([X|R], P) :- permute(R, P1), append(A, B, P1), append(A, [X|B], P).

% lookup(N, D, A) mit A uninstantiiert: A <- D[N] nachschauen
% lookup(N, D, A) mit A instantiiert: D[N] <- A setzen
lookup(N, [(N,A)|_], A1) :- !, A=A1.
lookup(N, [_|T], A) :- lookup(N, T, A).

```

# Unifikation

## Unifikator

- Gegeben: Menge  $C$  von Gleichungen über Terme
- Gesucht ist eine Substitution  $\sigma$ , die alle Gleichungen erfüllt: **Unifikator**
  - $\sigma$  unifiziert Gleichung " $\theta = \theta'$ ", falls  $\sigma\theta = \sigma\theta'$
  - $\sigma$  unifiziert  $C$ , falls  $\forall c \in C$  gilt:  $\sigma$  unifiziert  $c$
  - Schreibweise für Substitution:  $[Y \rightarrow f(a, b), D \rightarrow b, X \rightarrow g(b), Z \rightarrow b]$   
→ soll eigentlich outline von einem Dicken Pfeil sein.
- **most general unifier** ist der allgemeinste Unifikator (mit den wenigsten unnötigen Ersetzungen/Annahmen)
  - $\sigma$  ist mgu gdw.  $\forall$  Unifikator  $\gamma \exists$  Substitution  $\delta : \gamma = \delta \circ \sigma$

## Robinson-Unifikationsalgorithmus: **unify(C)** =

```
if C ==  $\emptyset$  then []
else let  $\{\theta_l = \theta_r\} \uplus C' = C$  in
  if  $\theta_l == \theta_r$  then unify(C')
  else if  $\theta_l == Y$  and  $Y \notin FV(\theta_r)$  then unify( $[Y \rightarrow \theta_r]C'$ )  $\circ [Y \rightarrow \theta_r]$ 
  else if  $\theta_r == Y$  and  $Y \notin FV(\theta_l)$  then unify( $[Y \rightarrow \theta_l]C'$ )  $\circ [Y \rightarrow \theta_l]$ 
  else if  $\theta_l == f(\theta_l^1, \dots, \theta_l^n)$  and  $\theta_r == f(\theta_r^1, \dots, \theta_r^n)$ 
    then unify( $C' \cup \{\theta_l^1 = \theta_r^1, \dots, \theta_l^n = \theta_r^n\}$ )
  else fail
```

**unify(C)** terminiert und gibt **mgu** für  $C$  zurück, falls  $C$  unifizierbar, ansonsten **fail**.

## Resolution

Resolutionsregel:

$$\frac{(\tau_1, \tau_2, \dots, \tau_n; \gamma) \text{ Terme plus Substitution; } \alpha : -\alpha_1, \dots, \alpha_k \text{ eine Regel; } \sigma \text{ mgu von } \alpha \text{ und } \gamma(\tau_1)}{(\alpha_1, \dots, \alpha_k, \tau_2, \dots, \tau_n; \sigma \circ \gamma)}$$

- $\gamma$ : bisherige Substitution (wird am Ende ausgegeben)
- $P \vdash \dots$  heißt herleitbar/abarbeitbar durch logisches Programm  $P$
- $P \models \dots$  heißt logische Konsequenz logisches Programm  $P$
- Resolutionsregel ist korrekt:  $P \vdash \tau_1, \dots, \tau_n \Rightarrow P \models \tau_1, \dots, \tau_n$
- Resolutionsregel ist vollständig:  $P \models \tau_1, \dots, \tau_n \Rightarrow P \vdash \tau_1, \dots, \tau_n$
- Prolog ist korrekt, aber nicht vollständig (wegen deterministischer Regelwahl)

| Name | Beschreibung   | Beispiel                                 |
|------|--|--|
| SISD | a single instruction stream operates on a single memory            | von Neumann Architektur                  |
| SIMD | one instruction is applied on homogeneous data (e.g. an array)     | vector processors of early supercomputer |
| MIMD | different processors operate on different data                     | multi-core processors                    |
| MISD | multiple instructions are executed simultaneously on the same data | redundant architectures                  |

## Parallelprogrammierung

**Uniform Memory access (UMA):** .

**Parallelismus:** Mindestens zwei Prozesse laufen gleichzeitig.

**Concurrency:** Mindestens zwei Prozesse machen Fortschritt.

**Amdahls' Law:**

$$S(n) = \frac{T(1)}{T(n)} = \frac{\text{execution time if processed by 1 processor}}{\text{execution time if processed by } n \text{ processors}} = \text{speedup}$$

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}} \text{ with } p = \text{parallelizable percentage of program}$$

**Data Parallelism:** Die gleiche Aufgabe wird parallel auf unterschiedlichen Daten ausgeführt.

**Task Parallelism:** Unterschiedliche Aufgaben werden auf den gleichen Daten ausgeführt.

## Flynn's Taxanomy

### MPI

```
// default communicator, i.e. the collection of all processes
MPI_Comm MPI_COMM_WORLD;
```

```
// returns the number of processing nodes
int MPI_Comm_size(MPI_Comm comm, int *size);
```

```
// returns the rank for the processing node, root node has rank 0
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

```
// initializes MPI
int MPI_Init(int *argc, char ***argv);
```

```
// Cleans up MPI (called in the end)
int MPI_Finalize();
```

```
// blocks until all processes have called it
int MPI_Barrier(MPI_Comm comm);
```

```
// blocking asynchronous send. blocks until message buffer can be reused, i.e. message has been received.
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

```
// blocking asynchronous receive. blocks until message is received in the buffer completely.
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

## Communication Modes

### Synchronous

No buffer, synchronization (both sides wait for each other)

### Buffered

Explicit buffering, no synchronization (no wait for each other)

### Ready

No buffer, no synchronization, matching receive must already be initiated

### Standard

May be buffered or not, can be synchronous (implementation dependent)

There is only one receive mode.

```
MPI_Send() // standard-mode blocking send
MPI_Bsend() // buffered-mode blocking send
MPI_Ssend() // synchronous-mode blocking send
MPI_Rsend() // ready-mode blocking send
```

```
// non-blocking send and receive operations
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Test(MPI_Request *r, int *flag, MPI_Status *s);
// blocking check
int MPI_Wait(MPI_Request *r, MPI_Status *s);
```

## Collective Operations

### MPI\_Bcast

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype t, int root, MPI_Comm comm);
```

$$\begin{bmatrix} A_0 \\ A_0 \\ A_0 \end{bmatrix} \xrightarrow{\text{Broadcast}} \begin{bmatrix} A_0 \\ A_0 \\ A_0 \end{bmatrix}$$

### MPI\_Scatter MPI\_Gather

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype datatype, void *recvbuf, int recvcount, MPI_Datatype datatype, MPI_Comm comm)
```

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype datatype, void *recvbuf, int recvcount, MPI_Datatype datatype, MPI_Comm comm)
```

$$\begin{bmatrix} A_0 & A_1 & A_2 \end{bmatrix} \xrightarrow[\text{gather}]{\text{scatter}} \begin{bmatrix} A_0 \\ A_1 \\ A_2 \end{bmatrix}$$

## MPI\_Allgather

```
int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

$$\begin{bmatrix} A_0 \\ B_0 \\ C_0 \end{bmatrix} \xrightarrow{\text{Allgather}} \begin{bmatrix} A_0 & B_0 & C_0 \\ A_0 & B_0 & C_0 \\ A_0 & B_0 & C_0 \end{bmatrix}$$

## MPI\_Alltoall

```
int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm)
```

$$\begin{bmatrix} A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \\ C_0 & C_1 & C_2 \end{bmatrix} \xrightarrow{\text{Alltoall}} \begin{bmatrix} A_0 & B_0 & C_0 \\ A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \end{bmatrix}$$

## MPI\_Reduce

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm);
```

$$\begin{bmatrix} A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \\ C_0 & C_1 & C_2 \end{bmatrix} \xrightarrow{\text{Reduce}} \begin{bmatrix} A_0 + B_0 + C_0 & A_1 + B_1 + C_1 & A_2 + B_2 + C_2 \\ A_0 + B_0 + C_0 & A_1 + B_1 + C_1 & A_2 + B_2 + C_2 \\ A_0 + B_0 + C_0 & A_1 + B_1 + C_1 & A_2 + B_2 + C_2 \end{bmatrix}$$

## MPI\_allreduce

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

$$\begin{bmatrix} A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \\ C_0 & C_1 & C_2 \end{bmatrix} \xrightarrow{\text{Allreduce}} \begin{bmatrix} A_0 + B_0 + C_0 & A_1 + B_1 + C_1 & A_2 + B_2 + C_2 \\ A_0 + B_0 + C_0 & A_1 + B_1 + C_1 & A_2 + B_2 + C_2 \\ A_0 + B_0 + C_0 & A_1 + B_1 + C_1 & A_2 + B_2 + C_2 \end{bmatrix}$$

## MPI\_Reduce\_scatter

```
int MPI_Reduce_scatter(const void *sendbuf, void *recvbuf, const int recvcounts[],
                       MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

$$\begin{bmatrix} A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \\ C_0 & C_1 & C_2 \end{bmatrix} \xrightarrow{\text{Reduce-scatter}} \begin{bmatrix} A_0 + B_0 + C_0 \\ A_1 + B_1 + C_1 \\ A_2 + B_2 + C_2 \end{bmatrix}$$

## MPI\_Scan

```
int MPI_Scan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
             MPI_Op op, MPI_Comm comm);
```

$$\begin{bmatrix} A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \\ C_0 & C_1 & C_2 \end{bmatrix} \xrightarrow{\text{Scan}} \begin{bmatrix} A_0 & A_1 & A_2 \\ A_0 + B_0 & A_1 + B_1 & A_2 + B_2 \\ A_0 + B_0 + C_0 & A_1 + B_1 + C_1 & A_2 + B_2 + C_2 \end{bmatrix}$$

## Java

### Multithreading

#### Race conditions

A race condition exists if the order in which threads execute their operations influences the result of the program.

### Mutual Exclusion

Mutual exclusion, sometimes called a critical section, only one thread is allowed to execute operations of a critical section. If one thread executes operations of a critical section, other threads will be blocked if they want to enter it as well.

```
// Synchronized block, someObject is used as the monitor
synchronized(someObject) {
    ...
}
```

```
// synchronized function
synchronized void foo() {
    ...
}
```

### Caching and code reordering

- cached variables can lead to inconsistency
- code can be reordered by the compiler

volatile ensures that changes to variables are immediately visible to all threads/processors.

- establishes a happens-before relationship
- values are not locally cached in a CPU cache
- no optimization by compiler

```
// declares a volatile variable
volatile int c = 420;
```

### Functional programming

```
// lambdas
int i, int j) -> i + j;
```

```
// functional interfaces
@FunctionalInterface
interface Predicate {
    boolean check(int value);
}
```

```
public int sum(List<Integer> values, Predicate predicate)
    ...
};
```

```
sum(values, i -> i > 5);
```

```
// method reference to static function
SomeClass::staticFunction;
// method reference to object function
someObject::function;
```

### Executors

- Executors abstract from thread creation.
- provides an execute method that accepts a Runnable
 

```
void execute(Runnable runnable);
```
- ExecutorService is an interface that provides further lifecycle management logic



```
Callable<Integer> myCallable = () -> { return currentValue; };
Future<Integer> myFuture = executorService.submit(myCallable);
// ensures \result == \old{top()};
// ensures true; // trivial constraint
@*/
Object pop() { ... }
```

## Streams

Provides functions like

- filter
- map, reduce
- collect
- findAny, findFirst
- min, max

Any Java collection can be treated as a stream by calling the `stream()` method

### Example

```
List<Person> personsInAuditorium = ...;
double average =
    personsInAuditorium
    .stream()
    .filter(Person::isStudent)
    .mapToInt(Person::getAge) // converts a regular Stream to IntStream
    .average()
    .getAsDouble();

// collector
R collect(
    Supplier<R> supplier,
    BiConsumer<R, ? super T> accumulator,
    BiConsumer<R, R> combiner // only used for parallel streams
);

personsInAuditorium.stream().collect(
    () -> 0,
    (currentSum, person) -> { currentSum += person.getAge(); },
    (leftSum, rightSum) -> { leftSum += rightSum; }
);

// parallel stream
someValues.parallelStream();
```

## Design by Contract

Form of a Hoare triple  $\{P\} C \{Q\}$

- $P$ : precondition  $\rightarrow$  specification what the supplier expects from the client
- $C$ : series of statements  $\rightarrow$  the method of body
- $Q$ : postcondition  $\rightarrow$  specification of what the client can expect from the supplier if the precondition is fulfilled
- client has to ensure that the precondition is fulfilled
- client can expect the postcondition to be fulfilled, if the precondition is
- **Non-Redundancy-Principle:** the body of a routine shall not test for the routine's precondition

```
/*@ requires size > 0;
   @ ensures size == \old(size) - 1;
```

## Liskov Substitution Principle

- preconditions must not be more restrictive than those of the overwritten method:  $\text{Precondition}_{\text{Super}} \Rightarrow \text{Precondition}_{\text{Sub}}$
- postcondition must be at least as restrictive as those of the overwritten methods:  $\text{Postcondition}_{\text{Sub}} \Rightarrow \text{Postcondition}_{\text{Super}}$

## Compiler

### Basics

- **Lexikalische Analyse:**
  - Eingabe: Sequenz von Zeichen
  - Aufgaben:
    - \* erkenne bedeutungstragende Zeichengruppen: Tokens
    - \* überspringe unwichtige Zeichen (Leerzeichen, Kommentare, ...)
    - \* bezeichne identifizieren und zusammenfassen in Stringtabelle
  - Ausgabe: Sequenz von Tokens
- **Syntaktische Analyse:**
  - Eingabe: Sequenz von Tokens
  - Aufgaben:
    - \* überprüfe, ob Eingabe zu kontextfreier Sprache gehört
    - \* erkenne hierarchische Struktur der Eingabe
  - Ausgabe: Abstrakter Syntaxbaum (AST)
- **Semantische Analyse:**
  - Eingabe: Syntax Baum
  - Aufgaben: kontextsensitive Analyse (syntaktische Analyse ist kontextfrei)
    - \* Namensanalyse: Beziehung zwischen Deklaration und Verwendung
    - \* Typanalyse: Bestimme und prüfe Typen von Variablen, Funktionen, ...
    - \* Konsistenzprüfung: Alle Einschränkungen der Programmiersprache eingehalten
  - Ausgabe: Attributierter Syntaxbaum
  - ungültige Programme werden spätestens in Semantischer Analyse abgelehnt
- **Codegenerierung:**
  - Eingabe: Attributierter Syntaxbaum oder Zwischen-code
  - Aufgaben: Erzeuge Code für Zielmaschine
  - Ausgabe: Program in Assembler oder Maschinencode

## Linksrekursion

- Linksrekursive kontextfreie Grammatiken sind für kein  $k$  SLL( $k$ ).
- Für jede kontextfreie Grammatik  $G$  mit linksrekursiven Produktionen gibt es eine kontextfreie Grammatik  $G'$  ohne Linksrekursion mit  $L(G) = L(G')$

# Grammatiken

## Java Bytecode

### General

```
; this list partly is stolen from some guy on discord, but I forgot which one  
; types
```

```
i -> int  
l -> long  
s -> short  
b -> byte  
c -> char  
f -> float  
d -> double  
a -> reference
```

```
; load constants on the stack  
aconst_null ; null object  
dconst_0 ; double 0  
dconst_1 ; double 1  
fconst_0 ... fconst_2 ; float 0 to 2  
iconst_0 ... iconst_5 ; integer 0 to 5
```

```
; push immediates  
bipush i ; push signed byte i on the stack  
sipush i ; push signed short i on the stack
```

```
; variables (X should be replaced by a type, for example i (integer))  
; there exists Xload_i for i in [0, 3] to save a few bytes  
Xload i ; load local variable i (is a number)  
Xstore i ; store local variable i
```

```
; return from function  
return ; void return  
Xreturn ; return value of type X
```

```
; conditional jumps  
if_icmpeq label ; jump if ints are equal  
if_icmpge label ; jump if first int is >=  
if_icmpgt label ; jump if first int is >  
if_icmple label ; jump if first int is <=  
if_icmplt label ; jump if first int is <
```

```
ifeq label ; jump if = zero  
ifge label ; jump if >= zero  
ifgt label ; jump if > zero  
iflt label ; jump if < zero  
ifle label ; jump if <= zero  
ifne label ; jump if != zero
```

```
ifnull label ; jump if null  
ifnonnull label ; jump if not null
```

```
; Arithmetic, always operates on stack  
; push left side first, then right side  
iinc var const ; increment variable var (number) by const (immediate)  
isub ; Integer subtraction, for stack top -> [1,2,3] it would be 2 - 1  
iadd ; Integer addition  
imul ; Integer multiplication  
idiv ; Integer division  
ineg ; negate int
```

```
ishl ; shift left (arith)  
ishr ; shift right (arith)
```

```
; Logic (für [i, l])  
iand ; Bitwise and  
ior ; Bitwise or  
ixor ; Bitwise or
```

```
; Method calls. Stack: [objref, arg1, arg2] <-  
invokevirtual #desc ; call method specified in desc  
invokespecial #desc ; call constructor  
invokeinterface #desc ; call method on interface  
invokestatic #desc ; call static method (no objref)
```

```
; Misc  
nop ; No operation
```

```
; Arrays  
newarray T ; new array of type T  
Xaload ; load type X from array [Stack: arr, index] <-  
Xastore ; store type X in array [Stack: arr, index, val] <-  
arraylength ; length of array
```

### Examples

#### Arithmetic

```
Java:  
void calc(int x, int y) {  
    int z = 4;  
    z = y * z + x;  
}
```

#### Bytecode:

```
iconst_4 // lege eine 4 auf den stack  
istore_3 // pop stack und speichere Wert in Variable 3 (z)  
iload_2 // lade Variable 2 (y) und lege sie auf den stack  
iload_3 // lade Variable 3 (z) und lege sie auf den stack  
imul // multipliziere die oberen zwei elemente und lege da  
iload_1 // lade Variable 1 (x) und lege sie auf den Stack  
iadd // addiere die oberen zwei Elemente und lege sie auf  
istore_1 // pop stack und speichere Wert in Variable 3 (z)
```

#### Loops

##### Java:

```
public int fib(int steps) {  
    int last0 = 1;  
    int last1 = 1;  
    while (--steps > 0) {  
        int t = last0 + last1;  
        last1 = last0;  
        last0 = t;  
    }  
}
```

##### Bytecode:

```
iconst_1 // put 1 on stack  
istore_2 // store top of stack in var 2  
iconst_1 // put 1 on stack  
istore_3 // store top of stack in var 3
```

```

loop_begin: // label
    iinc 1 -1 // increment var 1 by -1
    iload_1 // load var 1 and put on stack
    ifle after_loop // if top of stack <= 0, jump to after_loop
    iload_2 // put var 2 on stack
    iload_3 // put var 3 on stack
    iadd // add top two elements and put on stack
    istore 4 // store top of stack in var 4
    iload_2 // load var 2 and put on stack
    istore_3 // store top of stack in var 3
    iload 4 // load var 4 and put on stack
    istore_2 // store top of stack in var 2
    goto loop_begin // jump to loop_begin
after_loop: // label
    iload_2 // load var 2 and put on stack
    ireturn // return top of stack

```