Programmierparadigmen Cheat Sheet

Wintersemester 2023/24 | Linus Schöb (Vorlage von Darius Schefer, Max Schik)

Contents

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

```haskell
-- 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)
-- prefix notation takes precedence over infix notation
id' l = head l : tail l

-- 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:l) -> "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 interfac
class Eq t where
```

```haskell
  (==) :: 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

```haskell
-- 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):

```haskell
-- 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

α-equivalence (Renaming)

Two terms t_1 and t_2 are α-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: $\lambda x.x \stackrel{\alpha}{=} \lambda y.y$

η-equivalence (End Parameters)

Two terms $\lambda x.f\ x$ and f are η-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

β-reduction

A λ-term of the shape $(\lambda x.\, t_1)\ t_2$ is called a Redex. The β-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 β-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:

$$\begin{aligned}&(\lambda y.\ (\lambda x.\ y\ (\lambda z.\ z)\ x))\ ((\lambda x.\ x)\ (\lambda y.\ y))\\ \Rightarrow&(\lambda x.\ ((\lambda x.\ x)\ (\lambda y.\ y))\ (\lambda z.\ z)\ x) \not\Rightarrow\end{aligned}$$

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:

$$\begin{aligned}&(\lambda y.\ (\lambda x.\ y\ (\lambda z.\ z)\ x))\ ((\lambda x.\ x)\ (\lambda y.\ y))\\ \Rightarrow&(\lambda y.\ (\lambda x.\ y\ (\lambda z.\ z)\ x))\ (\lambda y.\ y)\\ \Rightarrow&(\lambda x.\ (\lambda y.\ y)\ (\lambda z.\ z)\ x) \not\Rightarrow\end{aligned}$$

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 λ 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 Funktionals

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

Typen

Regelsysteme

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

$$\wedge\mathsf{I}\ \frac{\vdash \psi \qquad \vdash \varphi}{\vdash \psi \wedge \varphi} \qquad \wedge\mathsf{E}_1\ \frac{\vdash \psi \wedge \varphi}{\vdash \psi} \qquad \wedge\mathsf{E}_2\ \frac{\vdash \psi \wedge \varphi}{\vdash \varphi}$$

$$\forall\mathsf{E}\ \frac{\vdash \forall x.\ \varphi}{\vdash \varphi[x \mapsto \psi]} \qquad \forall\mathsf{I}\ \frac{\vdash \varphi[x \mapsto c] \qquad \text{Variable } c \text{ kommt nicht in } \varphi \text{ vor}}{\vdash \forall x. \varphi}$$

$$\mathsf{MP}\ \frac{\vdash \psi \Rightarrow \varphi \qquad \vdash \psi}{\vdash \varphi} \qquad \mathsf{LEM}\ \frac{}{\vdash \varphi \vee \neg\varphi}$$

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

Typsysteme

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

Regeln

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

$$\textsc{Const}\ \frac{c \in Const}{\Gamma \vdash c : \tau_c} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.\, t : \tau_1 \to \tau_2}$$

$$\textsc{Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash t_1 : \tau_2 \to \tau \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau}$$

Typschema

- "$\forall\alpha_1 \ldots . \forall\alpha_n.\tau$" heißt *Typschema* (Kürzel ϕ)
  - Es bindet freie Typvariablen $\alpha_1, \ldots, \alpha_n$ in τ
- Var-Regel muss angepasst werden:

$$\textsc{Var}\ \frac{\Gamma(x) = \phi \qquad \phi \succeq \tau}{\Gamma \vdash x : \tau}$$

- Neu:

$$\textsc{Let}\ \frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma, x : ta\,(\tau_1, \Gamma) \vdash t_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : \tau_2}$$

- $ta\,(\tau, \Gamma)$: Typabstraktion
  - Alle freien Typvariablen von τ, die nicht frei in Typannahmen von Γ vorkommen, werden allquantifiziert; also alle "wirklich unbekannten" (auch global)

Typinferenz

Gegeben: Term t und Typannahmen Γ
Gesucht: Lösung (σ, τ), sodass $\sigma\Gamma \vdash t : \tau$

1. Erstelle Herleitungsbaum anhand syntaktischer Struktur und Typisierungsregeln; dabei:
   - verwende zunächst überall rechts von : frische Typvariablen α_i
   - extrahiere Gleichungssystem C für die α_i gemäß Regeln
2. Bestimme mgu σ von C
3. Lösung: $(\sigma, \sigma(\alpha_1))$, wobei α_1 erste Typvariablen für t

Bei Let:

1. Sammle Gleichungen aus linkem Teilbaum in C_{let}
2. Berechne σ_{let} von C_{let}
3. $\Gamma' := \sigma_{let}, x : ta(\sigma_{let}(\alpha_i), \sigma_{let}(\Gamma))$
4. Benutze Γ' im rechten Teilbaum

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. es stets 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
```

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

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

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 rückwerts  (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:

```prolog
% 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 uninstanziiert: A <- D[N] nachschauen
% lookup(N, D, A) mit A instanziiert: 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 σ, die alle Gleichungen erfüllt: **Unifikator**
  - σ unifiziert Gleichung "$\theta = \theta'$", falls $\sigma\theta = \sigma\theta'$
  - σ unifiziert C, falls $\forall c \in C$ gilt: σ unifiziert c
  - Schreibweise für Substitution: $[Y \to f(a,b), D \to b, X \to g(b), Z \to b]$
    $\to$ soll eigentlich outline von einem Dicken Pfeil sein.
- **most general unifier** ist der allgemeinste Unifikator (mit den wenigsten unnötigen Ersetzungen/Annahmen)
  - σ ist mgu gdw. $\forall$ Unifikator γ $\exists$ 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` θ_l `==` θ_r `then unify(C')`

  `else if` θ_l `== Y and Y` $\notin$ `FV(`θ_r`) then unify([Y` $\to$ θ_r`]C')` $\circ$ `[Y` $\to$ θ_r`]`

  `else if` θ_r `== Y and Y` $\notin$ `FV(`θ_l`) then unify([Y` $\to$ θ_l`]C')` $\circ$ `[Y` $\to$ θ_l`]`

  `else if` θ_l `== f(`$\theta_l^1, ..., \theta_l^n$`) and` θ_r `== f(`$\theta_r^1, ..., \theta_r^n$`)`

    `then unify(C'` $\cup \{\theta_l^1 = \theta_r^1, ..., \theta_l^n = \theta_r^n\}$`)`

  `else fail`

Intuitiv: - Nimm immer irgendeine Gleichung - schon gleich $\Rightarrow$ ignorieren - eine Seite Variable $\Rightarrow$ substituiere sie durch andere Seite - beide Seiten gleiches, gleichstelliges Wurzelatom $\Rightarrow$ Argumente gleichsetzen

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

Resolution

Resolutionsregel:

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

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

Parallelprogrammierung

Uniform Memory access (UMA): .

Parallelismus: Mindestens zwei Prozesse laufen gleichzeitig.

Concurrency: Mindestens zwei Prozesse machen Fortschritt.

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

Amdahls' Law: $S(n) = \frac{1}{(1-p)+\frac{p}{n}}$ with $p =$ 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 Taxonomy

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, Pipelines

C/C++

Deklaration vor Verwendung (insbesondere bei Hilfsfunktionen!). Deklaration != Definition.

```
// Arrays vs Pointers
int arr[] = { 45, 67, 89 };
int *p;
p = arr; // p is assigned the address of the first element of arr
p = &arr[0]; // same effect as above
// p and arr are almost always interchangeable, except when using with extern and lying in one declaration
int matrix[4][4];

// always fetch from main memory; no registers, no optimization
volatile int x;
```

MPI

Meta

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

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

// get 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);
// usage in main:
MPI_Init(&argc, &args);

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

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

Main Operations

```
// send communication modes
MPI_Send();  // standard: implementation dependent (maybe heuristic optimizing)
MPI_Bsend(); // buffered: forced buffering, no synchronization
MPI_Ssend(); // synchronous: no buffer, forced synchronization (both sides wait for each other)
MPI_Rsend(); // ready: no buffer, no synchronization -> matching receive must already be initiated
// only one receive mode -> matches all sends

// parameters are the same for all communication modes:
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
                                   MPI_INT, MPI_LONG_LONG_INT, MPI_CHAR  // data types
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status);
         MPI_ANY_SOURCE  MPI_ANY_TAG   // wildcards

// simultaneous send and receive
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status)
// use the same buffer for send and receive (in a OUT-IN fashion)
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, int dest, int sendtag,
                         int source, int recvtag,
                         MPI_Comm comm, MPI_Status *status)

// blocking operations block until the buffer can be (re)used
// orthogonally, all operations have a non-blocking/immediate counterpart:
int MPI_Isend(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);

// send and receive operations can be checked for completion (flag == 1 iff completed)
int MPI_Test(MPI_Request* r, int* flag, MPI_Status* s);
// blocking check
int MPI_Wait(MPI_Request* r, MPI_Status* s);
```

There is no global order on communication events, but events within the same sender-receiver pair stay in order.

Collective Operations

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

$$\begin{bmatrix} A_0 & A_1 & A_2 \\ & & \\ & & \end{bmatrix} \overset{Broadcast}{\rightarrow} \begin{bmatrix} A_0 & A_1 & A_2 \\ A_0 & A_1 & A_2 \\ A_0 & A_1 & A_2 \end{bmatrix}$$

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

int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

int MPI_Scatterv(void* sendbuf, int* sendcounts, int* displacements, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Usually: `sendcount` = `recvcount` (here 1)
`sendcounts[i]` = how many elements to send to proc i

`displacements[i]` = first element to send to proc i (no overlap allowed, but gaps

$$\begin{bmatrix} A_0 & A_1 & A_2 \\ & & \\ & & \end{bmatrix} \underset{gather}{\overset{scatter}{\rightleftarrows}} \begin{bmatrix} A_0 & & \\ A_1 & & \\ A_2 & & \end{bmatrix}$$

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

$$\begin{bmatrix} A_0 & & \\ B_0 & & \\ C_0 & & \end{bmatrix} \overset{Allgather}{\rightarrow} \begin{bmatrix} A_0 & B_0 & C_0 \\ A_0 & B_0 & C_0 \\ A_0 & B_0 & C_0 \end{bmatrix}$$

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

(like matrix transposing)

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

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

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

`op` can be:

- Logical: `MPI_LAND`, `MPI_BAND`, `MPI_LOR`, `MPI_BOR`, ...
- Arithmetic: `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`, ...
- Arg (get causing rank): `MPI_MINLOC`, `MPI_MAXLOC`

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

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

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

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

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

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

Java

Functional programming

```java
@FunctionalInterface
interface Predicate {
  boolean check(int value);
}
```

```java
// lambda
(int i, int j) -> i + j;
// method reference to static function
SomeClass::staticFunction;
// method reference to object function
someObject::function;
```

Streams

Use `Collection.stream()` or `Collection.parallelStream()` to obtain a stream.
Methods: `filter`, `map`, `mapToInt`, `reduce`, `findAny`, `findFirst`, `min`, `max`, `average`, `limit`, `skip`, `distinct`, `sorted`, `toList`

```java
personsInAuditorum.stream().collect(
  () -> 0, // supplier of neutral value
  (currentSum, person) -> { currentSum += person.getAge(); } // accumulator of acc and elem
  (leftSum, rightSum) -> { leftSum += rightSum; } // combiner of multiple accs (for parallel)
);
```

Multithreading

Race conditions

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

Mutual Exclusion

A code section that only one thread is allowed to execute at a time is called 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. ⇒ only one thread per monitor is allowed to be in the section, but it may be multiple times (recursion). Test using `Thread.holdsLock(Object obj)`.

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

```java
// synchronized method, this is used as monitor
synchronized void foo() {
  ...
}
```

`wait` and `notify`

```java
// put this thread to sleep (always use in while loop!)
public final void wait() throws InterruptedException;
// put this thread to sleep, be ready again in timeout milliseconds
public final void wait(long timeout) throws InterruptedException;
// make ANY other sleeping thread ready (never use!)
public final void notify();
// make ALL other sleeping threads ready
public final void notifyAll();
// interrupt thread (signal is not lost, if it is not currently waiting)
Thread t;
t.interrupt();
// make sure to catch InterruptedException and cease work in that thread!
```

Happens-before Relation

If t1 “happens before” t2, it is guaranteed that potential side effects of t1 are visible to t2.

Rules that create “happens-before”-relationship:

- same thread + data dependency
- statements in parent thread before `Thread.start` -> statements in the thread
- statements in the thread -> statements in the parent thread after `Thread.join`
- between synchronized blocks of the same monitor
- write to a volatile variable -> every subsequent read to that variable

`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

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

Executors

- Executors abstract from thread creation
- provide method `execute` that runs a `Runnable` in a thread according to strategy
- `ExecutorService` is an interface that provides further lifecycle management logic (e.g. `Futures`):

```
ExecutorService executor = Executors.newCachedThreadPool();
Callable<Integer> myCallable = () -> { return 42; };
Future<Integer> myFuture = executor.submit(myCallable);
int x = myFuture.get();
int x = myFuture.get(1, TimeUnit.SECONDS); // may throw TimeoutException
```

Atomic

Atomic operations are either executed completely or not at all. Atomic operations:

- reads and writes of reference variables
- reads and writes of 32-bit primitives
- reads and writes of all variables using `volatile`
- NOT: `i++`, `x=y+1`

```
class AtomicInteger {
  int get()
  int incrementAndGet()
  int decrementAndGet()
  boolean compareAndSet(int oldValue, int newValue)
  // more like tryReplace, result iff successful
}
```

Design by Contract

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

- P: precondition → specification what the supplier can expect from the client
- C: series of statements → the method of body
- Q: postcondition → specification of what the client can expect from the supplier if the precondition is fulfilled
- **Non-Redundancy-Principle:** the body of a routine shall not test for the routine's precondition
- **Precondition Availability:** precondition should be understandable by every client
- **Assertion Violation Rule:** a runtime assertion violation is the manifestation of a bug in the software

```
class Stack {
  //@ invariant size >= 0

  /*@ requires size > 0;
    @ ensures size == \old(size) - 1;
    @ ensures \result == \old(top());
    @ ensures (\forall int i; 0 <= i && i < size; \old(elements[i]) == elements[i]);
    @ assignable size; // redundant
    @ signals (IllegalOperationException ioEx) size == 0; // redundant
    @*/
  Object pop() { ... }

  // also: ==>, <==>, <=!=>, \exists, @ pure
  /*@ nullable @*/ /*@ pure @*/ Object top() { ... }
}
```

Liskov Substitution Principle

Guarantees may strengthen, requirements may weaken.

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

Compiler

Basics

- **Lexikalische Analyse (Lexing)**
  - Eingabe: Sequenz von Zeichen
  - Aufgaben:
    - erkenne Tokens = bedeutungstragende Zeichengruppen
    - überspringe unwichtige Zeichen (Whitespace, Kommentare)
    - Bezeichner identifizieren und zusammenfassen in Stringtabelle
  - Ausgabe: Sequenz von Tokens und Stringtabelle
- **Syntaktische Analyse (Parsing)**
  - Eingabe: Sequenz von Tokens
  - Aufgaben:
    - überprüfe, ob Eingabe zu kontextfreier Sprache gehört
    - erkenne hierarchische Struktur der Eingabe
  - Ausgabe: Abstrakter Syntaxbaum (AST)
- **Semantische Analyse**
  - Eingabe: Syntaxbaum
  - 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 (Pfeile von Verwendung zu Definition)
  - Ungültige Programme werden spätestens in Semantischer Analyse abgelehnt
- **Zwischencodegenerierung, Optimierung**
- **Codegenerierung**
  - Eingabe: Attributierter Syntaxbaum oder Zwischencode
  - Aufgaben: Erzeuge Code für Zielmaschine (Maschinenbefehle wählen, Scheduling, Registerallokation, Nachoptimierung)
  - Ausgabe: Program in Assembler oder Maschinencode

Grammatiken

Eigenschaften

Links- / Rechtsableitung: linkestes / rechtestes Nichtterminal wird zuerst weiterverarbeitet
Eindeutig: für jedes Wort existiert nur eine Ableitungsbaum
Konkreter Syntaxbaum (CST) = Ableitungsbaum (jedes Zeichen ein eigener Knoten, alle Terminale sind Blätter)

Konstruktion

- Operator precedence: Mehr Nichtterminale, schwächer bindende bilden auf stärker bindende ab, Klammern am Ende
  - Bsp: $P = \{E \to T, T \to F, T \to T * F, E \to E + T, F \to id, F \to (E)\}$
- Linksassoziativ: $E \to E + T$, Rechtsassoziativ: $E \to T + E$

LL, First, Follow

- LL: Parser liest einmal von Links nach rechts und baut Linksableitung auf (top-down, recursive decent)
- LR: Parser liest einmal von Links nach rechts und baut Rinksableitung auf (bottom-up)
- SLL(k) / SLR(k): vorherige Zeichen nicht relevant, k langer Lookahead

Für $\chi \in (\Sigma \cup V)^*$:
$\text{First}_k(\chi) = \{\beta \mid \exists \tau \in \Sigma^* : \chi \Rightarrow^* \tau \land \beta = \tau[..k]\}$ = k-Anfänge der Strings, die aus χ generiert werden können
$\text{Follow}_k = \{\beta \mid \exists \alpha, \omega \in (\Sigma \cup V)^* \text{ mit } S \Rightarrow^* \alpha\chi\omega \land \beta \in \text{First}_k(\omega)\}$ = k-Anfänge der Strings, die **hinter** χ generiert werden können

- Kontextfreie Grammatik ist SLL(k) gdw. für alle Produktionen eines Nichtterminals $A \to \alpha$ die Mengen $\text{First}_k(\alpha\text{Follow}_k(A))$ unterschiedlich sind.
  - $k = 1, \alpha \not\Rightarrow^* \epsilon, \beta \not\Rightarrow^* \epsilon$: genügt, wenn $\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$
  - $k = 1, \alpha \Rightarrow^* \epsilon, \beta \not\Rightarrow^* \epsilon$: genügt, wenn $\text{Follow}(A) \cap \text{First}(\beta) = \emptyset$
- 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 $L(G) = L(G')$

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:

```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 das ergebnis auf den stack (y * z)
iload_1 // lade Variable 1 (x) und lege sie auf den Stack
iadd // addiere die oberen zwei Elemente und lege sie auf den stack
istore_1 // pop stack und speichere Wert in Variable 3 (z)
```

Loops

Java:

```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 bar 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
```