ADVANCED PROGRAMMING PARADIGMS

120min, alle schriftlichen Unterlagen, keine elektronische Geräte

Introduction (1 Woche)

Programming Paradigms

paradigm	theory of ideas about how something should be done (e.g. pattern)
programming paradigm	fundamental style of programming, with explicit aspects (e.g. state, concurrency/parallelism, nondeterm.) e.g. 'see below' and constraint programming, concurrent programming and parallel programming
software quality	• reliability (**correctness**, robustness) • modularity (extendibility / reusability) • compatibility, efficiency, portability, ease of use, timeliness
Multiparadigm	Several paradigms can be combined into a single language ML -> functional with imperative features C# -> object-oriented with functional features F# -> functional with object-oriented features Scala -> functional + object-oriented Curry -> function + logic Curry is based on Haskel
Correctness	program should be correct with respect to its specifications • testing (find faults/bugs) -> choose **input**, **run**, and check **output** • proving (show the absence of faults) -> **no input, nor exec**, but apply **mathematical rules**
Verification	tools for object-oriented programs: Spec#, **Dafny** first step towards program verification: ill-typed expression will not compile (automatic, light-weight)
Example	Theorem: $(a+b)^2 = a^2 + 2ab + b^2$ Es kann mit **endlichen** vielen Schritten gezeigt werden, dass es für **unendlich** viele Werte gilt.
Referential Transparency	LEIBNIZ = substitution of equals for equals = referential transparency -> order has no influence on result
Program transformation	$x = f(a), \quad and, \quad x + x = 2 * x$ $x + x = 2 * x = f(a) + x = x + f(a) = f(a) + f(a) = 2 * f(a)$
Misuse of the Equality Symbol	assignments like $x \coloneqq x + 1$ has not the slightest similarity to equality x becomes/gets/receives $x + 1$, but never x equals/is $x + 1$ ---> a different symbol should be used $\coloneqq$ or $\leftarrow$
Reducible expr	**redex**: e.g. $mult(x, y) = x * y$

Evaluation Strategies

	innermost (call-by-value) prefer leftmost	**outermost** (call-by-name) prefer leftmost	**lazy** (outermost + sharing) work with pointers
Example	$mult(1+2, 2+3)$ $= mult(3, 2+3)$ $= mult(3, 5)$ $= 3 * 5 = 15$	$mult(1+2, 2+3)$ $= (1+2) * (2+3)$ $= 3 * (2+3)$ $= 3 * 5 = 15$	$square(1+2)$ $= (1+2) * (1+2)$ $= 3 * (1+2)$ $= 3 * 3 = 9$
argument evaluated	exactly once	zero or more times	at most once

sharing: keep only a single copy of the argument expression and maintain a pointer to it
whenever there exists an order of evaluation that terminates, outermost (and thus lazy) evaluation finds it

Overview

	imperative	**object-oriented**	**functional**	**logic**
based on	read and update state (e.g. Turing machine)	<-- imperative with support for abstraction and modularization	λ-calculus and reduction (replace by simpler expr)	first-order logic (pedicate logic)
concepts	**data structures** (variable, records, array, pointers) **computations**: • expressions (literal, identifier, operation, function call) • commands (assign, composition, conditional, loop, procedure call) **abstraction:** function/procedure	objects as instances of classes encapsulation (inform. hiding) inheritance for modularity, subtyping, polymorphism, dynamic binding genericity	no state/cmds, but expr. no loops, but recursion functions (recursiv, anonym, curried, higher order), polymorphic overloaded types pattern matching type interface eager or lazy evaluation	logical formulas expr machine solves and programmer guides HORN clauses
examples	Ada, Algo, C, Cobol, Fortran, Modula, Pascal	C++, C#, Eiffel, Java, Objective-C, Simula Smalltalk	F#, Haskell (lazy eval),Lisp, ML (eager eval), OCaml	Prolog
consist of	n-expr: $y \coloneqq 0, \quad a \coloneqq 3$ n-cmds: function $f(x)$ begin $y \coloneqq y + 1$; return $x + y$ end		n-decl: $f(x) = 2 * x + 1$ $a = 3$ 1-expr: $a + f(a)$	
	n-exec: $f(a) + f(a)$ returns $4 + 5 = 9$		1-eval: $3 + f(3) = 10$	
order	no referential transparency		referential transparency	
syntax	expressions (-> yield value) + commands (-> new state)		expressions -> yield value	
semantics	values + environment + state		values + environment	
proving	possible but complicated, use HOARE logic/triple		easy	

Funktionale Programmierung - Programming in Haskell (5 Wochen)

Ch1-Ch3 – Introduction, First Steps, Types and Classes

Functional prog.	Programming style in which the basic method of computation is the **application of functions to arguments**.		
File suffix	.hs		
Compiler	GHC (Glasgow Haskell Compiler) is the leading implementation of Haskell, compiler and interpreter "ghci"		
Interpreter	: (mit Doppelpunkt)		
File/Script	:l FileName // = :load :r // = :reload :? oder :h // = :help	lade ein File reload script (no change detection) show all commands	
Types Uppercase, Typ-safe/error	e :: t // e has type t :t 1+1 // = :type 1+1	type inference -> autom. calculated at compile time show type without evaluating	
	Bool // False or True Char String // = [Char] Int Integer Float, Double	Logical values Single Character Strings of characters Fixed-precision integer Arbitrary-precision integer Floating-point numbers	
show	:set +t :unset +t	Show type in following expressions Hide type in following expressions	
type classes	Eq Show - Read Num Ord // Eq a => Ord Integral // (Num a, Ord a) => Integral Fractional // Num a => Fractional Enum – Bounded – Floating	Equality – all except IO and functions Showable / Readable – all except IO and functions Numeric – Int, Integer, Float, Double Ordered – all except IO and functions Integral – Int, Integer Fractional – Float, Double sequentially ordered – upper/lower bound - floating	
basic functions lower-case	+ - * negate, abs, signum ^ fromInteger / fromRational recip == /= < <= > >= min, max show read sqrt div, quot, rem, mod quotRem, divMod &&, || not	:: Num a => a -> a -> a :: Num a => a -> a :: (Num a, Integral b) => a -> b -> a :: Num a => Integer -> a :: Fractional a => a -> a -> a :: Fractional a => Rational -> a :: Fractional a => a -> a :: Eq a => a -> a -> Bool :: Ord a => a -> a -> Bool :: Ord a => a -> a -> a :: Show a => a -> String :: Read a => String -> a :: Floating a => a -> a :: Integral a => a -> a -> a :: Integral a => a -> a -> (a,a) :: Bool -> Bool -> Bool :: Bool -> Bool	
Cast	2 // Num p => p 2 :: Int // Int 2 :: Float // 2.0 Float (2 + 2) :: Double // 4.0 Double 2.0 // Fract.. p=>p 2.0 :: Int // error (2::Int)+(2::Double) // error [2, 2.0] // Fract.. a=>[a] [2::Float, 2::Double] // error	 No instance for (Fractional Int) arising from literal Couldn't match expected type with actual type Couldn't match expected type with actual type	
Declaration	x = 17 // or “let x = 17”		
List	[1,2,3] // Num a => [a] [False,'a',False] // error [['a'],['b','c']] // [[Char]] [] // []	Declare list, all elements must be from the same type Length not known during compile time list arguments have a 's' suffix empty list	
functions	head [1,2,3,4,5] // 1 head [] // exception	select the first element	:: [a]->a
	tail [1,2,3,4] // [2,3,4] tail [5] // [] tail "x" // "" -> type	remove the first element	:: [a]->[a]

	[1,2,3,4,5] !! 2	// 3	select the nth element	:: [a]->Int->a
	take 3 [1,2,3,4,5]	// [1,2,3]	select the first n elements	:: Int->[a]->[a]
	drop 3 [1,2,3,4,5]	// [4,5]	Remove the first n elements	:: Int->[a]->[a]
	length [1,2,3,4,5]	// 5	length of a list	:: [a]->Int
	sum [1,2,3,4,5]	// 15	sum of a list of numbers	:: Num a=>[a]->a
	product [1,2,3,4,5]	// 120	product of a list of numbers	:: Num a=>[a]->a
	[1,2,3] ++ [4,5]	// [1,2,3,4,5]	Prepend a lists	:: [a]->[a]->[a]
	'h' : "allo"	// "hallo"	prepend element to list	:: a->[a]->[a]
	reverse [1,2,3,4,5]	// [5,4,3,2,1]	Reverse a list	:: [a]->[a]
	init [1..5]	// [1,2,3,4]	remove the last element	:: [a]->[a]
Tuple	(False,'a') (True,['a','b']) (1) ()	// (Bool,Char) // (Bool,[Char]) // =1 // ()	List with different type, fix length during runtime Type of tuple encodes its size	

Functions	**Mathematics**
	f(x)
	f(x,y)
	f(g(x))
	f(x)g(y)
		Java
		f(x)
		f(x,y)
		f(g(x))
		f(a,b)+ c*d
			Haskel
			f x
			f x y // function has higher priority
			f (g x)
			f x * g y	

layout	f :: Int -> Int -- var A f x = x^2		{f :: Int -> Int; f x = x^2} // var B	
define	not :: Bool -> Bool not a = a == False		functions and arguments lowercase a function is a mapping from values of one type to values of another type	
	mult :: Num a => a -> a -> a mult x y = x*y			
	factorial (Enum a, Num a) => a -> a factorial n = product [1..n]			
	add :: Num a => (a, a) -> a add (x,y) = x+y			
	twice :: (t -> t) -> t -> t twice f x = f (f x)			
use	factorial factorial 10 factorial 10 20 add (2,3) ([abs, factorial] !! 1) 3	// error // 3628800 // error // 5 // 6	No instance for (Show (Integer -> Integer)) it :: (Num a, Enum a) => a Non type-variable argument in the constraint attention, takes a tuple as input works because of lazy evaluation	

Curried Functions (default)	add' x y = x + y
	mult (add’ 2 3) 5
	Int -> Int -> Int
	mult x y z
		// Int->(Int->Int)
		
		// Int -> (Int->Int)
		// ((mult x) y) z
			return functions as results
			this allows multiple arguments
			the arrow ‘->’ associates to the right
			natural functions associate to the left	

Polymorphic Functions	length :: [a] -> Int length [False,True] // 2 (a=Bool)		type contains one or more type variables (e.g. a) type variables are lower-case, and usually a,b,c, ...	
Overloaded Functions	(+) :: **Num a =>** a -> a -> a		type contains one or more **class constraints** e.g. Num is for Int and Float	

Layout rule	a = 10
	b = 20 // Good
		a = 10
		 b = 20 // Bad
			declaration must stay on the same column
			implicit grouping	

last value	it			

Ch4 – Defining functions

conditional expr	abs n = **if** n >= 0 **then** n **else** -n // abs (-4) signum n = **if** n < 0 **then** -1 **else if** n == 0 **then** 0 **else** 1 // 'else' is obligate	
Guarded Equations	abs n | n >= 0 = n | **otherwise** = -n	
Pattern matching (separate file)	{not False = True; not True = False}	patterns are matched order
	not :: Bool -> Bool not False = True not _ = False	more efficient (does not evaluate second arg if True) '_' is a wildcard pattern that matches any value
List patterns	[1,2,3,4] // = 1:(2:(3:(4:[]))) adds an element to the start of a list 1:[] // = [1] [1]:[] // = [[1]] [2]:[3]:[] // = [[2],[3]] ([]:[]):[] // = [[[]]]	internally, every non-empty list is constructed by repeated use of operator ":" called "cons" **[] = nil** 1:[2] // ok, [1,2] [1]:[2] // error []:[]:[] // ok, [[],[]]

	head (x:_) = x // head :: [a] -> a tail (_:xs) = xs // tail :: [a] -> [a]	functions on lists use this ":" operator x:xs patterns only match non-empty lists parenthesis due to priority (application over ":")
	f2 [x,y] = (x,y) // f2 [1,2] -> (1,2)	Exception by parameter missmatch
Lambda expressions	$\lambda x \to x + x$ // lambda is written as '\' double x = x + x	nameless function, usefule when defining functions that return functions as result
e.g.	odds n = map (\x -> x*2 + 1) [0..n-1] odds 10 // [1,3,5,7,9,11,13,15,17,19]	maps an anonymus function to a list
Operator Sections	1+2 == (+) 1 2 == (1+) 2 == (+2) 1 (/2)	sections of operation 1+2 halving function
	f x g == x `f` g	change operator from prefix to infix

ch5 – List comprehensions

Comprehension	$\{x^2 \| x \in \{1..5\}\}$	mathematic comprehension notation
Generator	[1..5] // [1,2,3,4,5]	
Lists comprehensions	[x^2 | x <- [1..5]] // [1,4,9,16,25] [(x,y) | x <- [1,2,3], y <- [4,5]]	define new lists based on old ones multiple ones are comma separated, order matters
Dependant Gen.	[(x,y) | x <- [1..3], y <- [**x**..3]]	they are like nested loops
concat	concat :: [[a]] -> [a] concat xss = [x | xs <- xss, x <- xs] concat [[1,2,3],[4,5]] // [1,2,3,4,5]	concatenates a list of lists to one list use dependant generators
guards	[x | x <- [1..9], even x] // [2,4,6,8]	restrict values produced by earlier generators
factors	factors :: Int -> [Int] factors n = [x | x<-[1..n], n`mod`x==0] factors 15 // [1,3,5,15]	factorize a number using list comprehension with guard
prime	prime :: Int -> Bool prime n = factors n == [1,n] prime 15 // False	detect if number is a prime
primes	primes :: Int -> [Int] primes n = [x | x <- [2..n], prime x] primes 30 // [,3,5,7,11,13,17,19,23,29]	list all primes until a number using list comprehension with guard
zip	zip :: [a] -> [b] -> [(a,b)] zip ['a'..'b'][0..] //[('a',0),('b',1)]	maps two lists to a list of pairs
pairs	pairs :: [] -> [(a,a)] pairs xs = zip xs (tail xs) pairs [1,2,3,4] // [(1,2),(2,3),(3,4)]	list of all pairs of adjacent elements from a list
sorted	sorted :: Ord a => [a] => Bool sorted xs = and [x<=y|(x,y)<-pairs xs] sorted [1,2,3,4] // True	check if a list is sorted using pairs
positions	positions :: Eq a => a -> [a] -> [Int] positions x xs = [i | (x',i) <- zip xs [0..], x == x'] positions 0 [1,0,0,1,0] // [1,2,4]	list of all positions of a value in a list
string comprehensions	"ab" :: String // == ['a','b']::[Char] zip "abc" [1,2] // [('a',1),('b',2)]	because a string is a char list any polymorphic function works on strings
count	count :: Char -> String -> Int count x xs = length [x'|x'<-xs,x==x'] count 's' "Mississippi" // 4	counting how many times a character occurs
pyths	pyths :: Int -> [(Int,Int,Int)] pyths z = [(x,y,z) | x<-[1..z], y<-[1..z], x^2+y^2 == z^2]	pythagorean: triple (x,y,z) of positive integers
perfects	perfects :: Integral a => a -> [a] perfects n = [n' | n' <- [1..n], sum (init (factors n')) == n'] perfects 500 // [6,28,496]	factor n', remove last element (init) and sum them, add only if equals n'
scalar product	scalar :: Num a => [a] -> [a] -> [a] scalar a b = [c | i <- [0..length a-1], c <- [a!!i*b!!i]] scalar [2,5,3] [6,4,2] // [12,20,6]	use iterater with length of list a, multiply each element of a and b

Excursion: Implication and Equivalence

implication →	(==>) :: Bool -> Bool -> Bool False ==> _ = True True ==> p = p	define a function ==> which needs two bools when first param is False it returns True when first param is True it returns the second param

equivalence
⇒
	`(<=>) :: Bool -> Bool -> Bool`
	`p <=> q = p == q`
		define a function <=> which needs two bools returns if 'p' is equal to 'q'

check correctness	`verifyImp p q = (p ==> q) <=> (not p || q)`
	`verifyEqu p q = (p <=> q) <=> ((p ==> q) && (q ==> p))`	

	`check verify = and [verify p q | p <- [False, True], q <- [False, True]]`
	`check verifyImp`
	`check verifyEqu`	

Ch6 – Recursive Functions

Recursion	`fac n | n == 0 = 1 | otherwise = n * fac(n-1)`	as guarded equation

	`rev [] = []`
	`rev (x:xs) = rev xs ++ [x]`
		as pattern matching

on lists	`product :: Num a => [a] -> a`
	`product [] = 1`
	`product (n:ns) = n * product ns`
		multiply each element of a list

	`length :: [a] -> Int`
	`lenght [] = 0`
	`length (_:xs) = 1 + length xs`
		length of a list

	`reverse :: [a] -> [a]`
	`reverse [] = []`
	`reverse [x:xs] = reverse xs ++ [x]`
		reverse a list

multiple args	`zip :: [a] -> [b] -> [(a,b)]`
	`zip [] _ = []`
	`zip _ [] = []`
	`zip (x:xs) (y:ys) = (x,y) : zip xs ys`
		zipping the elements of two lists

drop	`drop :: Int -> [a] -> [a]`
	`drop 0 xs = xs`
	`drop _ [] = []`
	`drop n (_:xs) = drop (n-1) xs`
		remove the first n elements from a list

append	`(++) :: [a] -> [a] -> [a]`
	`[] ++ ys = ys`
	`(x:xs) ++ ys = x : (xs ++ ys)`
		append two lists

Quicksort	`qsort :: Ord a => [a] -> [a]`
	`qsort [] = []`
	`qsort (x:xs) = qsort smaller ++[x]++ qsort larger`
	`    where`
	`        smaller = [a | a <- xs, a <= x]`
	`        larger =  [b | b <- xs, b > x]`
		split array by head element and sort

and	`and :: [Bool] -> Bool`
	`and [] = True`
	`and (x:xs) = x && and xs`
		logica and using recursion

concat	`concat :: [[a]] -> [a]`
	`concat [] = []`
	`concat (x:xs) = x ++ concat xs`
		concat a list of lists to a list

replicate	`replicate :: Int -> a -> [a]`
	`replicate 0 x = []`
	`replicate n x = x : replicate (n-1) x`
		adds an element n times to a list

select	`(!!) :: [a] -> Int -> a`
	`(x:xs) !! 0 = x`
	`(x:xs) !! n = xs !! (n-1)`
		select the n-th element of a list

elem	`elem :: Eq a => a -> [a] -> Bool`
	`elem y [] = False`
	`elem y (x:xs) = if x == y then True else elem y xs`
		check if a list contains an element

merge	`merge :: Ord a => [a] -> [a] -> [a]`
	`merge [] [] = []`
	`merge xs [] = xs`
	`merge [] ys = ys`
	`merge (x:xs) (y:ys) = if x < y then x : merge xs (y:ys) else y : merge (x:xs) ys`	

msort	`msort :: Ord a => [a] -> [a]`
	`msort [] = []`
	`msort xs = merge (qsort(take (length xs `div` 2) xs))`
	`                 (qsort(drop (length xs `div` 2) xs))`	

Ch7 – High-order functions

higher-order		taking a function as an argument or returning a function as a result

twice	`twice :: (a -> a) -> a -> a`
	`twice f x = f (f x)`
		takes function as input

map	`map :: (a -> b) -> [a] -> [b]`
	`map f xs = [f x | x <- xs] // list compreh.`
	`map f (x:xs) = f x : map f xs // recursion`
	`map (+1) [1,3,5,7] // [2,4,6,8]`
		apply a function to every element of a list

filter	`filter :: (a -> Bool) -> [a] -> [a]`
	`filter p xs = [x | x <- xs, p x]`
	`filter even [1..10] // [2,4,6,8,10]`
		selects every element from a list,
		that satisfies a predicate

foldr	`foldr :: (a -> b -> b) -> b -> [a] -> b`
	`foldr f v [] = v`
	`foldr f v (x:xs) = f x (foldr f v xs)`
		f maps the empty list to some value v,
		and non-empty list to some function f applied to its head and foldr of its tail

e.g.	`sum = foldr (+) 0`
	`product = foldr (*) 1`
	`or = foldr (||) False`
	`and = foldr (&&) True`
	`length = foldr (λ_ n -> 1+n) 0`
	`reverse = foldr (λx xs -> xs ++ [x]) []`
	`(++ ys) = foldr (:) ys`
		it is defined with recursion,
		but it is best to think of non-recursive.
		replace each (:) in a list with a given function, and [] with a value

composition	`(.) :: (b -> c) -> (a -> b) -> (a -> c)`
	`f . g = λx -> f (g x) // f after g`
	`map((*2).(+1)) [1,2,3] // [4,6,8]`
	`compiler = codeGen.typeChecker.parser.scanner`
		two functions composite to one

e.g.	`odd :: Int -> Bool`
	`odd = not . even`	

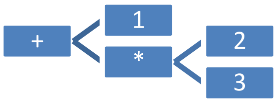
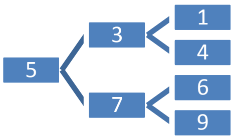
all	`all :: (a -> Bool) -> [a] -> Bool`
	`all p xs = and [p x | x <- xs]`
	`all even [2,4,6,8] // True`
		decide if every element of a list satisfies a given predicate p

any	`any :: (a -> Bool) -> [a] -> Bool`
	`any p xs = or [p x | x <- xs]`
	`any (== ' ') "abc def" // True`
		decide if at least one element of a list satisfies a predicate

takeWhile	`takeWhile :: (a -> Bool) -> [a] -> [a]`
	`takeWhile p [] = []`
	`takeWhile p (x:xs)`
	`    | p x = x:takeWhile p xs`
	`    | otherwise = []`
	`takeWhile (/= ' ') "abc def" // "abc"`
		selects elements from a list while a predicate holds of all the elements

dropWhile	`dropWhile :: (a -> Bool) -> [a] -> [a]`
	`dropWhile p [] = []`
	`dropWhile p (x:xs)`
	`    | p x = dropWhile p xs`
	`    | otherwise = x:xs`
	`dropWhile (== ' ') "   abc   " // "abc   "`
		selects elements from a list while a predicate holds of all the elements

Ch8 – Declaring Types and Classes

type declaration	type String = [Char] type Pos = (Int,Int)	String is an array of Chars
e.g.	origin :: Pos origin = (0,0)	defines the origin
	left :: Pos -> Pos left (x,y) = (x-1,y) left origin // (-1,0)	move position one to the left
with params	type Pair a = (a,a) mult :: Pair Int -> Int mult (m,n) = m*n	
	copy :: a -> Pair a copy x = (x,x)	
nested	type Trans = Pos -> Pos	can be nested
~~recursive~~	~~type Tree = (Int,[Tree])~~	cannot be recursive
data declaration (new type, like an enum)	data Answer = Yes | No | Unknown answers :: [Answer] answers = [Yes,No,Unknown]	Answer is the new type Yes, No and Unknown are data constructors both must start with upper-case letter
function	flip :: Answer -> Answer flip Yes = No flip No = Yes flip Unknown = Unknown	
with params	data Shape = Circle Float | Rect Float Float	like functions: Rect::Float->Shape
	square :: Float -> Shape square n = Rect n n area :: Shape -> Float area (Circle r) = pi * r^2 area (Rect x y) = x * y	
with type params	data Maybe a = Nothing | Just a safediv :: Int -> Int -> Maybe Int safediv _ 0 = Nothing safediv m n = Just (m `div` n)	
	safehead :: [a] -> Maybe a safehead [] = Nothing safehead xs = Just (head xs)	
recusive types	data Nat = Zero | Succ Nat	natural numbers
convert to	nat2int :: Nat -> Int nat2int Zero = 0 nat2int (Succ n) = 1 + nat2int n	convert our type to Int using recursion
convert from	int2nat :: Int -> Nat int2nat 0 = Zero int2nat n = Succ (int2nat (n-1))	convert Int to our type using recursion
function	add :: Nat -> Nat -> Nat add Zero n = n add (Succ m) n = Succ (add m n)	avoid conversion with function add
arithmetic expressions	data Expr = Val Int | Add Expr Expr | Mul Expr Expr	
eval	eval :: Expr -> Int eval (Val n) = n eval (Add x y) = eval x + eval y eval (Mul x y) = eval x * eval y eval (Add (Val 1) (Mul (Val 2) (Val 3))) // 7	evaluate an arithmetic expression
Binary Trees two-way-branching structure	data Tree a = Leaf a | Node (Tree a) a (Tree a) t :: Tree Int t = Node (Node (Leaf 1) 3 (Leaf 4)) 5 (Node (Leaf 6) 7 (Leaf 9))	
occurs	occurs :: Eq a => a -> Tree a -> Bool occurs x (Leaf y) = x == y occurs x (Node l y r) = x == y || occurs x l || occurs x r	

flatten	`flatten :: Tree a -> [a]`
	`flatten (Leaf x) = [x]`
	`flatten (Node l x r) = flatten 1`
	`                      ++ [x]`
	`                      ++ flatten r`	

Ch9 – The Countdown problem

	kein Prüfungsstoff	

Ch10 – Interactive programming

Until know: **New (impure):**	input -> program -> output input+keyboard -> program -> output+screen	pure functions (no side effects) interactive programs (with side effects)

Input/Output	`IO Char`
	`IO () // tuples with no component`
		the type of actions that return a character
		the type of purely side effecting actions (no result)

actions	`getChar :: IO Char`	reads a character from the keyboard, echoes it to the screen an returns it
	`putChar :: Char -> IO ()`	writes a character c to the screen and returns no value
	`return :: a -> IO a`	returns the value without any interaction
exec action	evaluating an action executes its side effects, with the final result value being discarded	
Sequencing	combine actions	

e.g.	`act :: IO (Char,Char)`
	`act = do x <- getChar`
	`         getChar   --ignored`
	`         y <- getChar`
	`         return (x,y)`
	`act`
	`1 3 // -> (1,3)`
		«do» ist syntaktischer Zucker für ">>=" (bind)
		
		liest drei character,
		auch möglich: "`_ <- getChar`"

getLine	`getLine :: IO String`
	`getLine = do x <- getChar`
	`             if x == '\n' then`
	`               return []`
	`             else`
	`               do xs <- getLine`
	`                  return (x:xs)`	

putStr	`putStr :: String -> IO ()`
	`putStr [] = return ()`
	`putStr (x:xs) = do putChar x`
	`                   putStr xs`
	`putStr “hello world\n”`
		write a string to the screen

putStrLn	`putStrLn :: String -> IO ()`
	`putStrLn xs = do putStr xs`
	`                 putChar '\n'`
	`putStrLn “hello world”`
		write a string and move to a new line

strLen	`strLen :: IO ()`
	`strLen = do putStr "Enter a string: "`
	`            xs <- getLine`
	`            putStr "The string has "`
	`            putStr (show (length xs))`
	`            putStrLn " characters"`
	`strLen // Enter a string:`
	`Hello  // The string has 5 characters`
		prompt for a string to be entered and display it length

Programmverifikation (4 Wochen)

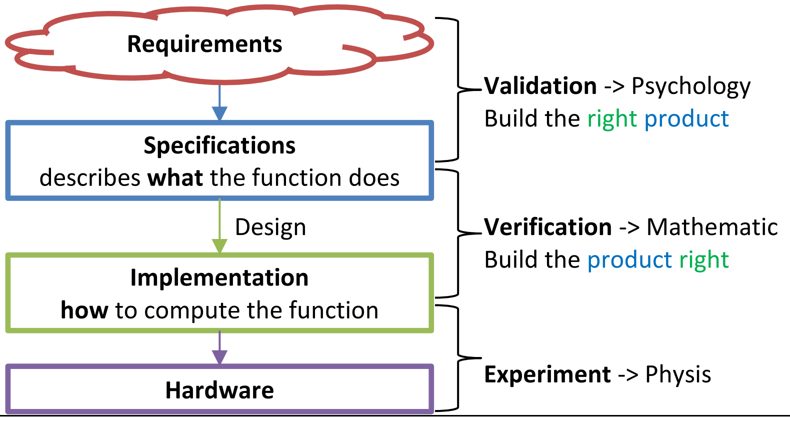
Problem of Erroneous Software

produce high cost disclaimer instead of guarantee	**types**: unspectacular, but many errors (e.g. office) seldom, but spectacular errors (e.g. ariane, intel pentium, airport of denver)

Software Qualities

Reliabilty: **correctness**, robustness
Dependability: knowing that software is reliable, certification
Correctness is the most important of all software qualities, and it is only sensible against a specification

Overview


Requirements
Specifications
describes **what** the function does
Design
Implementation
how to compute the function
Hardware

Validation -> Psychology
Build the right product

Verification -> Mathematic
Build the product right

Experiment -> Physis

Testing
Execute program with chosen input. Output consistent with specification.
show the **presence of bugs**,
but never to show their absence!

Proving
program will not be executed,
show the **absence of bugs**

Consistency: Testing + Proving

IML

Imperative Mini Language – consist of preconditions, postconditions and commands

Assertions "Zusicherung"

should yield always true. If it does not, the program is in error.
Assert statements are a simple yet powerful possibility to check assertions at run time.

bool expr + state

boolean expression + state -> true or false
boolean expression + true -> set of states

Implication

		Implication	**Contrapositive**	
$\boldsymbol{A}$	$\boldsymbol{B}$	$\boldsymbol{A \Rightarrow B} \equiv \neg A \vee B$	$\boldsymbol{\neg B \Rightarrow \neg A}$	
True	True	True	True	ok
True	False	False	False	not ok
False	True	True	True	ex falso quodlibet
False	False	True	True	(from false what you like)
Example		If I win, I'll eat my hat	I can't eat my hat, I can't win	

Hoare Triple

Syntax
$\{P\}\ C\ \{Q\}$
P: assertion (precondition) of hoare triple
C: command
Q: assertion (postcondition) of hoare triple
prestate = state before execution
poststate = state after execution

Example
$\{x > 5\}\ x \coloneqq x + 1\ \{x > 6\}$

Properties
$\infty\ loop \rightarrow ok$
P stronger than Q

Validity vs Truth

valid: true in all states	$\vDash x + 5 = 5 + x$	$s_0(x) = 0 \rightarrow true$ $s_1(x) = 1 \rightarrow true$
valid Hoare triple $\vDash \{P\}\ C\ \{Q\}$	$\vDash \{x > 5\}\ x \coloneqq x + 1\ \{x > 6\}$	
not valid: not true in all states	$\nvDash x + 5 = y$	$s_0(x) = 3, s_0(y) = 8 \rightarrow true$ $s_1(x) = 3, s_0(y) = 7 \rightarrow false$
non-valid Hoare triple $\nvDash \{P\}\ C\ \{Q\}$	$\nvDash \{x = 5\}\ x \coloneqq x + 1\ \{x = 17\}$	

Partial correct

if the program ever terminates, then the result is correct.
-> a program that does not "crash" but produces a wrong result is generally by far more dangerous.

total correct

the program is partial correct and will terminate.

specification of imperative program

Syntax:
$\{P\}\ x \coloneqq ?\ \{Q\}$

precondition P
List x of variables that might be changed, others are forbidden to change
postcondition Q

Rigid variables

variables for specifications, do not occur in program
also called ghost variables or local variable

$\{x = \boldsymbol{X}\}\ x \coloneqq ?\ \{x = \boldsymbol{X} + 6\}$
$old(x)$ refers to x in the prestate

WP: weakest preconditions $wp(C, Q)$

set of all prestates in which execution of **C terminates in a poststate satisfying Q**, or in which execution of **C does not terminate**.

Theorem: start with the postcondition to arrive at the precondition
$\vDash \{P\}\ C\ \{Q\}\ is\ equivalent\ to\ \vDash P \Longrightarrow wp(C, Q)$

Example:
$C: x \coloneqq x + 1$
$Q: x > 5$
$wp(C, Q) = x > 4$

Program Verification

prove that a Hoare triple $\{P\}\ C\ \{Q\}$ is valid
from the back to the front -> looks strange, but simpler

usually long and boring -> automatically but proof problem is undecidable in general

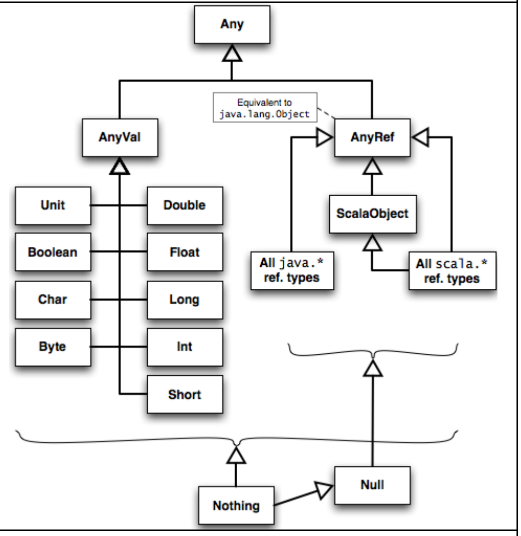
Inference Rules	let $f, f_1, \dots, f_n$ be boolean formulas -> here assertions or hoard triples, $n \ge 0$ $\frac{f_1, \dots, f_n}{f} \rightarrow \frac{premises\ or\ hypotheses}{conclusion}\ (kein\ Bruch)$ the interference rule is correct, if the validity of the conclusion follows from the validity (premises or hypotheses) $if\ n = 0\ then\ we\ call\ it\ axiom$	C: Commands C_t, C_e: cmd's P,Q,R,S: assert E: Expression id: variable B: bool expr

	Theorem	n	**Example**
Skip Axiom	$\overline{\{P\}\ skip\ \{P\}}$	0	$\vDash \{x > 5\}\ skip\ \{x > 5\}$
Skip WP	$P \in wp(skip, P)$		
Assignment Axiom	$\overline{\{P[id \leftarrow E]\}\ id := E\ \{P\}}$	0	$\vDash \{(x+1) = 5\}\ x := x + 1\ \{x = 5\}$ (Textual Substitution: replace id with E)
Assignment WP	$defined(E) \wedge P[id \leftarrow E] \in wp(id := E, P)$ ensure that E is defined in the prestate		$wp\left(z := \frac{y}{x-1}, z \ge 1\right) = x - 1 \ne 0 \wedge \frac{y}{x-1} \ge 1$
Rule of Consequence	$\frac{P \Rightarrow Q,\ \{Q\}C\{R\},\ R \Rightarrow S}{\{P\}C\{S\}}$		$\frac{\vDash x > 6 \Rightarrow x > 5, \vDash \{x > 5\}skip\{x > 5\}, \vDash x > 5 \Rightarrow x > 5}{\Rightarrow \{x > 6\}skip\{x > 5\}}$ (interface between Hoare logic and ordinary math)
Composition Rule	$\frac{\{P_0\}C_1\{P_1\}, \dots, \{P_{n-1}\}C_n\{P_n\}}{\{P_0\}C_1; \dots; C_n\{P_n\}}$	≥ 2	$\vDash \{y = A \wedge x = B\}\ h := x\ \{y = A \wedge h = B\}$ $\vDash \{y = A \wedge h = B\}\ x := y\ \{x = A \wedge h = B\}$ $\vDash \{x = A \wedge h = B\}\ y := h\ \{x = A \wedge y = B\}$ $\vDash \{y = A \wedge x = B\}\ h := x; x := y; y := h\ \{x = A \wedge y = B\}$
Composition WP	$P_0 \in wp(C_1; \dots; C_n, P_n)$	≥ 2	strange swap: $\vDash\ -x := x - y;\ x := x + y;\ x := y - x\ \{-x = A \wedge y = B\}$
Conditional Rule	$\frac{\{P \wedge B\}C_t\{Q\}, \{P \wedge \neg B\}C_e\{Q\}}{\{P\}\ if\ B\ then\ C_t else\ C_e\ endif\ \{Q\}}$		$if(\ x \le\ y\ )then\ skip\ else\ h := x;\ x := y;\ y := h;\ endif$ $\vDash \{true\}\ C\ \{x \le\ y\}$
Conditional WP	$P_t \in wp(C_t, Q)$ $P_e \in wp(C_e, Q)$ $(P_t \wedge B) \vee (P_e \wedge \neg B) \in$ $wp(if\ B\ then\ C_t\ else\ C_e endif, Q)$ $(B \Rightarrow P_t) \wedge (\neg B \Rightarrow P_e) \in$ $wp(if\ B\ then\ C_t\ else\ C_e endif, Q)$		
Invariants	$\vDash \{I\}\ C\ \{I\}$		$x - y = \Delta;\ x := x + 1;\ y := y + 1$
Invariants of a loop	$\frac{\{I \wedge B\}\ C\ \{I\}}{\{I\}\ while\ B\ do\ C\ endwhile\ \{I\ \wedge \neg B\}}$		$while\ i > 0\ do\ i := i - 1\ endwhile$ Invariant: $i \ge 0$
Loop **Loop with init**	$\vDash \{P\}\ while\ B\ do\ C\ endwhile\ \{Q\}$ $\vDash \{P\}\ C_{ini};\ while\ B\ do\ C_{rep}\ endwhile\ \{Q\}$		
WP of Loop	too complicated		

Proof Procedure	prove if $\{P\}\ C\ \{Q\}$ is valid 1. compute the weakest precondition $wp(C, Q)$ 2. determine the **verification condition (VC)** $P \Rightarrow wp(C, Q)$ automatically done by **verification condition generator** 3. prove the verification condition valid (Discharging) automatically discharged by an **automated theorem prover**	Example: $\{x > 6\}\ skip\ \{x > 5\}$ 1. $wp(skip, x > 5) = x > 5$ 2. $x > 6 \Rightarrow x > 5$ 3. prove that VC valid
Proof Outline	a program annotated with assertions (as comments) between each pair of commands	
Annotated program	a program annotated with assertions (as comments or assert-commands) for purposes of documentation	
good practise	“enough assertions should be inserted to make the program understandable, but not so many that the program is hidden from view.”	

Multiparadigmen- und stark getypte Programmierung (4 Wochen)

Scala – Multiparadigm Language

Multiparadigm	Multi-paradigm programming (functional and object-oriented) Objects and Syntax from Java/C++/C/Smalltalk/Simula, Functional programming von Haskel/ML/Lisp, Actors von Erlang, Pattern matching von Prolog	
Properties	Object-oriented, Functional, type safe, performant, agile, lightweight syntax	
Java (anno domini)	**Pro**: popularity, acceptance, object-riented and strong typing, library, tools, JVM (platform independent) **Cons**: Very imperative, not highly concurrent, verbose (to much boilerplate code) -> source code generator	
based on JVM	Clojure, Groovy, JRuby, Jython, Scala, Kotlin, Ceylon	
Scala Pro over Java (anno domini)	functions are classes and can be passed around, values are objects (pure object-oriented), operators are just methods, statically typed (as Java) but uses type inference, supports the principle of uniform access, supports concurrency, is concise (short and precise)	
Example	`class Person (val name: String, var age: Int)`	
Pro	Scala is scalable and extensible	
Types	**Byte -> Short -> Int -> Long -> Float -> Double**. **Char:** assignment compatibility Char -> Int **Boolean** **Unit** (void), only one value "()" **Null:** subtype of all reference types, only instance is null **Nothing:** bottom type, is a subtype of all types, no instance **Lists** (concrete classes, no interface, linked, immutable) **String** (lot of methods, interpolation, multiline) **Tuples** (fixed size, different types, access is 1-based) **Maps** (pairs, immutable -> mutable variants exist) **Any** = Scala base type (isInstanceOf, asInstanceOf) **AnyRef** = root of all reference (equals, eq, hascode, ...) **AnyVal** = root of all values types have methods (5.toFloat) operators are method calls (10 ./(3)	 Any Equivalent to java.lang.Object AnyVal AnyRef Unit Double ScalaObject Boolean Float All java.* ref. types All scala.* ref. types Char Long Byte Int Short Nothing Null

Variable decl	**val** (const, final)
	var
		`val year = 1989`
		`var age = 27; age = 28`

Control expr	**if**, no ternary "?"
	while
	do-loop (expr of type Unit)
	for-comprehension (expr of type Unit or first generator)
		`val res = if(false) { println(1) } else 2 // 2`
		`val res = while (age > 10) age -= 1 // ()`
		`val res = do age -= 1 while ( age > 10 ) // ()`
		`val res = for(i <- 1 to 10 if i%2==0) yield (i*i)`
		`// Vector(4, 16, 36, 64, 100)`

Classes	abstract, final, single inheritance, nested classes
	
	members: values (var or val), methods (def),
	types (type)
	-> default visibility is public
	
	every class has a primary constructor
		`class CreditCard(val numb: Int, var limit: Int) {`
		`  def this(numb: Int) = this(numb, 1000) // aux cons`
		`  println("new card created") // exec in primary cons`
		`  private var sum = 0`
		`  def buy(amount: Int) {`
		`  if(sum + amount > limit) throw new RuntimeException`
		`    sum += amount`
		`  }`
		`  def remainder = limit-sum // method without param`
		`}`
		`val a = new CreditCard(2000);`

Inheritance abstract class	doSmth() must be overwritten	`abstract class Base(param: String) {`
		`  def doSmth: String    // without body abstract`
		`  override def toString() = "^"+ super.toString()`
		`}`
		`class Derived extends Base("0") {`
		`  def doSmth = "working"`
		`}`

Methods	similar to Java, default val
	multiple returns with Tuples
	param called by name
	curried param list
		`def add(x: Int, y: Int = 1): Int = {return x+y}`
		`def quorem(m: Int, n: Int) : (Int, Int) = (m/n, m%n)`
		`quorem(n = 2, m = 4)`
		`def sub(m: Int)(n: Int) = m-n; println(sub(2){5})`

Singleton Objects	can be accessed by its name
	Name represents the single instance
	Singleton can be passed to functions with parameter ColorFactory.type
		`class Color(val r: Int, val g: Int, val b: Int)`
		`object ColorFactory{`
		`  private val cols = Map(`
		`    "red" -> new Color(255,0,0),`
		`    "blue" -> new Color(0,0,255),`
		`    "green" -> new Color(0,255,0))`
		`  def getColor(color: String) =`
		`    if(cols contains color) cols(color) else null }`
		`val c = ColorFactory.getColor("red")`

Companion Objects	similar to "friends" in C++ classes and companion objects can access private fields	`class Color private (val r:Int, val g:Int, val b:Int)`
		`object Color {`
		`  def getColor() = new Color(255,0,0)`
		`}`

Functions	are instance of class FunctionX
	X=(0..22)
	curried
	
	tuppled
	curried definition
	type interference
	subclass of FunctionX
		`val add = (m: Int, n: Int) => m + n     add(2,3) //5`
		`                                        add.apply(2,3)//5`
		`val addc = add.curried`
		`val inc = addc(1)                       inc(5) //6`
		`val addt = add.tupled                   addt(2 -> 3) //5`
		`val add1 = (x: Int) => (y: Int) => x+y //curried`
		`val add2 : Int => Int => Int = (x:Int) => (y:Int) => x+y`
		`object add extends Function2[Int, Int, Int] {`
		`    def apply(x: Int, y: Int) = x+y  }`

Pattern Matching	similar to switch-case
	Match expression, No fall-through
	throws error if no pattern matches
		`def patternMatching(i : Int) = {`
		`  i match {`
		`    case 0 => "Null"`
		`    case 1 => "One"`
		`    case _ => "?"`

	with types and guards
	
	can match lists, tuples
		`def patternMatching(any : Any) =`
		`  any match {`
		`    case i : Int => "Int: " + i`
		`    case s : String => "String: " + s`
		`    case d : Double if d > 0 => "Pos Double: " + d`
		`    case List() => "empty list"`
		`    case any => any.toString  }`

	matching lists	`def length(list: List[Any]) : Int= {`
		`  list match {`
		`  case List() => 0`
		`  case x :: xs=> 1 + length(xs)  }  }`

	matching tuples	`def process(input: Any) = {`
		`  input match {`
		`    case (a,b) => printf("Processing (%d,%d)...\n", a, b)`
		`    case "done" => println("done")`
		`    case _ => null  }  }`

Case classes	**new** is not mandatory
	getter are automatically defined
	equals(), hasCode(), toString()
	decompe with pattern matching
		`abstract class Tree`
		`case class Sum(x: Tree, y: Tree) extends Tree`
		`case class Prod(x: Tree, y: Tree) extends Tree`
		`case class Var(n: String) extends Tree`
		`case class Const(v: Int) extends Tree`

		`def eval(t: Tree, env: Map[String,Int]) : Int = t match {`
		`  case Sum(x, y) => eval(x, env) + eval(y, env)`
		`  case Prod(x, y) => eval(x, env) * eval(y, env)`
		`  case Var(n) => env(n)`
		`  case Const(v) => v  }`

Scala Traits

Multiple inheritance	Unterscheiden zwischen: Interface Inheritance oder Code Inheritance Works fine when you combine classes that have nothing in common. Problem with multiple inherited methods (solved in C# (explicit interface implem.), not in C++ or Java) Problem with diamond inheritance problem (use virtual inheritance in C++). Java: Single implementation inheritance, multiple interface inheritance -> duplicated code

Traits in Scala

analogous to Java interfaces, but with implementations and fields and dynamic composition
Solution for the diamond problem and linearization problem
e.g. «TraitA with TraitB with TraitC» is the data type
super.log invokes next trait in the trait hierarchy (stackable modifications) and not the base class
traits can not be instantiated, but can be added to new objects

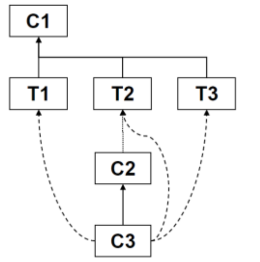
Self types

```
trait ExceptionLoggerextends Logger {
    this: Exception =>
    def log() { log(getMessage()) }
}
```

In the trait methods, any methods of the self type can be invoked
a trait with a self type is similar to a trait with a supertype

Linearization

1. actual type as first
2. add right to left
3. remove duplicates left to right
4. append AnyRef and Any



```
class C3 extends C2 with T1 w. T2 w. T3
```

$$\mathcal{L}(C_3) = C_3 + \underbrace{\mathcal{L}(T_3)}_{T_3C_1} + \underbrace{\mathcal{L}(T_2)}_{T_2C_1} + \underbrace{\mathcal{L}(T_1)}_{T_1C_1} + \underbrace{\mathcal{L}(C_2)}_{C_2\ \underbrace{T_2}_{T_2C_1}}$$

$$\mathcal{L}(C_3) = C_3T_3T_1C_2T_2C_1 + AnyRef + Any$$

$$\rightarrow supercall$$
$$\leftarrow init$$

```
class C1 { print(">>C1"); // constr
  def m = List("C1")
}
trait T1 extends C1 { print(">>T1")
  override def m = { "T1" :: super.m }
}
trait T2 extends C1 { print(">>T2")
  override def m = { "T2" :: super.m }
}
trait T3 extends C1 { print(">>T3")
  override def m = { "T3" :: super.m }
}
class C2 extends T2 { print(">>C2")
  override def m = { "C2" :: super.m }
}
class C3 extends C2 with T1 with T2 with T3 {
  print(">>C3")
  override def m = { "C3" :: super.m }
}
```

```
val a  = new C1 // >>C1
print(a.m)      // List(C1)

val a  = new C3
// >>C1>>T2>>C2>>T1>>T3>>C3
print(a.m)
// List(C3, T3, T1, C2, T2, C1)

a.isInstanceOf[C1 with T2] //true
a.isInstanceOf[T1 with T3] //true
```

???

```
def ??? : Nothing = throw new NotImplementedError
```

mark methods to be implemented

Parameterized Types

Type parameters

```
// on classes or traits
case class Pair[T, S](val first: T, val second: S)
val p1 = Pair(42, "String")
// on functions or methods
def getMiddle[T](a: Array[T]) = a(a.length / 2)
```

change override

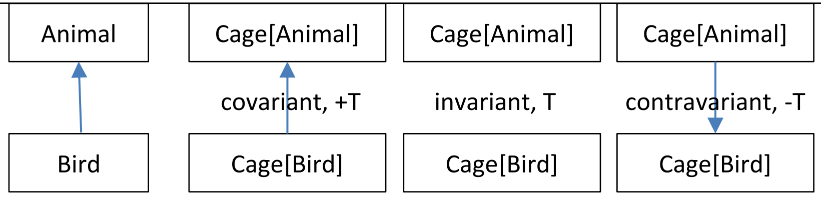
überschriebene Methoden dürfen mehr liefern als verlangt (return type) -> covariant (erlaubt in Java)
überschriebene Methoden dürfen weniger erwarten als verlangt (params) -> contravariants (nicht in Java)

mutable
immutable

likely to be changed
unable to change

Java: use-site declaration (?super, ?extend)
Scala: declaration-site declaration (+/-T)

Variance of Subtyping



invariant is default

```
class Animal; class Bird extends Animal;
class Cage[A]   // invariant (default)
class Cage1[+A] // covariant, C#: out
class Cage2[-A] // contravariant, C#: in
val animalCage: Cage1[Animal] = new Cage1[Bird] // allowed when [+A]
val birdCage: Cage2[Bird] = new Cage2[Animal] // allowed when [-A]
```

Implicit Converions

implicit functions	`class BlingString(string: String) { def bling = "*" + string + "*" }`
	`implicit def blingToString(s: String) = new BlingString(s)`
	`print("Hello".bling) // *Hello*`

	Method bling is now available on all strings (as if it were defined in class String)

implicit classes	`implicit class BlingString(string: String) { def bling = "*" + string + "*" }`
	`print("Hello".bling) // *Hello*`

usages	`val f: Fraction = 12 // type differs from expected type`
	`"hello".bling // non-existent member access`
	`3 * Fraction(4,5) // Int.* does not accept a Fraction arg`

rules	No implicit conversions if the code compiles without it The compiler will NEVER attempt multiple conversions Ambiguous conversions are an error implicit conversion must be in scope

implicit parameters	`case class Delimiters(left: String, right: String)`
	`def quote(text: String)(implicit delims: Delimiters) =`
	`   delims.left + text + delims.right`
	`print(quote("Bonjour")(Delimiters("«", "»"))) // «Bonjour»`
	`//quote("Hello") - error: could not find implicit value for parameter delims`

	only works for the last parameter list
type classes	most powerful features in Haskell They allow you to define generic interfaces that provide a common feature set over a wide variety of types. Type classes define a group (class) of types which satisfy some contract (defined in a trait).

	`trait Monoid[A] {`
	`  def op(x: A, y: A) : A`
	`  def unit : A`
	`}`
	`implicit object stringMonoid extends Monoid[String] {`
	`  def op(x: String, y: String) = x + y`
	`  def unit = ""`
	`}`
	`implicit object addMonoid extends Monoid[Int] {`
	`  def op(x: Int, y: Int) = x + y`
	`  def unit = 0`
	`}`