📖 **Funktionen**

Prolog

Meta

```
atom(X). /* Ist atom */
integer(X). /* Ist integer */
atomic(X). /* Ist atom oder integer */
var(X). /* Ist *uninstanziierte* Variable */

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

Operatoren

```
+, -, *, / /* Auswertung mit is: X is A + 20 */
A = B /* Unifikationsconstraint */
A == B /* Erfolgreich nur, *falls schon unifiziert* */
=:=, =\=, <, =<, >, >= /* Arithmetische Vergleiche: =, ≠, <, ≤, >, ≥, erfordern Instanziierung! */
```

Listen

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

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

delete([X|L],X,L).
delete([X|L],Y,[X|L1]) :- delete(L,Y,L1).

rev([],[]).
rev([X|R],Y) :- rev(R,Y1),append(Y1,[X],Y).
/* Effizientes rev */
rev(X,Y) :- rev1(X,[],Y).
rev1([],Y,Y).
rev1([X|R],A,Y) :- rev1(R,[X|A],Y).

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

map(F,[],[]) :- !. /* ? map(sqrt,[1,4,9,16],L). ⇒ Yes, L = [1, 2, 3, 4]. */
map(F,[H|T], [NH|NT]) :- G =..[F,H,NH], call(G), map(F,T,NT).
```

Sonstiges

```
/* Lookup in normalen Dict: [(A, b)] für A => b */
lookup(N,[(N,A)|_],A1) :- !,A=A1.
lookup(N,[_|T],A) :- lookup(N,T,A).
```

Haskell

Num / Enum

```
ceil  :: (RealFrac a, Integral b) => a -> b
floor :: (RealFrac a, Integral b) => a -> b
```

```
exp   :: Floating a => a -> a                 -- exponential function.
log   :: Floating a => a -> a
round :: (RealFrac a, Integral b) => a -> b -- rounds its argument to the nearest integer.
sqrt :: Floating a => a -> a

gcd :: Integral a => a -> a -> a
lcm :: Integral a => a -> a -> a -- returns the least common multiple of its two integral arguments

-- Ord / Enum
max  :: Ord a => a -> a -> a
min  :: Ord a => a -> a -> a
pred :: Enum a => a -> a
succ :: Enum a => a -> a

-- Operators
div :: Integral a => a -> a -> a -- integer division of integral arguments
mod :: Integral a => a -> a -> a -- integer modulus of integral arguments

^,**           -- power for int,floats.
/=, ==, <=, >= -- not equal, equal, less equal, greater
```

Foldable / List

```
-- Base functions
filter :: (a -> Bool) -> [a] -> [a]
foldl  :: (a -> b -> a) -> a -> [b] -> a  -- ((init + 1) + 2) + 3
foldr  :: (a -> b -> b) -> b -> [a] -> b  -- 1 + ( 2 + ( 3 + init))
map    :: (a -> b) -> [a] -> [b]

-- Arithmetic
maximum :: Ord a => [a] -> a
minimum :: Ord a => [a] -> a
product :: Num a => [a] -> a
sum     :: Num a => [a] -> a

-- Creation
concat    :: [[a]] -> [a]
iterate   :: (a -> a) -> a -> [a] -- [x,f(x),f(f(x)),...]
repeat    :: a -> [a]             -- infinite list of same value
replicate :: Int -> a -> [a]      -- replicates item n times
zip       :: [a] -> [b] -> [(a,b)]
zipWith   :: (a -> b -> c) -> [a] -> [b] -> [c]

-- Checks
all     :: (a -> Bool) -> [a] -> Bool
and     :: [Bool] -> Bool
or      :: [Bool] -> Bool
any     :: (a -> Bool) -> [a] -> Bool
elem    :: Eq a => a -> [a] -> Bool
notElem :: Eq a => a -> [a] -> Bool
length  :: [a] -> Int
null    :: [a]

-- Extraction (partial functions!)
head   :: [a] -> a   -- [1, 2, 3] -> 1
last   :: [a] -> a   -- [1, 2, 3] -> 3
init   :: [a] -> [a] -- [1, 2, 3] -> [1, 2]
tail   :: [a] -> [a] -- [1, 2, 3] -> [2, 3]

-- Dicts
lookup :: Eq a => a -> [(a, b)] -> Maybe b

-- Modification
break     :: (a -> Bool) -> [a] -> ([a],[a]) -- break on first satisfiead (snd=[] if never)
span      :: (a -> Bool) -> [a] -> ([a],[a]) -- splits list as takeWhile and rest
```

```
dropWhile :: (a -> Bool) -> [a] -> [a]       -- drops while predicate is satisfied
takeWhile :: (a -> Bool) -> [a] -> [a]
splitAt   :: Int -> [a] -> ([a],[a])
reverse   :: [a] -> [a]
sort      :: Ord a => [a] -> [a]
```

Strings

```
lines :: String -> [String]--split by newline.
unlines :: [String] -> String
unwords :: [String] -> String--join with spaces.
words :: String -> [String]--split with spaces.
```

Chars

```
chr :: Int -> Char
digitToInt :: Char -> Int
isAlpha :: Char -> Bool
isDigit :: Char -> Bool
ord :: Char -> Int -- returns ASCII codepoint
```

Misc

```
-- Integer conversion
fromInt     :: Num a => Int -> a
fromInteger :: Num a => Integer -> a

-- Pairs
fst :: (a, b) -> a
snd :: (a, b) -> b

-- Bools
not :: Bool -> Bool

-- Functions
until :: (a -> Bool) -> (a -> a) -> a -> a -- applies function to value until predicate is satisfied
show  :: Show a => a -> String

-- Magic
undefined :: a
```

Java Bytecode

Type specifiers

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

Opcodes

```
// Constants
aconst_null // null obj ref
```

```
dconst_i    // i in [0, 1]
fconst_i    // i in [0, 2]
iconst_i    // i in [m1, 0..5]

bipush i    // signed byte pushen
sipush i    // signed short pushen

// Variables. Dedicated ops for [a, i, l, d, f], rest is i
Xload_i  // i in [0, 3] | load local var i
Xload i  // load local var i

Xstore_i // i in [0, 3] | store local var i
Xstore i // store local var i

// Misc. Dedicated ops for [a, i, l, d, f], rest is i
return  // return from void
Xreturn // return value of type X

// Comparisons
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 ≤

// Comparison to zero
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 (für [i, l, f, d])
iinc var const // increment variable var by const
isub           // Integer subtraction
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  // Array anlegen vom Typ T
Xaload      // Lade typ X von array [Stack: arr, index] <-
Xastore     // Speichere typ X in array [Stack: arr, index, val] <-
arraylength // Länge eines arrays
```