Haskell

Operatoren

```
==, /=, >, <, <= , >=
```

Arithmetikbefehle

```
2 + 3 = 5
2 - 3 = -1
2 * 3 = 6
3 / 2 = 1.5
sqrt 9 = 3.0
max 2 3 = 3
min 2 3 = 2
sin x = ...
div 27 6 = 4
mod 27 6 = 3
succ 9 = 10
```

Listenbefehle

```
[1..10] = [1,2,...,10]
1:[2,3] = [1,2,3]
[1,2]++[3,4] = [1,2,3,4]
length [1,2,3] = 3
head [1,2,3] = 1
tail [1,2,3] = [2,3], tail [3] = []
init [1,2,3] = [1,2]
last [1,2,3] = 3
null [1,2,3] = False, null [] = True
take 2 [1,2,3] = [1,2]
drop 2 [1,2,3] = [3]
reverse [1,2,3] = [3,2,1]
maximum [1,3,2] = 3
minimum [3,2,1] = 1
sum [1,2,3] = 6
product [1,2,3] = 6
2 `elem` [1,2,3] = True
map (\x -> x+3) [1,2,3] = [4,5,6]
map (+3) [1,2,3] = [4,5,6]
filter (>1) [1,2,3] = [2,3]
foldr (+) 0 [1,2,3] = (1+(2+(3+0)))
foldl (+) 0 [1,2,3] = (((0+1)+2)+3)
concat [[1,2],3] = [1,2,3]
zipWith (+) [1,2] [3,4] = [4,6]
```

Arithmetikfunktionen

```
-- max
max x y
  | (x>y) = x
  | otherwise = y

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

-- factorial default
fak n = if (n==0) then 1 else (n * fak (n-1))

-- factorial with accumulator
fakAcc n acc = if (n==0) then acc else fakAcc (n-1) (n * acc)
fak n = fakAcc n 1

-- fibonacci
fib n
  | (n==0) = 0
  | (n==1) = 1
  | otherwise = fib (n-1) + fib (n-2)

-- fibonacci with accumulators
fibAcc n n1 n2
  | (n==0) = n1
  | (n==1) = n2
  | otherwise fibAcc (n-1) n2 (n1+n2)
```

Listenfunktionen

```
-- list length (not needed, see list above)
length l = if (null l) then 0 else 1+(length(tail l))

-- list length using pattern matching
length [] = 0
length (x:xs) = 1 + length xs

-- list maximum (not needed, see list above)
lmax [] = error "empty"
lmax(x:[]) = x
lmax (x:xs) = max x (lmax xs)

-- append lists (not needed, see list above)
app [] r = r
app (x:xs) r = x:(app xs r)

-- reverse lists
rev [] = []
rev (x:xs) = app (rev xs) [x]

-- prime sieve
primes :: Integer -> [Integer]
primes n = sieve [2..n]
  where sieve [] = []
        sieve (p:xs) = p : sieve (filter (not . multipleOf p) xs)
        multipleOf p x = x `mod` p == 0

-- first n squares
squares n = [x * x | x <- [0..n]]

-- quicksort
qsort (p:ps) =      (qsort (filter (<=p) ps))
               ++ p:(qsort (filter (>p) ps))
```

Damen-Problem

```
type Conf = [Int]

successors :: Conf -> [Conf]
successors board = map (:board) [1..8]

threatens :: Int -> (Int, Int) -> Bool
threatens row1 (diag, row2) = row1 == row2
  || abs (row1-row2) == diag

legal :: Conf -> Bool
legal [] = True
legal (row:rest) = not (any (threatens row) (zip [1..] rest))

solution :: Conf -> Bool
solution board = (length board) == 8

backtrack :: Conf -> [Conf]
backtrack conf =
  if (solution conf) then [conf]
  else concat (map backtrack (filter legal (successors conf)))

queensSolutions :: [Conf]
queensSolutions = backtrack []
```

Klausurlösungen

```
-- recursively sum up list by halfing
sumDQ2 [] 0 = 0.0
sumDQ2 [x] 1 = x
sumDQ2 l n = let n2 = n `div` 2 in
             sumDQ2 (take n2 l) n2 + sumDQ2 (drop n2 l) (n - n2)
sumDQ xs = sumDQ2 xs (length xs)
```

Prolog

Vordefinierte Atome und Prädikate

```
halt % Prolog verlassen
listing % Inhalt Datenbank ausgeben
consult('datei') % Datenbank laden
write(wert) % Wert ausgeben
format('Hello ~s', [name]) % Ausgabe Hello Name
nl % Neue Zeile ausgeben
assert(fakt) % neuen Fakt hinzufügen
retract(fakt) % alten Fakt entfernen
```

Arithmetikfunktionen

```
% fibonacci
fib(0,0).
fib(1,1).
fib(X,Y) :- X>1,
  X1 is X-1, X2 is X-2,
  fib(X1,Y1), fib(X2,Y2),
  Y is Y1+Y2.

% test if natural number.
nat(0).
nat(X) :- nat(Y), X is Y+1.

% compute approximate square root.
sqrt(X,Y) :- nat(Y),
  Y2 is Y*Y, Y3 is (Y+1)*(Y+1),
  Y2 =< X, X < Y3.
```

```
% test evenness.
even(0).
odd(1).
even(X) :- X > 0, X1 is X-1, odd (X1).
odd(X) :- X > 1, X1 is X-1, even(X1).
```

Listenfunktionen

```
% tests if list includes element.
member(X,[X|R]).
member(X,[Y|R]) :- member(X,R).

% appends element to list.
append([],L,L).
append([X|R],L,[X|T]) :- append(R,L,T).

% reverses list.
rev([],[]).
rev([X|R],Y) :- rev(R,Y1),append(Y1,[X],Y).

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

% quicksort
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).
```

Cut-Funktionen

```
% green-cut max
max(X,Y,X) :- X>Y, !.
max(X,Y,Y) :- X=<Y.

% red-cut max
max(X,Y,Z) :- X>Y, !, Z=X.
max(X,Y,Y).

% negation
not(X) :- call(X), !, fail.
not(X).
```

MPI

Base methods

- `MPI_INIT(&argc, &args)`: initialize MPI
- `int MPI_Comm_rank(MPI_Comm comm, int* rank)`: retrieve number of processing node (rank) withing communicator
- `int MPI_Comm_size(MPI_Comm comm, int* size)`: retrieve total number of processes in communicator
- `int MPI_BARRIER(MPI_Comm comm)`: block until all processes have called it
- `int MPI_Send(void* buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`: blocking, asynchronous. Blocks until message buffer can be reused
  - `buffer`: pointer to sender's buffer (free choice type)
  - `count`/`datatype`: number/type of buffer's elements
  - `dest`: rank of the destination process
  - `tag`: message "context" (e.g. conversion ID)
  - `comm`: communicator of process group
- `int MPI_Rect(void* buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status)`
  - `source`: wildcard possible (`MPI_ANY_SOURCE`)
  - `tag`: wildcard possible (`MPI_ANY_TAG`)
  - `MPI_PROBE`: can be used to retrieve messages of unknown length
  - `MPI_Status`: required, because tag + source can be unknown when using wildcards
- `MPI_Finalize()`: clean up after using MPI

Operation Modes (blocking)

- `MPI_Send`: standard-mode blocking send
- `MPI_Bsend`: buffered-mode blocking send
- `MPI_Ssend`: synchronous-mode blocking send
- `MPI_Rsend`: ready-mode blocking send

Operation Modes (non-blocking)

- `int MPI_Isend(void* buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request* request)`
- `int MPI_Irecv(void* buf, int count, MPI_Datatype type, int src, int tag, MPI_Comm comm, MPI_Request* request)`
  - `request`: pointer to status information about operation
- `int MPI_Test(MPI_Request* r, int* flag, MPI_Status* s)`: non-blocking check, flag := 1 if operation completed (0 otherwise)
- `int MPI_Wait(MPI_Request* r, MPI_Status* s)`: blocking check

Broadcasting

- `int MPI_Bcast(void* buffer, int count, MPI_Datatype t, int root, MPI_Comm comm)`
  - `root`: rank of message sender, uses `buffer` to provide data
  - receivers use `buffer` to receive data (other parameters must be identical)
- `int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`: all receivers get equal-sized, but *content-different* data
  - `sendcount/recvcount`: # elements sent/received by one process, usually equal
- `int MPI_Scatterv(void* sendbuf, int* sendcounts, int* displacements, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`: vector variant of `MPI_Scatter`
  - allows varying counts for data sent to each process
  - `sendcounts`: int array with number of elements to send to each process
  - `displacements`: int array, entry `i`: displacement relative to `sendbuf` from which to take outgoing data to process `i` (gaps allowed, no overlaps)
  - `sendtype`: data type of send buffer elements (handle)
  - `recvcount`: number of elements in receive buffer (int)
  - `recvtype`: data type of receive buffer elements (handle)
- `int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`:
  - `root`'s buffer contains collected data *sorted by rank*, including `root`'s own buffer contents
  - receive buffer ignored by all non-root processes
  - `recvcout`: # items received from *each process*
  - = inverse of `MPI_Gater`
  - vector variant: `MPI_Gaterv`
- `int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
  - ≅ gather + broadcast
  - for each process p_j in `comm`, p_j collects + sends same data to all other proceses in `comm`
  - ⇝ buffer of each process in `comm` has *same* data (including own data) in *same* order
  - vector variant: `MPI_Allgatherv`
- `int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
  - sender p_s: sends to receiver p_r only its r-th element
  - receiver p_r: stores information from sender p_s at position s in its buffer
  - `MPI_Alltoallw`: separate specification of count, displacement, datatype for each block
  - vector variant: `MPI_Alltoallv`

Reduce

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

- applies operation to data in `sendbuf`, stores result in `recvbuf` of root process
- `count`: # columns in output buffer
- `op`: either custom defined or
  - logical/bitwise and/or: `MPI_LAND`, `MPI_BAND`, `MPI_LOR`, `MPI_BOR`
  - `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`,…
  - `MPI_MINLOC`, `MPI_MAXLOC`: return rank of minimum/maximum

Java Parallel

Functional Interfaces

```
@FunctionalInterface
interface Predicate {
```

```
  boolean check(int value);
}

public int sum(List<Integer> values, Predicate predicate) {
  int result = 0;
  for (int value : values) {
    if (predicate.check(value)) result += value;
  }
  return result;
}

// example use
sum(values, i -> i > 5);
```

Method References

```
class SimpleCheckers {
  public static boolean checkGreaterThanFive(Integer value) {
    return value > 5;
  }
}

// example use
sum(values; SimpleCheckers::checkGreaterThanFive);
```

Thread Class

```
class SayHelloThread extends Thread {
  public void run () {
    System.out.println("Hello");
  }

  public static void main(String[] args) {
    (new SayHelloThread()).start();
  }
}
```

Methods:

- `run()`: defined thread task
- `start()`: starts thread
- `isAlive()`: returns if thread is still running
- `sleep(long milliseconds)`: sleeps thread for given time
- `interrupt()`: sets interrupted flag for thread
- `isInterrupted()`: return whether interrupted flag is set

Runnable Class

```
class SayHelloRunnable implements Runnable {
  public void run() {
    System.out.println("Hello");
  }

  public static void main(String[] args) {
    (new Thread(new SayHelloRunnable())).start();
  }
}
```

Runnable using Lambda

```
  class SayHelloRunnable implements Runnable {
    public static void main(String[] args) {
      (new Thread(() -> System.out.println("Hello"))).start();
    }
  }
```

Thread Joining

Here: force output of "Hello" before "Goodbye"

```
class SayHelloRunnable implements Runnable {
  public void run() {
    System.out.println("Hello");
  }
  public static void main(String[] args)
    throws InterruptedException {
    Thread thread = new Thread(new SayHelloRunnable());
    thread.start();
    thread.join();
    System.out.println("Goodbye");
  }
}
```

Thread Result Passing

Define `getResult`-Method returning result after joining thread

```
MyThread t = new MyThread();
t.start();
t.join();
String result = t.getResult();
```

Thread Priorities

- `void setPriority(int priority)`: set priority of a thread
- by default, threads inherit priority of creating thread
- predefined: `Thread.MIN_PRIORITY`/`NORM_PRIORITY`/`MAX_PRIORITY`

Synchronization — Monitors

```
public void doSomething() {
  synchronized(someObject) {
    // critical, lock held on someObject
  }
}

// whole method critcal, monitor = this
public synchronized void doSomething2() {
  // critical
}
```

volatile Keyword

- ensures that changes to variables are immediately visible to all threads

→ happens-before relationship: write to volatile happens-before every subsequent read of that volatile

Guarded Blocks

- poll condition that must be true before proceeding
- Idea: `while (!condition){}; doSomething();`
- **Signals**: = operations that can be callled on monitors for coordination
- only thread "owning" monitor can use signals on it → only use inside synchronized block!
- `wait()`: release monitor so another thread can enter
  - thread waits for `notify()` or `notifyAll()`
  - can throw `InterruptedException`
  - overloaded method with timeout parameter available
- `notify()`: wake up one thread that called `wait()` on monitor
  - can wake "wrong" thread (e.g. in producer-consumer context)
- `notifyAll()`: wake all threads waiting for monitor
  - safer choice than `notify()`

```
public class MyBlockingStack {
  public synchronized void push (Object o) {
    while (count == max) {
      try {
        this.wait();
      } catch (InterruptedException e) {
        // ...
      }
    }
    stack.push(o);
    this.notifyAll();
  }

  public synchronized Object pop () {
    while (count == 0) {
      try {
        this.wait();
      } catch (InterruptedException e) {
        // ...
      }
    }
    this.notifyAll();
    return stack.pop();
  }
}
```

Immutable Objects

- simple way to avoid concurrency problems
- make class immutable:
  1. declare all fields `private` and `final`
  2. no setter methods
  3. no overwriting → declare class as `final`
  4. references to mutable classes are not modifiable through methods
  5. references to mutable classes are not shared

Atomic Types

Types supporting atomic operations on single variables (e.g. AtomicInteger from java.util.concurrent.atomic)

Locks

```java
public void doSomething () {
  lock.lock();
  try {
    // critical
  } finally {
    lock.unlock();
  }
}
```

- `tryLock()`: acquire lock if possible (can be used to acquire several locks without blocking)

Counting Semaphore

- `Semaphore(int capacity, boolean fair)`
- critical section can be entered n times
- **permit**: one access to critical section
- `aquire()`: takes permit, reduces # available permits, blocks if all permits aquired
- `release()`: releases permit, increases # available permits, potentially releases blocking aquirer
- `tryAquire()`: non-blocking `aquire()`
- all calls possible with arbitrary # permits (`void aquire(int amount)`)

Barriers

- `CyclicBarrier(int n)`
  - `await()` blocks calling thread — if called n times, all threads resume
  - can be reused afterwards
- `CountDownLatch(int n)`
  - `await()` blocks calling thread
  - if `countdown()` called n times, all threads resume
  - latch cannot be restarted afterwards
  - further calls to `await()` return immediately

`Exchanger<V>()`: synchronous exchange of two objects between two threads
  - `V exchange(V objectToExchange)`: exchange method has to be called by both threads, blocks until both have called

Executor

- abstract from thread creation
- simple implementation: only start thread
- complex implementation: e.g. reuse already created threads
- `Executor`: simple interface, supports task execution
  - `void execute(Runnable runnable)`
- `ExecutorService`: most important interface
  - subinterface of `Executor`, provides further lifecycle management logic
- `Executors`: class providing convenient factory methods for creating `ExecutorService`
  - `newSingleThreadExecutor()`: creates `Executor` using single thread
  - `newFixedThreadPool(int)`: creates thread pool with reused threads, fixed size
  - `newCachedThreadPool()`: creates thread pool with reused threads, dynamic size

```java
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(() -> {
  String threadName = Thread.currentThread().getName();
  System.out.println("Hello " + threadName);
})

try {
  executor.shutdown();
  executor.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException ex) {
  // handle
} finally {
  if (!executor.isTerminated()) {
    executor.shutdownNow();
  }
}
```

Callable Interface

- allows threads to return results
- similar to `Runnable`: `call()` instead of `run()`

```java
public class MyCallable implements Callable<String> {
  int id;

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

  public String call () {
    return "Run " + id;
  }
}
```

Futures

```java
ExecutorService executorService = Executors.newCachedThreadPool();
List<Future<Integer>> futures = new ArrayList<Future<Integer>>();

for (int i = 0; i < 10; i++) {
  final int currentValue = i;
  futures.add(executorService.submit(() -> { return currentValue; }))
}
for (Future<Integer> future : futures) {
  try {
    Integer result = future.get(); // like JS await
    System.out.println(result);
  } catch (ExecutorException ex) {}
}
executorService.shutdown();
```

Scheduled Executor Service

`ScheduledExecutorService`: subinterface of `ExecutorService`, supports scheduled task execution

- Future: `schedule(Runnable task, long delay, TimeUnit timeunit)`
- Periodic: `scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit timeunit)`

```java
ScheduledExecutorService executor =
  Executors.newScheduledThreadPool(1);
Runnable task = () -> System.out.println("Scheduling: "
  + System.nanoTime());
ScheduledFuture<?> future =
  executor.schedule(task, 3, TimeUnit.SECONDS);
TimeUnit.MILLISECONDS.sleep(1337);
long remainingDelay = future.getDelay(TimeUnit.MILLISECONDS);
System.out.println("Remaining Delay: " + remainingDelay);
```

Completable Future

- `Future` drawback: caller can query result, but not register callback
- `CompletableFuture`: provides `thenApply`, like `then` in ES6

```java
CompletableFuture<String> modified =
  futureCount.thenApply((Integer count) -> {
    int transformedValue = count * 10;
    return transformedValue;
  })
  .thenApply(transformed -> "Finally create a string: "
    + transformed);
System.out.println(modified.get());
```

Fork and Join

- Java provides abstract `ForkJoinPool` and `ForkJoinTasks`
- Concretizations: `RecursiveAction` (no result), `RecursiveTask` (result)
- can be executed by `ForkJoinPool` calling its `invoke()` method

Task implementation:

```java
public class MyTask extends RecursiveTask<Integer> {
  @Override
  public static Integer compute () {
    // calculate sth, return if problem small

    // divide task
    MyTask leftTask = new MyTask(...);
    MyTask rightTask = new MyTask(...);
    leftTask.fork();
    rightTask.compute(); // one subtask in-place
    leftTask.join();

    // compute result
  }
}
```

Thread-safe Classes

- can be used safely by multiple threads concurrently
- `BlockingQueue` (interface):
  - queue with ops that block if queue full/empty/putting/retrieving

- put(...), take()
- Implementations:
  - ArrayBlockingQueue: limited capacity
  - LinkedBlockingQueue: optionally limited capacity
  - PriorityBlockingQueue: sorted
- ConcurrentHashMap

Streams

```java
public class Person {
  private final boolean isStudent;
  private final int age;
  public boolean isStudent () { return isStudent; }
  public int getAge () { return age; }

  public Person (boolean isStudent, int age) {
    this.isStudent = isStudent;
    this.age = age;
  }
}

double average = personsInAuditorium
  .stream()
  .filter(Person::isStudent)
  .mapToInt(Person::getAge)
  .average()
  .getAsDouble();

// collect op example
// 1. supplier: supplies new result container
// 2. accumulator: incorporates new element into result
// 3. combiner: combines two values, must be
//    compatible with result
//    (note: not used here, only for parallel comp.)
personsInAuditorium.stream().collect(
  () -> 0,
  (currentSum, person) -> { currentSum += person.getAge(); },
  (leftSum, rightSum) -> { leftSum += rightSum })
```

- predefined collectors exist
- e.g. mapping, summing up, grouping,...

Parallel Stream

Stream executing certain operations automatically in parallel

```java
double average = personsInAuditorium
  .parallelStream()
  .filter(Person::isStudent)
  .mapToInt(Person::getAge)
  .average()
  .getAsDouble();
```

Design By Contract

Java Modeling Language (JML)

- @requires: pre-condition
- @ensures: post-condition
- a ==> b: a implies b
- a <==> b: a iff b
- a <=!=> b: !(a iff b)
- \result: result of method call
- \old(E): value of E in state before method execution
- \forall [declaration, range, body]
- \exists [declaration, range, body]
- \max, \min, ...

```java
/*@ 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])
*/
Object pop () { ... }
```

JML — Class Invariants

- define conditions containing class must hold in all user visible states

```java
class Stack {
  //@ invariant size >= 0 && size <= 10;
  private Object[] elements = new Object[10];
  private int size = 0;
  ...
}
```

JML — Liskov Substitution

- Applies to overwriting methods
- **Preconditions**: must *not* be more restrictive than those overwritten: $\mathrm{Pre}_{\mathrm{Sup}} \Rightarrow \mathrm{Pre}_{\mathrm{Sub}}$
- **Postconditions**: must be *at least* as restrictive as those of overwritten method: $\mathrm{Post}_{\mathrm{Sub}} \Rightarrow \mathrm{Post}_{\mathrm{Sup}}$
- **Invariants**: must be *as least* as restrictive as those of super-class: $\mathrm{Inv}_{\mathrm{Sub}} \Rightarrow \mathrm{Inv}_{\mathrm{Sup}}$

JML – Pure Methods

- *not allowed* to reference methods causing side effects in contracts
- mark methods as **pure**:

```java
public /*@ pure @*/ int getElementCount () {
  return size;
}
```

Java Byte Code

Laufzeitsystem

- **Heap**: Speicher Objektinstanzen, gemeinsam für alle Threads
- **Method Area**: Methodencode, nur lesbar
- **Runtime Constant Pool**: Konstante Daten (z.B. Typinformationen)
- **Threads**: je Thread
  - Program Counter
  - JVM Stack: Activation Records (Stackframes)
  - Native Method Stack: für Laufzeitsystem (meist C/C++)

Instruktionen

- Typen wie in Java, Instruktionen typisiert (z.B. iadd(int))
- lokale Variablen lesen/schreiben: ?load, ?store <x>
- Felder lesen/schreiben: getfield, putfield
- Springen: ifeq, ifnull, tableswitch
- Methodenaufrufen: invokevirtual, invokestatic
- Objekterzeugung: new, newarray
- Arithmetische Berechnungen: ?mul, ?add

Simples JBC

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

```
iconst 4 // lade Konstante 4
istore 3 // schreibe in z
iload 2 // lade y
iload 3 // lade z
imul // y * z
iload 1 // lade x
iadd // (y * z) + x
istore 1 // speichere in x
```

JBC — Fibonacci

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

```
iconst 1
istore 2
iconst 1
istore 3
loop_begin:
iinc 1 -1
iload 1
ifle after_loop
iload 2
iload 3
iadd
istore 4
iload 2
istore 3
```

```
  iload 4
  istore 2
  goto loop_begin
  after_loop
  iload 2
  ireturn
```

JBC — Arrays

```
  public void arr () {
    int[] array = new int[10];
    array[7] = 42;
  }
```

```
  bipush 10
  newarray int
  astore 1
  aload 1
  bipush 7
  bipush 42
  iastore
  return
```

JBC — Felder

```
  class Foo {
    public Bar field;
    public void setNull () {
      field = null;
    }
  }
```

```
  aload 0 // Parameter 0 ist IMMER this!!!
  aconst_null
  putfield Foo.field:LBar;
  return
```

JBC — if

```
  if (x == 4) { A } else { B }
```

```
  iload 0
  bipush 4
  if_icmpeq label0
  goto label1
label0:
  A // then-teil
  goto label2
label1:
  B // else-teil
  goto label2
label2:
  // rest
```

JBC — while

```
  while (x < 10) { A }
```

```
loopheader:
  iload 0
  bipush 10
  if_icmplt loopbody
  goto afterloop
loopbody:
  A // body
  goto lopheader
afterloop:
  // rest
```

JBC — Negation

Kein Bytecode-Befehl ⇝ Sprungziele vertauschen