4.0 TYPEN & TYPKLASSEN

Typensignaturen:
Num kann alle mögliche Zahlentypen sein (Int, Integer, Doblue, Factorial etc.)

Funktionstyp:
```
data (->) a b ; a ist Name des Datentyps uns b Konstruktor Bsp: data Color = Red | Green | Blue
(data TypeName = Constructor1 | Constructor2 | ... | ConstructorN)
infixr 0 `(->)`; rechtsassoziativ a -> b -> c -> d = a -> (b -> (c -> d))
```

Typconstraints: a muss dem Datentypen bei jeder möglichen Operation entsprechen.
```
Num a => a -> a -> a (+,-,*)
Fractional a => a -> a -> a (/)
```
Jedoch muss die Ausgabe nicht a sein: Foldable f=> a -> f a -> Bool
mehrere Constraints werden geklammert: (Num a, Num b) => a -> b -> b; (Ord a, Num a) => a -> a -> Ordering

Typconstraints und konkrete Typen:
```
fifteen = 15 -- :: Num a => a
fifteenInt = fifteen :: Int
fifteenDouble = fifteen :: Double
fifteenDouble + fifteen -- ok
fifteenInt + fifteen -- ok
fifteenInt + fifteenDouble -- !!!Typfehler
```

Currying: Currying ist das Verschachteln mehrerer Funktionen mit einem Parameter und erzeugt die Illusion, dass die definierte Funktion mehrere Parameter hätte
```
add :: Num a => a -> a -> a
add :: Num a => a -> (a -> a)
```

Uncurrying: eine nicht currysierte Funktion bekommt alle Argumente in ein Tupel verpackt
```
addUncurry :: Num a => (a, a) -> a
addUncurry (x,y) = x + y

curry :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a, b) -> c
addSeven = curry addUncurry 7
```
curry zerlegt die addUncarry Funktion dass es für die carry-Art passt

Sectioning;
```
(^2); (2<); (++"!!!")
isBetween2And12 = (`elem` [2..12])
```

Polymorphismus:
parametric polymorphism, auch ad-hoc
```
id :: a -> a
id x = x
```
oder
```
class Show a where
    show :: a -> String
```
constrained polymorphism, über Typklassen realisiert, z.B.
```
elem :: Eq a => a -> [a] -> Bool
```

Brauche Double, biete Int
```
fromIntegral :: (Num b, Integral a) => a -> b
```
erlaubt dass Rechung von Int und Double überhaupt stattfindet, Num ist Ergebnis

```
13 / fromIntegral (length [1..8]); length [1..8] = 8
```

Typklassen:
Eq
- Ermöglicht das Testen auf Gleichheit und Ungleichheit (`==` und `/=`).
- Beispiel: `5 == 5` ist `True`.

Ord:
   - Bietet Vergleichsoperatoren (`<`, `<=`, `>`, `>=`) für geordnete Typen.
   - Beispiel: `compare 3 4` gibt `LT` zurück.

3. Show:
   - Ermöglicht die Konvertierung von Werten in Zeichenketten (`show`).
   - Beispiel: `show 123` ergibt `"123"`.

4. Read:
   - Ermöglicht die Konvertierung von Zeichenketten in Werte (`read`).
   - Beispiel: `read "123" :: Int` ergibt `123`.

5. Num:
   - Beinhaltet grundlegende arithmetische Operationen (`+`, `-`, `*`, `abs`, `signum`, `fromInteger`).
   - Beispiel: `5 + 3` ergibt `8`.

6. Integral:
   - Beinhaltet ganzzahlige Operationen (`div`, `mod`).
   - Beispiel: `div 10 3` ergibt `3`.

7. Fractional:
   - Beinhaltet Bruchoperationen (`/`, `fromRational`).
   - Beispiel: `10 / 3` ergibt `3.3333`.

8. Functor:
   - Definiert die `fmap`-Funktion, die eine Funktion auf die Inhalte eines Funktors anwendet.
   - Beispiel: `fmap (+1) [1, 2, 3]` ergibt `[2, 3, 4]`.

9. Applicative:
   - Erweitert `Functor` und ermöglicht das Anwenden von Funktionen, die in Funktoren eingeschlossen sind (`<*>`).
   - Beispiel: `pure (+) <*> Just 3 <*> Just 4` ergibt `Just 7`.

10. Monad:
    - Erweitert `Applicative` und bietet `>>=` (Bind-Operator) für Sequenzierung von Berechnungen.
    - Beispiel: `Just 3 >>= \x -> Just (x + 4)` ergibt `Just 7`.

11. Foldable:
    - Bietet die Möglichkeit, Datenstrukturen zu falten (reduktion) (`foldr`, `foldl`).
    - Beispiel: `foldr (+) 0 [1, 2, 3]` ergibt `6`.

Instanzieren von Typklassen:
Das Instanziieren von benutzerdefinierten Datentypen mit Typklassen ermöglicht es, die Funktionen und Methoden dieser Typklassen zu definieren und zu nutzen.

```
data Tree a = Empty | Node a (Tree a) (Tree a)
  deriving (Eq, Show)
```

Eq und Show werden automatisch durch deriving abgeleitet. Dadurch können wir Bäume vergleichen und als Zeichenketten darstellen
Instanz für Functor: erlaubt es uns, eine Funktion auf alle Elemente im Baum anzuwenden.

```
instance Functor Tree where
    fmap _ Empty = Empty
    fmap f (Node x left right) = Node (f x) (fmap f left) (fmap f right)
```

Instanz für Foldable: erlaubt es uns, den Baum zu falten (reduzieren)

```
instance Foldable Tree where
    foldMap _ Empty = mempty
    foldMap f (Node x left right) = foldMap f left `mappend` f x `mappend` foldMap f right
    foldr _ z Empty = z
    foldr f z (Node x left right) = foldr f (f x (foldr f z right)) left

data DayOfWeek =
Mon | Tue | Weds | Thu | Fri | Sat | Sun
instance Eq DayOfWeek where
Mon == Mon = True
Tue == Tue = True
Weds == Weds = True
Thu == Thu = True
Fri == Fri = True
Sat == Sat = True
Sun == Sun = True
_ == _ = False

type DayOfMonth = Int
data Date = Date DayOfWeek DayOfMonth
instance Eq Date where
    (Date weekday dayOfMonth) ==
      (Date weekday' dayOfMonth')
         = weekday == weekday'
            && dayOfMonth == dayOfMonth'
```

Instanzen für Typklassen mit Parametern:

```
data Identity a = Identity a
instance Eq (Identity a) where
   (==) (Identity v) (Identity v') = v == v'
```

noch besser wegen Vermeidung von Typfehler:

```
instance Eq a => Eq (Identity a) where
(==) (Identity v) (Identity v') = v == v'
```

5.0 FUNKTIONALE MUSTER

Pattern Matching:
switch-case in C-ähnlichen Sprachen

```
data StudyCourse = DC | IC | IF
studyCourseName :: StudyCourse -> String
studyCourseName DC = "Data Science & Scientific Computing"
studyCourseName IC = "Scientific Computing"
studyCourseName IF = "Informatik"
```

Mit Tupeln:
```
f (a, b) (c, d) = ((b, d), (a, c))
fst3 :: (a, b, c) -> a
fst3 (x, _, _) = x
```
Pattern Matching im Ausdruck – case:
```
data StudyCourse = DC | IC | IF
studyCourseName :: StudyCourse -> String
studyCourseName course = "Bachelor "
    ++ case course of
        DC -> "Data Science & Scientific Computing"
        IC -> "Scientific Computing"
        IF -> "Informatik"
```

Guards/Wächter:

Funktionen höherer Ordnung(HOF):
Funktionen, die Funktionen als Parameter oder als Ergebnis haben
```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
Prelude> map show [1..10]
Prelude> map (\x -> if x<3 then "zu klein" else show x) [1..5]
["zu klein","zu klein","3","4","5"]
```

Funktionskomposition:
```
negate . sum $ [1, 2, 3, 4, 5] statt negate (sum [1, 2, 3, 4, 5])
f = \x -> (negate . sum) x statt f x = negate $ sum x
punktfrei : print :: Show a => a -> IO ()
```

Rekursion:
```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Nothing nicht genug:
```
data Either a b = Left a | Right b
safeDiv :: Integer -> Integer -> Either String Integer
safeDiv x 0 = Left "divide by zero"
safeDiv x y = Right $ x `div` y
```

6. FALTEN
```
foldr (+) 0 [1..5] – 15
scanr (+) 0 [1..5] – [15,14,12,9,5,0]
foldl (+) 0 [1..5] – 15
scanl (+) 0 [1..5] – [0,1,3,6,10,15]
```

7.0 ALGEBRAISCHE DATENSTRUKTUREN
Typen und Kinds:Typen klassifizieren Werte
Bool, Int und (Int, Float) “fertige” Typen
z.B. [] oder Maybe sind noch nicht “fertig”
um Regeln zu definieren, wie Typen zusammengebaut werden, brauchen wir eine
Klassifizierung von Typen: Kinds
```
k Int :: * => fertiger Typconstraints
k (Maybe Bool) :: *
k Maybe :: * -> * => kein fertiger Typkonstruktor
k [] :: * -> *
k Either :: * -> * -> *
k Maybe Maybe :: ERROR 2 nicht fertige Typkonstruktor,Expecting one more argument to ‘Maybe’
k Maybe (Maybe Int) :: *
```

Datenkonstruktoren:
```
data Point = Point Int Int
:t Point => Point :: Int -> Int -> Point
:t Point 3 => Point 3 :: Int -> Point
:t Point 3 5 => Point 3 5 :: Point
map (uncurry Point) [ (x,y) | x <- [1..2], y <- [1..2]] => [Point 1 1,Point 1 2,Point 2 1,Point 2 2]
map (`Point` 12) [1..3] => [Point 1 12,Point 2 12,Point 3 12]
```

Getter:
```
getX (Point x _) = x
getY (Point _ y) = y
```

Setter: Da wird nichts gesetzt => NEUER Punkt!
```
setX (Point xOld y) x = Point x y
setY (Point x yOld) y = Point x y
```

Record-Syntax:
```
data Point = Point { x :: Int, y :: Int}
```
Compiler generiert authomatisch: x :: Point -> Int; y :: Point -> Int
```
p = Point 5 3
```

```
x p => 5
q = p { y = 7 }
q => Point {x = 5, y = 7}

type vs. data:
type erzeugt Typsynonym (Alias), für einen bestehenden Typ
type Grade = Int : jede Funktion die auf einen Int angewendet werden, auch auf einen Grade
angewendet werden
type Point = (Int, Int)

data erzeugt neuen (komplexen) Datentyp
data Shape = Circle Float | Rectangle Float Float: nur speziell für Shape definierte Funktionen
können auf einen Grade angewendet werden

Instanzen für data:
data Person = Person { name :: String, age :: Int }
instance Eq Person where
    (Person name1 age1) == (Person name2 age2) = name1 == name2 && age1 == age2

type: Typ-Aliase selbst können keine Instanzen für Typklassen haben, aber du kannst die Instanzen
für den zugrunde liegenden Typ verwenden.

newtype: newtype erschafft einen neuen Typ, wie data, hat aber zur Laufzeit keinen Overhead, wie
type
Nachteil: newtype kann nur einen Datenkonstruktor mit einem Parameter haben
Vorteil: der Typchecker überprüft den Typ mit Grade, aber zur Laufzeit bleibt nur noch der Int übrig
newtype PositiveInt = PositiveInt Int
instance Show PositiveInt where
    show (PositiveInt n) = "PositiveInt " ++ show n
instance Show Grade where
   show (Grade 1) = "sehr gut"...show _ = "ungültige Note"

Polymorphe Funktionen:
newtype Students = Students { headcount :: Int }
newtype Staff = Staff { headcount :: Int }
!!! FEHLER: Multiple declarations of ‘headcount’
möglich:
newtype Students =
Students { headcountStudents :: Int }
newtype Staff = Staff { headcountStaff :: Int }
besser:
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
class Headcount a where
   headcount :: a -> Int
instance Headcount Int where
   headcount i = i
newtype Students = Students Int
   deriving (Headcount)
newtype Staff = Staff Int
   deriving (Headcount)
```

8. MONOIDE

```
gleichartige Strukturen sollten abstrahiert werden
Eine Halbgruppe: ein Datentyp mit einer binären Operation
class Semigroup a where
  (<>) :: a -> a -> a

Monoid: Datentyp, binäre, assoziative Operation, mit einer Identität
class Semigroup a => Monoid a where
   mempty :: a
   mappend :: a -> a -> a
   mappend = (<>)
   mconcat :: [a] -> a
   mconcat = foldr mappend mempty
   {-# MINIMAL mempty #-}
mconcat [[1..3], [4..6]] => [1,2,3,4,5,6]
foldr mappend mempty [[1..3], [4..6]] => [1,2,3,4,5,6]

Maybe: auch Maybe a ist Instanz von Monoid, wenn der Typ der darin “eingepackt” ist, ein Monoid is
Just "Hallo" <> Just " " <> Just "Welt" => Just "Hallo Welt"

Beispiel für Semigroup- und Monoid-Instanz
instance Semigroup StudentGroups where
   (StudentGroups sizeX xs) <>
     (StudentGroups sizeY ys) =
   mkStudentGroups (max sizeX sizeY)
      (Students $ sum xs + sum ys)
instance Monoid StudentGroups where
mempty = StudentGroups 0 []
```

```
Monoidgesetze:
Linksidentität mempty <> x === x
Rechtsidentitä x <> mempty === x
Assoziativität x <> (y <> z) === (x <> y) <> z

Monoid-Instanz, aber kein Monoid!:
instance Monoid StudentGroups where
   mempty = StudentGroups 1 []
 (StudentGroups sizeX xs) `mappend`
   (StudentGroups sizeY ys) =
   mkStudentGroup sizeX
   (Students (sum xs + sum ys))
```

9. FUNKTOREN: Ein Argument pro Funktion

```
mit Hilfe eines Funktors die Funktion f auf x anwenden und bekommen einen Wert vom Typ “verpacktes
b” zurück
instance Functor [] where
   fmap = map
(<$>) = fmap
fmap (+1) [1..5] => [2,3,4,5,6]
fmap show $ Just 17 => Just "17"
fmap show Nothing => Nothing
show <$> Just 17 => Just "17"
fmap (fmap (+1)) $ fmap Just [1..3] => [Just 2,Just 3,Just 4]
```

10. APPLIKATIVE FUNKTOREN: mehrere Argumente möglich

```
pure
pure hat die Typensignatur pure :: a -> f a.
Es nimmt einen Wert vom Typ a und verpackt ihn in einen kontextabhängigen Typ f a.
Es ist im Wesentlichen eine Möglichkeit, einen "normalen" Wert in einen Kontext zu heben.
class Functor f => Applicative f where
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
(<$>) = fmap
Prelude> [(+1), (*2)] <*> [1..3]
[2,3,4,2,4,6]
Prelude> [(+),(*)] <*> [1..3] <*> [5,6]
[6,7,7,8,8,9,5,6,10,12,15,18]
Prelude> concat [[x+y,x*y] | x<-[1..3], y<-[5,6]]
[6,5,7,6,7,10,8,12,8,15,9,18]
Gesetze für applikative Funktoren
• Identität
pure id <*> v === v
• Komposition
pure (.) <*> u <*> v <*> w ===
u <*> (v <*> w)
• Homomorphismus
pure f <*> pure x === pure (f x)
• (spezielle Art von) Kommutativität
u <*> pure y === pure ($ y) <*> u
```

11. Monaden

```
lass Applicative m => Monad m where
(>>=) :: m a -> (a -> m b) -> m b -- heisst bind
(>>) :: m a -> m b -> m b -- heisst then
return :: a -> m a -- analog zu pure
fail :: String -> m a

Vergleich mit Funktoren:
Prelude> fmap (+1) [1..3]
[2,3,4]
Prelude> [1..3] >>= return . (+1)
[2,3,4]

Keine veränderliche Variablen!
Folgendes sieht aus als ob s ein neuer Wert zugewiesen wird
f = do
   s <- getLine
   putStrLn s
   s <- getLine
   putStrLn s
aber in Wirklichkeit wird s durch eine neue gebundene Variable mit selben Namen verdeckt (Klammern
nicht notwendig, nur zur Veranschaulichung)
f = getLine >>= (\s ->
   putStrLn s >>
   getLine >>= (\s ->
   putStrLn s))
```

Monaden Gesetze
• Identität
```
m >>= return === m
return x >>= f === f x
```
• Assoziativität
```
(m >>= f) >>= g === m >>= (\x -> f x >>= g)
```

State-Monade
Beispiel ohne Monade: Stack
```
type Stack = [Int]
pop' :: Stack -> (Int, Stack)
pop' (x:xs) = (x, xs)
push' :: Int -> Stack -> ((), Stack)
push' a xs = ((), a : xs)
stackManip :: Stack -> (Int, Stack)
stackManip stack =
let ((), newStack1) = push' 3 stack
(a, newStack2) = pop' newStack1
in pop' newStack2
```

Beispiel: Stack mit der State-Monade
```
import Control.Monad.State
pop :: State Stack Int
pop = state $ \(x:xs) -> (x, xs)
push :: Int -> State Stack ()
push a = state $ \xs -> ((), a : xs)
stackManip' :: State Stack Int
stackManip' = do
push 3
pop
pop
> runState stackManip' [5,8,2,1]
(5,[8,2,1])
```