Prolog

var(Term) succeeds if Term is currently uninstantiated

nonvar(Term) succeeds if Term is currently instantiated

atom(Term) succeeds if Term is currently instantiated to an atom.

integer(Term) succeeds if Term is currently instantiated to an integer.

atomic(Term) succeeds if Term is currently instantiated to an atom, an integer or a floating point number.

list(Term) succeeds if Term is currently instantiated to a list

Term1 \= Term2 succeeds if Term1 and Term2 are not unifiable

Term1 = Term2 unifies Term1 and Term2.

Expr1 =:= Expr2 succeeds if eval(Expr1) = eval(Expr2).

Expr1 =\= Expr2 succeeds if eval(Expr1) ≠ eval(Expr2).

Expr1 < Expr2 succeeds if eval(Expr1) < eval(Expr2).

Expr1 =< Expr2 succeeds if eval(Expr1) ≤ eval(Expr2).

Expr1 > Expr2 succeeds if eval(Expr1) > eval(Expr2).

Expr1 >= Expr2 succeeds if eval(Expr1) ≥ eval(Expr2).

succ(X, Y) is true if Y is the successor of the non-negative integer X.

append(List1, List2, List12) succeeds if the concatenation of the list List1 and the list List2 is the list List12.

member(Element, List) succeeds if Element belongs to the List.

reverse(List1, List2) succeeds if List2 unifies with the list List1 in reverse order.

delete(List1, Element, List2) removes all occurrences of Element in List1 to provide List2.

select(Element, List1, List2) removes one occurrence of Element in List1 to provide List2.

subtract(List1, List2, List3) removes all elements in List2 from List1 to provide List3.

permutation(List1, List2) succeeds if List2 is a permutation of the elements of List1.

prefix(Prefix, List) succeeds if Prefix is a prefix of List.

suffix(Suffix, List) succeeds if Suffix is a suffix of List.

last(List, Element) succeeds if Element is the last element of List.

length(List, Length) succeeds if Length is the length of List.

nth(N, List, Element) succeeds if the Nth argument of List is Element.

min_list(List, Min) succeeds if Min is the smallest number in List.

max_list(List, Max) succeeds if Max is the largest number in List.

sum_list(List, Sum) succeeds if Sum is the sum of all the elements in List.

sort(List1, List2) succeeds if List2 is the sorted list corresponding to List1 where duplicate elements are merged.

msort(List1, List2) is similar to sort except that duplicate elements are not merged.

, und

; oder

Java Byte code

bipush = push a byte onto the stack as an integer value
iconst_x = load the int value x onto the stack, x ∈ [-1, 5]
iload x = load an int value from a local variable #x
istore x = store int value into variable x
ldc x = push a constant x

invokestatic x = invoke static method -> put the result on the stack; method is identified by method reference x in constant pool
invokevirtual x = invoke virtual method -> put the result on the stack; method is identified by method reference x in constant pool

newarray type = create new array with count (top of stack) elements of primitive type identified by type
aload x = push object reference in local variable x onto stack
astore x = store top of stack in local variable x
iastore = store top of stack (int) into an array

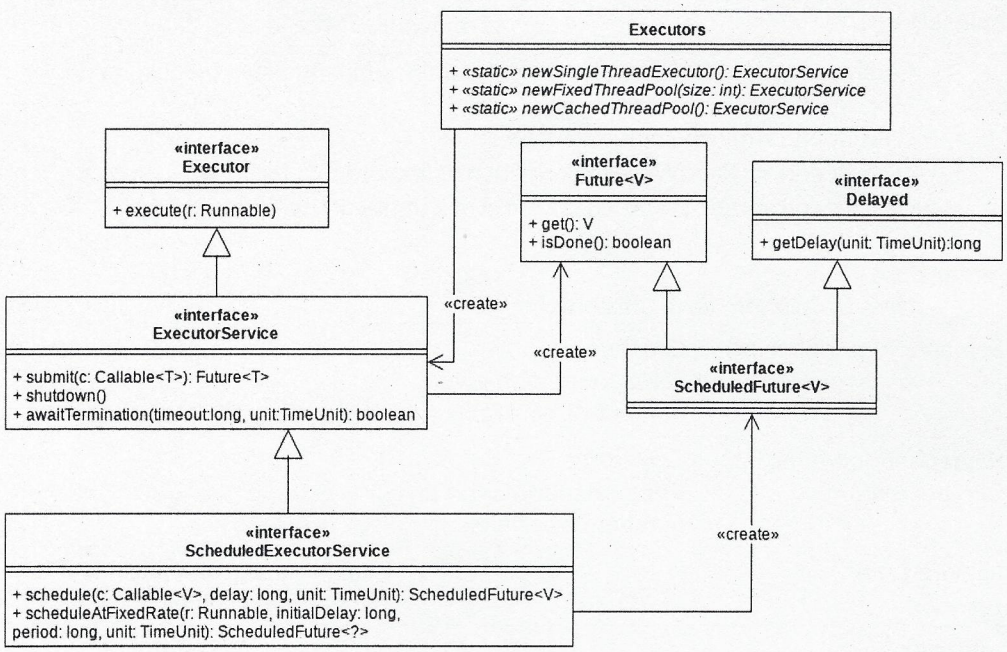
iadd = add two ints
isub = int subtract
imul = multiply two integers
idiv = divide two integers
iand = perform a bitwise AND on two integers
ineg = negate int
iinc <varnum> <n> inc local var varnum by n

if_icmpeq x = if ints are equal, branch to instruction at x
if_icmpge x = if value1 is greater than or equal to value2, branch to instruction at x
if_icmpgt x = if value1 is greater than value2, branch to instruction at x
if_icmple x = if value1 is less than or equal to value2, branch to instruction at x
if_icmplt x = if value1 is less than value2, branch to instruction at x
if_icmpne x = if ints are not equal, branch to instruction at x

(all instructions can be modified for other datatypes by replacing 'i' with 'f'(float), 'd'(double), ...)

goto x = goes to instruction at x
swap = swaps two top words on the stack
return = return void from method
ireturn = return an integer from a method

Executor



CompletableFuture

- class CompletableFuture<V> (java.util.concurrent) implements Future<V>

static *supplyAsync*(() → V): CompletableFuture<V>	CompletableFuture that will contain the result	Runs the supplier asynchronously
thenApply((result: super V) → extends T): CompletableFuture<T>	CompletableFuture that will contain functions' result as new Future	Runs the given function (blocking) with result

Fork-Join

- Fork-Join is a pattern for effectively computing divide-and-conquer algorithms in parallel
- Divide Problem in left and right side
- fork subtask for left side, compute right side in-place, join left side

Java

	was acquired	time of invocation
unlock(): void	-	Releases the lock

- Catch exceptions between locking and unlocking and place unlock() in finally block to ensure unlocking
- class ReentrantLock implements Lock
- class ReentrantReadWriteLock.ReadLock implements Lock
- class ReentrantReadWriteLock.WriteLock implements Lock

Semaphore

- class Semaphore (java.util.concurrent)

Semaphore(permits: int)	*Constructor* Semaphore with given number of permits	-
Semaphore(permits: int, fair: boolean)	*Constructor* Fair Semaphore with given number of permits	-
acquire(): void	-	Takes a permit (blocking if no permit available)
release(): void		Gives back a permit
acquire(n: int): void	-	Takes n permits (blocking until all permits available)
release(n: int): void		Gives back n permits
tryAcquire(): boolean	True iff a permit was acquired	Acquires a permit iff one is free at the time of invocation

Barriers

CyclicBarrier(n: int)	await():void	-	Thread is blocked. All threads are resumed after n invocations.
CountDownLatch(n: int)	await():void	-	Thread is blocked.
	countDown(): void	-	All threads are resumed after n invocations.
Exchanger<V>()	exchange(obj: V): V	Object from other thread	Blocks thread until a second thread has called this method and exchanges objects.

Java

Atomic Values

- class AtomicInteger (java.util.concurrent.atomic)

Method	Returns	Effect
get(): int	Current value	-
incrementAndGet(): int	Updated value	Atomically increases current value by one
compareAndSet(expect: int, new: int): boolean	True iff current value equals expect	Sets current value to new if expect equals current value

- Other classes: AtomicBoolean, AtomicLong, ...

The **volatile** keyword

- **Does not** guarantee *atomicity* of complex operations (e.g. x++)
- **Does** guarantee *visibility*
  - variable is written/read directly to/from main memory → changes are instantaneously visible in every thread
  - (since Java 5) *full visibility*: Changes to volatile and non-volatile variables prior to writes to volatile variables in thread A are visible in thread B.
- **Does** *restrict compile-time reordering* (introduces happens-before relationship) of instructions (since Java 5) to guarantee full visibility

Interfaces useful for concurrent operations

- interface Runnable (java.lang)

run(): void	-	Implemented by class

- interface Callable<V> (java.util.concurrent)

call(): V	V	Implemented by class

- interface Future<V> (java.util.concurrent)
  represents the result of an asynchronous computation

get(): V	Retrieves result (blocking)	-
isDone(): boolean	True iff task completed	-

Threads

- class Thread (java.lang)

run(): void	-	Defines the "task" of a thread. Usually overriden by subclass. Otherwise runs Runnable given in constructor or does nothing
start(): void	-	starts the thread
isAlive(): boolean	True iff running	-

static *sleep*(milliseconds: long): void	-	Causes the currently executing thread to sleep
interrupt(): void	-	Interrupt flag is set
isInterrupted(): boolean	Interrupt flag	-
join(): void	-	Waits for this thread to die
setPriority(prio: int): void		Effects scheduling

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

Mutual Exclusion
If one thread executes operations of a *critical section*, other threads will be blocked if they want to enter it as well.

- In Java: *Monitors*
  - method level: public synchronized void doSomething()
  - statement: synchronized(obj){...}
  - reentrant: Thread can enter the same monitor multiple times

Deadlock

- All of the folowing conditions must hold:
  - Mutal exclusion, hold and wait, no preemption, circular wait
- Lifelocks are similar but with oscillating states without true progress

notify/wait

- class Object (java.lang)

notify(): void	-	Wakes up a single thread that is waiting on this object's monitor
notifyAll(): void	-	Wakes up all threads that are waiting on this object's monitor
wait(): void	-	Wait until woken up

- Notify/wait may only be used by the owner of a object's monitor.
- Wait must be placed in a *Guarded Block* "while(!condition) {...}"
  - System might wake up threads randomly, race conditions,...

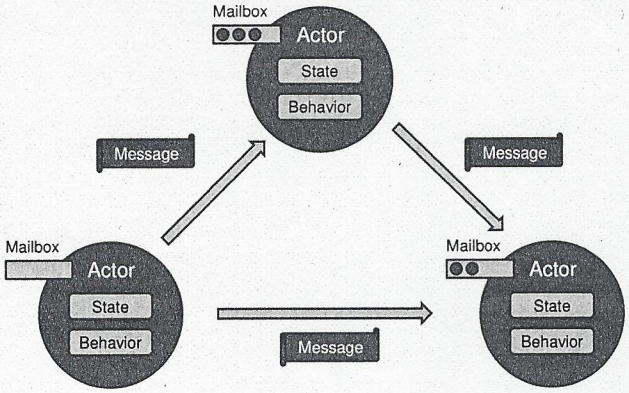
Locks

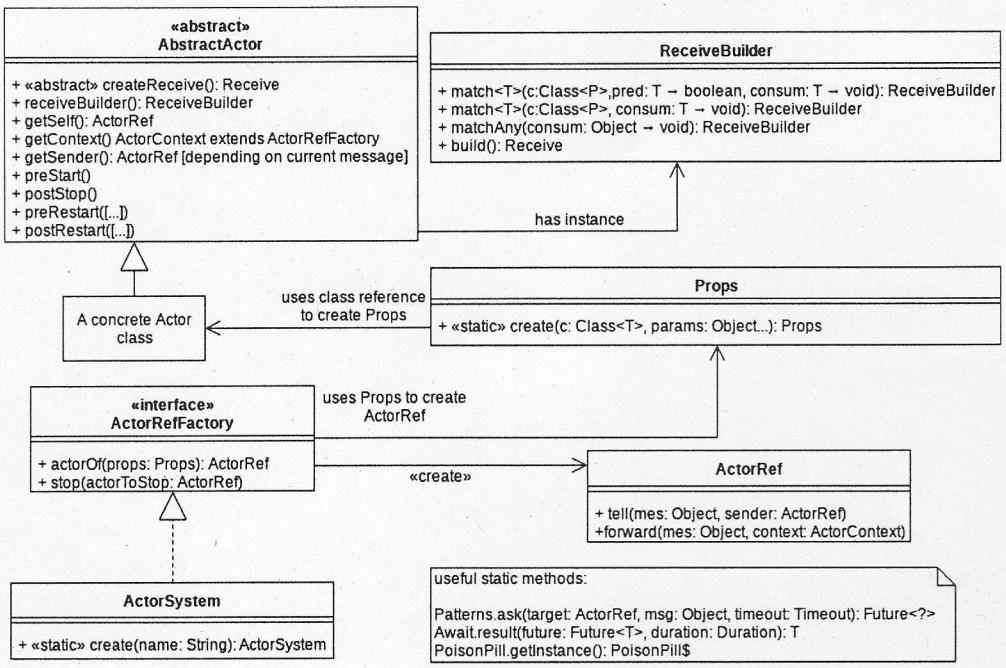
- interface Lock (java.util.concurrent.locks)

lock(): void	-	Acquires the lock
tryLock(): boolean	True iff the lock	Acquires the lock iff it is free at the

Java

Akka Actors





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

Input Parameters:

count: number of elements in send buffer

op: reduce operation

root: rank of root process

Output Parameters:

recvbuf: address of receive buffer

Reduces values on all processes to a single value

MPI_Op:

MPI_MAX: maximum
MPI_MIN: minimum
MPI_SUM: sum
MPI_PROD: product
MPI_LAND: logical and
MPI_BAND: bit-wise and
MPI_LOR: logical or
MPI_BOR: bit-wise or
MPI_LXOR: logical xor
MPI_BXOR: bit-wise xor

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

Parameters: see MPI_Gather

Gathers data from all tasks and distribute the combined data to all tasks

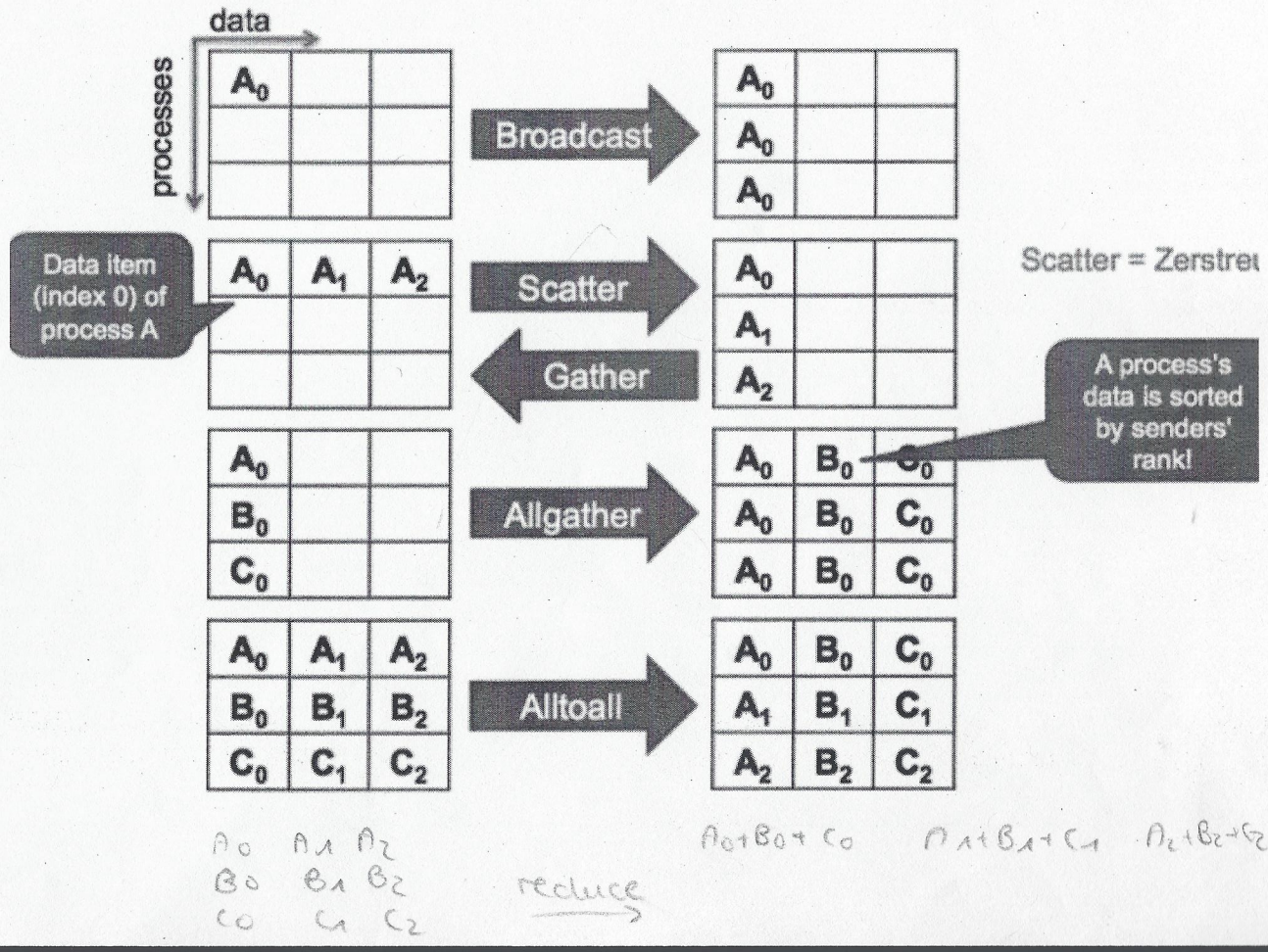
int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

Parameters: see MPI_Allgather

Sends data from all to all processes

MPI_Datatype:

MPI_INT
MPI_DOUBLE
MPI_FLOAT
MPI_CHAR
...



int MPI_Comm_rank(MPI_Comm comm, int *rank)
-> *rank = rank of the calling process in the group of comm (integer)

int MPI_Comm_size(MPI_Comm comm, int *size)
-> *size = number of processes in the group of comm (integer)

MPI_Comm_WORLD
collection of all processes

int MPI_Barrier(MPI_Comm comm)
Blocks the caller until all processes in the communicator (comm) have called it

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
source: source rank, or MPI_ANY_SOURCE (integer)
tag: tag value or MPI_ANY_TAG (integer)
Blocking test for a message

int MPI_Init(int *argc, char ***argv)
argc: Pointer to the number of arguments
argv: Pointer to the argument vector
Initialize the MPI execution environment

int MPI_Finalize()
All processes must call this routine before exiting.

int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
Input Parameters:
buf: address of send buffer
count: number of elements in send buffer
datatype: datatype of each send buffer element
dest: rank of destination
tag: message tag
Performs a blocking send

int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
Input Parameters: see MPI_Send
Output Parameters
request: communication request
Begins a nonblocking send

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
MPI_STATUS_IGNORE
Output Parameters:
buf: initial address of receive buffer
status: status object
Input Parameters:
count: maximum number of elements in receive buffer
source: rank of source (integer)

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
Parameters: see MPI_Isend, MPI_Recv
Begins a nonblocking receive

int MPI_Sendrecv(const 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)
Input Parameters:
sendbuf: initial address of send buffer

sendcount: number of elements in send buffer
sendtype: type of elements in send buffer
dest: rank of destination
sendtag: send tag
recvcount: number of elements in receive buffer
recvtype: type of elements in receive buffer
source: rank of source
recvtag: receive tag

Output Parameters:

recvbuf: initial address of receive buffer
status: status object. This refers to the receive operation.

Sends and receives a message (blocking).

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

Input Parameters:

request: MPI request

Output Parameters:

flag: true if operation completed
status: status object. May be MPI_STATUS_IGNORE.

Tests for the completion of a request

int MPI_Wait(MPI_Request *request, MPI_Status *status)

Input Parameters:

request: request

Output Parameters:

status: status object. May be MPI_STATUS_IGNORE.

Waits for an MPI request to complete

int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,MPI_Comm comm)

Input/Output Parameters:

buffer: starting address of buffer

Input Parameters:

count: number of entries in buffer
datatype: data type of buffer
root: rank of broadcast root

Broadcasts a message from the process with rank "root" to all other processes of the communicator

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

Input Parameters:

sendbuf: starting address of send buffer
sendcount: number of elements in send buffer
sendtype: data type of send buffer elements
recvcount: number of elements for any single receive
recvtype: data type of recv buffer elements
root: rank of receiving process

Output Parameters:

recvbuf: address of receive buffer

Gathers together values from a group of processes

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

Parameters: see MPI_Gather

Sends data from one process to all other processes in a communicator

Haskell

all :: (a -> Bool) -> [a] -> Bool
Applied to a predicate and a list, all determines if all elements of the list satisfy the predicate.

any :: (a -> Bool) -> [a] -> Bool
Applied to a predicate and a list, any determines if any element of the list satisfies the predicate.

const :: a -> b -> a
Constant function.

length :: [a] -> Int
maximum :: Ord a => [a] -> a
minimum :: Ord a => [a] -> a

(!!) :: [a] -> Int -> a
List index (subscript) operator, starting from 0.

elem :: Eq a => a -> [a] -> Bool
notElem :: Eq a => a -> [a] -> Bool
filter :: (a -> Bool) -> [a] -> [a]
filter, applied to a predicate and a list, returns the list of those elements that satisfy the predicate

concat :: [[a]] -> [a]
Concatenate a list of lists.

concatMap :: (a -> [b]) -> [a] -> [b]
Map a function over a list and concatenate the results.

cycle :: [a] -> [a]
cycle ties a finite list into a circular one, or equivalently, the infinite repetition of the original list.

repeat :: a -> [a]
repeat x is an infinite list, with x the value of every element.

replicate :: Int -> a -> [a]
replicate n x is a list of length n with x the value of every element

iterate :: (a -> a) -> a -> [a]
iterate f x returns an infinite list of repeated applications of f to x:
>iterate f x == [x, f x, f (f x), ...]

reverse :: [a] -> [a]
fst :: (a, b) -> a
snd :: (a, b) -> b
splitAt :: Int -> [a] -> ([a], [a])
splitAt n xs returns a tuple xs prefix of length n and second element is the remainder of the list

map :: (a -> b) -> [a] -> [b]
map f xs is the list obtained by applying f to each element of xs,

fmap :: Functor f => (a -> b) -> f a -> f b
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl, applied to a binary operator, a starting value (typically the left-identity of the operator),
and a list, reduces the list using the binary operator, from left to right

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr, applied to a binary operator, a starting value (typically the right-identity of the operator),
and a list, reduces the list using the binary operator, from right to left

sum :: Num a => [a] -> a

scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f z [x1, ...] = [z, z `f` x1, (z `f` x1) `f` x2, ...]

The sum function computes the sum of a finite list of numbers.

null :: [a] -> Bool
Test whether a list is empty.

head :: [a] -> a
last :: [a] -> a
tail :: [a] -> [a]

init :: [a] -> [a]
Return all the elements of a list except the last one. The list must be non-empty.

take :: Int -> [a] -> [a]
take n, applied to a list xs, returns the prefix of xs of length n, or xs itself if n

takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile, applied to a predicate p and a list xs, returns the longest prefix (possibly empty) of xs of elements that satisfy p

drop :: Int -> [a] -> [a]\
drop n xs returns the suffix of xs after the first n elements, or [] if n > length xs

dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p xs returns the suffix remaining after takeWhile p xs

sort :: Ord a => [a] -> [a] (Blatt2)

zip :: [a] -> [b] -> [(a, b)]
zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith generalises zip by zipping with the function given as the first argument, instead of a tupling function.
For example, zipWith (+) is applied to two lists to produce the list of corresponding sums.

unzip :: [(a, b)] -> ([a], [b])
unzip3 :: [(a, b, c)] -> ([a], [b], [c])

maximumBy :: (a -> a -> Ordering) -> [a] -> a (Blatt3)
The maximumBy function takes a comparison function and a list and returns the greatest element of the list by the comparison function.

compare :: Ord a => a -> a -> Ordering
E.g. compare ‘on‘ snd

negate :: Num a => a -> a
abs :: Num a => a -> a
even :: Integral a => a -> Bool
odd :: Integral a => a -> Bool
max :: Ord a => a -> a -> a
min :: Ord a => a -> a -> a

error :: [Char] -> a (Blatt1)
error stops execution and displays an error message.

chr :: Int -> Char (Z.Blatt1)
ord :: Char -> Int (Z.Blatt1)

maybe :: b -> (a -> b) -> Maybe a -> b
data Maybe a = Nothin | Just a

show :: Show a => a -> String

data Powers = Ps [Double]
instance (Num Powers) where