HASKELL 1: Kurze Programmieraufgaben in Haskell

panitz

Zusammenfassung

Dieser Kurs besteht aus 25 kleine Programmieraufgaben in der Programmiersprache Haskell. Erste einfache Funktionen. Funktionen für einen Listendatentyp. Funktionen auf den Standardlistendatentyp und Funktionen für Binärbäume.

Frage: Fakultät

Schreiben Sie eine **rekursive** Fakultätsmethode. Haskell

```
      module Fac where
fac :: (Ord t, Num t) => t -> t
fac _ = 42
```

Haskell

Lösung

Haskell

```haskell
        module Fac where
fac :: (Ord t, Num t) => t -> t
fac x
  |x<=0 = 1
  |otherwise = x*fac(x-1)
```

Haskell

Erläuterung

Es braucht eine Abbruchbedingung (hier: `x<=0`) und einen Rekursionsfall (hier: `fac(x-1)`).

Frage: Quersumme

Schreiben Sie eine Funktion, die die mehrfache Quersumme realisiert. Es soll nur eine einstellige Zahl als Ergebnis haben. Die Quersumme von 89, sei also nicht 17, sondern 8. Haskell

```haskell
      module Quer where
quersumme :: Integral t => t -> t
quersumme _ = 42
```

Haskell

Lösung

Haskell

```haskell
        module Quer where
quersumme :: Integral t => t -> t
quersumme x
  |x<10 = x
  |otherwise = quersumme ((x `mod` 10) +quersumme (x `div` 10 ))
```

Haskell

Erläuterung

Die letzte Ziffer lässt sich mit der Modulorechnung `mod` ermitteln, die übrigen Ziffern durch ganzzahlige Division `div`. Ist das Ergebnis nicht kleiner 10, ist die Quersumme weiter zu bilden.

Frage: Länge einer Listenstruktur

Schreiben Sie für die Listenstruktur `Li` eine Längenfunktion, die die Anzahl der Elemente berechnet.
Haskell

```haskell
      module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

laenge :: Num t => Li t1 -> t
laenge _ = 0
```

Haskell

Lösung

Haskell

```haskell
        module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

laenge :: Num t => Li t1 -> t
laenge Nil = 0
laenge (_:>xs) = 1+ (laenge xs)
```

Haskell

Erläuterung

Man kann mit *pattern matching* die Fälle unterscheiden. Die leere Liste hat die Länge 0. Ansonsten ist das Ergebnis eins länger als die Länge der `tail`-Liste.

Frage: In Listenstruktur enthalten

Schreiben Sie für die Listenstruktur `Li` eine Funktion, die testet, ob ein bestimmtes Element in der Liste enthalten ist. Haskell

```haskell
      module Li where
infixr 5 :>
data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

enthaelt :: Eq t => t -> Li t -> Bool
enthaelt o _ = False
```

Haskell

Lösung

Haskell

```haskell
        module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

enthaelt :: Eq t => t -> Li t -> Bool
enthaelt o Nil = False
enthaelt o (x:>xs) = o==x || enthaelt o xs
```

Haskell

Erläuterung

Fallunterscheidung mit *pattern matching*. Die leere Liste enthält kein Element. Ansonsten kann das erste Element mit dem gesuchten Element verglichen werden, oder rekursiv weiter in der Restliste gesucht werden.

Frage: Aneinanderhängen einer Listenstruktur

Schreiben Sie für die Listenstruktur `Li` eine Funktion, die zwei Listen aneinander hängt und so zu einer verbindet. Die Funktion sei als Infix-Operation `<+>` realisiert. Haskell

```haskell
      module Li where
infixr 5 :>
infixl 4 <+>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

(<+>) :: Li a -> Li a -> Li a
_ <+> _ = Nil
```

Haskell

Lösung

Haskell

```haskell
        module Li where
infixr 5 :>
infixl 4 <+>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

(<+>) :: Li a -> Li a -> Li a
xs <+> Nil = xs
Nil <+> xs = xs
(x:>xs)<+>ys = x:>(xs<+>ys)
```

Haskell

Erläuterung

Ist eine der beiden Listen leer, so ist das Ergebnis die jeweils andere Liste. Ansonsten kann die Rekursion für den *tail* der ersten Liste aufgerufen werden und auf das Ergebnis noch das erste Element der ersten Liste vorne angefügt werden.

Frage: Anwendung auf eine Listenstruktur

Schreiben Sie für die Listenstruktur `Li` eine Funktion, die durch Anwendung einer Funktion auf alle Listenelemente eine neue Liste erzeugt. Haskell

```haskell
      module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

anwendung :: (t -> a) -> Li t -> Li a
anwendung _ xs = Nil
```

Haskell

Lösung

Haskell

```haskell
        module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

anwendung _ Nil = Nil
anwendung f (x:>xs) = f x :> anwendung f xs
```

Haskell

Erläuterung

Für die leere Liste ist das Ergebnis auch eine leere Liste. Ansonsten ist die als Argument übergebene Funktion auf das erste Element anzuwenden und der rekursive Aufruf auf die *tail*-Liste durchzuführen.

Frage: Letzte Element einer Listenstruktur

Schreiben Sie für die Listenstruktur `Li` eine Funktion, die das letzte Element extrahiert. Das Ergebnis sei vom Standarddatentyp `Maybe` und sei `Nothing` sollte die Liste leer sein. Haskell

```haskell
      module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

letzte :: Li a -> Maybe a
letzte _ = Nothing
```

Haskell

Lösung

Haskell

```haskell
        module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

letzte :: Li a -> Maybe a
letzte Nil = Nothing
letzte (x:>Nil) = Just x
letzte (_:>xs) = letzte xs
```

Haskell

Erläuterung

Für die leere Liste gibt es kein letztes Element. Für einelementige Listen, ist das erste auch das letzte Element. Die Fälle lassen sich mit *pattern matching* unterscheiden. at die Liste mehr als ein Element, so kann rekursiv auf die Restliste verfahren werden.

Frage: Ersten Elemente einer Listenstruktur

Schreiben Sie für die Listenstruktur `Li` eine Funktion, die die Teilliste aus den ersten `n` Elementen erzeugt. Ist `n` größer als die Länge der Liste, so besteht das Ergebnis aus allen Elementen. Haskell

```haskell
      module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

nimm :: (Eq t, Num t) => t -> Li a -> Li a
nimm n _ = Nil
```

Haskell

Lösung

Haskell

```haskell
        module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

nimm :: (Eq t, Num t) => t -> Li a -> Li a
nimm 0 _ = Nil
nimm _ Nil = Nil
nimm n (x:>xs) = x:>nimm (n-1) xs
```

Haskell

Erläuterung

Zwei Abbruchbedingungen lassen sich über *pattern matching* definieren. Der Fall wenn `n` gleich 0 ist, also das Ergebnis keine ELement enthalten soll und als zweite Abbruchbedingung, dass die Liste kein Element enthält. Ansonsten wird eine Liste mit dem ersten Element und dem rekursiven Aufruf erzeugt.

Frage: Ohne die ersten Elemente einer Listenstruktur

Schreiben Sie für die Listenstruktur `Li` eine Funktion, die die Liste ohne die ersten `n` Elemente als Ergebnis hat. Ist `n` größer der Länge der Liste, dann ist das Ergebnis die leere Liste. Haskell

```haskell
      module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

fallenlassen :: (Eq t, Num t) => t -> Li a -> Li a
fallenlassen _ _ = Nil
```

Haskell

Lösung

Haskell

```haskell
        module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

fallenlassen :: (Eq t, Num t) => t -> Li a -> Li a
fallenlassen 0 xs = xs
fallenlassen _ Nil = Nil
fallenlassen n (_:>xs) = fallenlassen (n-1) xs
```

Haskell

Erläuterung

Zwei Abbruchbedingungen lassen sich über *pattern matching* definieren. Der Fall wenn `n` gleich 0 ist, also das Ergebnis die gesamte Liste sein soll und als zweite Abbruchbedingung, dass die Liste kein Element enthält. Ansonsten wird eine Liste mit dem rekursiven Aufruf erzeugt.

Frage: Test auf Präfix einer Listenstruktur

Schreiben Sie für die Listenstruktur `Li` eine Funktion, die testet ob das erste Argument der Anfang des zweiten Arguments ist. Haskell

```haskell
      module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

istPraefix :: Eq a => Li a -> Li a -> Bool
istPraefix _ _ = False
```

Haskell

Lösung

Haskell

```haskell
        module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

istPraefix :: Eq a => Li a -> Li a -> Bool
istPraefix Nil _ = True
istPraefix _ Nil = False
istPraefix (x:>xs)(y:>ys) = x==y && istPraefix xs ys
```

Haskell

Erläuterung

Ist das erste Argument die leere Liste, so ist die Aussage wahr. Ist ansonsten das zweite Argument die leere Liste, so ist die Aussage falsch. Ansonsten müssen die jeweils ersten Elemete gleich sein und der rekursive Aufruf auf die beiden Restlisten wahr ergeben.

Frage: Faltung auf einer Listenstruktur

Schreiben Sie für die Listenstruktur `Li` eine Funktion, die ausgehend von einem initialen Startwert, alle Elemente der Liste mit einer Operatorfunktion verknüpft. Haskell

```haskell
      module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

falte :: a -> (a -> b -> a) -> Li b -> a
falte start _ _ = start
```

Haskell

Lösung

Haskell

```haskell
        module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

falte :: a -> (a -> b -> a) -> Li b -> a
falte start _ Nil = start
falte start op (x:>xs) = falte (start `op` x) op xs
```

Haskell

Erläuterung

Für eine leere Liste wird der Startwert als Ergebnis genommen. Ansonsten wird der Startwert durch die übergebene Operatorfunktion mit dem ersten Element verknüpft und das Ergebnis als Startwert für den rekursiven Aufruf auf die Restliste genommen.

Frage: Löschen in einer Listenstruktur

Schreiben Sie für die Listenstruktur `Li` eine Funktion, die eine Liste für eine Liste erzeugt, in der alle Auftreten eines bestimmten Elements gelöscht sind. Haskell

```haskell
      module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

loesche :: Eq a => a -> Li a -> Li a
loesche _ _ = Nil
```

Haskell

Lösung

Haskell

```haskell
        module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

loesche :: Eq a => a -> Li a -> Li a
loesche _ Nil = Nil
loesche o (x:>xs)
  |o==x = loesche o xs
  |otherwise = x:>loesche o xs
```

Haskell

Erläuterung

Aus einer leeren Liste ist nichts mehr zu löschen. Ansonsten wird das erste Element geprüft, ob es gleich dem zu löschenden Element ist. Wenn ja wird das Ergebnis aus dem rekursiven Aufruf auf die Restliste ermittelt. Wenn nein, dann kommt das erste Element noch zur Ergebnisliste hinzu.

Frage: Einfügen in eine Listenstruktur

Schreiben Sie für die Listenstruktur `Li` eine Funktion, die eine neue Liste durch einfügen eines Elements an einer bestimmten Position erzeugt. Die Position 0 ist das erste Element der neuen Liste. Ist die Position größer als die Länge der Liste, so wird das Element das letzte Element der neuen Ergebnisliste.
Haskell

```haskell
      module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

einfuegen :: (Eq n, Num n) => n -> t -> Li t -> Li t
einfuegen _  _ xs = xs
```

Haskell

Lösung

Haskell

```haskell
        module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

einfuegen :: (Eq t1, Num t1) => t1 -> t -> Li t -> Li t
einfuegen 0 o xs = o:>xs
einfuegen _ o Nil  = o:>Nil
einfuegen n o (x:>xs)  = x:>einfuegen (n-1) o xs
```

Haskell

Erläuterung

Terminierende Fälle sind die Position 0 und die leere Liste. Beide Fälle sind durch eigene Gleichungen im *pattern matching* definierbar. Ansonsten wird das erste Element für die Ergebnisliste übernommen und ein rekursiver Aufruf mit $n-1$ und der Restliste definiert.

Frage: Umdrehen einer Listenstruktur

Schreiben Sie für die Listenstruktur `Li` eine Funktion, die die Elemete in umgekehrter Reihenfolge in eine neue Liste einfügt.. Haskell

```
      module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

umdrehen :: Li a -> Li a
umdrehen xs = xs
```

Haskell

Lösung

Haskell

```haskell
        module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

umdrehen :: Li a -> Li a
umdrehen = aux Nil
  where
    aux res Nil = res
    aux res (x:>xs) = aux (x:>res) xs
```

Haskell

Erläuterung

Es wird eine akkumulierende Hilfsfunktion aufgereufen. Das erste Argument dieser ist das akkumulierende Ergebnis. Schrittweise werden die Elemente in dieser von der Liste auf das erste Argument umschiftet.

Frage: Filter auf einer Listenstruktur

Schreiben Sie für die Listenstruktur `Li` eine Funktion, die die Liste aller Elemente erzeugt, für die ein bestimmtes Prädikat wahr ergibt. Haskell

```haskell
      module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

filtere :: (t -> Bool) -> Li t -> Li t
filtere _ xs = xs
```

Haskell

Lösung

Haskell

```
        module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

filtere :: (t -> Bool) -> Li t -> Li t
filtere _ Nil = Nil
filtere p (x:>xs)
  |p x = x :> filtere p xs
  |otherwise =  filtere p xs
```

Haskell

Erläuterung

Für eine leere Liste ist auch das Ergebnis leer. Ansonsten wird mit der Prädikatfunktion getestet, ob das erste Element in die Ergebnisliste kommt oder nicht.

Frage: Aufsplitten einer Listenstruktur nach einer Eigenschaft

Schreiben Sie für die Listenstruktur `Li` eine Funktion, die ein Paar aus zwei Ergebnisliste erstellt. In der erste Listes des Paares sind alle Elemente der Argumentliste, für die eine Prädikatsfunktion wahr ergibt, in der zweiten alle die, für die die Prädikatsfunktion nicht wahr ergibt. Haskell

```haskell
      module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

teilen :: (t -> Bool) -> Li t -> (Li t,Li t)
teilen _ xs = (Nil,Nil)
```

Haskell

Lösung

Haskell

```haskell
        module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

teilen :: (t -> Bool) -> Li t -> (Li t,Li t)
teilen p Nil  = (Nil,Nil)
teilen p (x:>xs)
  |p x = (x:>yes,nos)
  |otherwise = (yes,x:>nos)
  where
   (yes,nos) = teilen p xs
```

Haskell

Erläuterung

Für die leere Liste wird das Paar von zwei leeren Listen erzeugt. Ansonsten wird der rekursive Aufruf auf die Restliste vorgenommen und geschaut, an welche der beiden Listen des Ergebnisses das erste Element vorne anzuhängen ist.

Frage: Insertion Sort

Schreiben Sie für die Listenstruktur `Li` eine Funktion, die Sortierung per Einfügen in eine Ergebnisliste vor nimmt. Schreiben Sie hierzu eine Hilfsfunktion mit akkumulierenden Ergebnisargument und eine Funktion zum Einfügen von Elementen. Haskell

```haskell
      module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

einfuegeSortierung :: Ord t => Li t -> Li t
einfuegeSortierung xs = xs
```

Haskell

Lösung

Haskell

```haskell
        module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

einfuegeSortierung :: Ord t => Li t -> Li t
einfuegeSortierung  = es Nil
  where
    es res Nil = res
    es res (x:>xs) = es (einfuegen res x) xs

    einfuegen Nil x = x:>Nil
    einfuegen xs@(y:>ys) x
      |x<y       = x:>xs
      |otherwise = y:>einfuegen ys x
```

Haskell

Erläuterung

Die lokale Einfügefunktion geht die Liste durch, bis ein Element gefunden wird, das größer ist als das einzufügende Element. Die akkumulierende Funktion entspricht eigentlich einer Faltung.

Frage: Bubble Sort

Schreiben Sie für die Listenstruktur `Li` eine Funktion, die für den *bubble sort* Algorithmus einmal durch die Liste geht und jeweils zwei nebeneinander liegende Elemente vertauscht, wenn Sie gemäß der Ordnung in falscher Reihenfolge sind. Das bool'sche Argument zeigt an, ob bereits eine Vertauschung stattgefunden hat. Das Ergenis ist ein paar aus einem bool'schen Wert, der anzeigt, ob es eine Vertauschung gab und der Ergebnisliste. Haskell

```haskell
      module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

blubberSortierung :: Ord t => Li t -> Li t
blubberSortierung xs
  |ready = geblubber
  |otherwise = blubberSortierung geblubber
  where
    (ready,geblubber) = blubber True xs

    blubber :: Ord t => Bool -> Li t -> (Bool, Li t)
    blubber r xs = (r,xs)
```

Haskell

Lösung

Haskell

```haskell
        module Li where
infixr 5 :>

data Li a = Nil
  |(:>) a (Li a) deriving (Show,Eq)

blubberSortierung xs
  |ready = geblubber
  |otherwise = blubberSortierung geblubber
  where
    (ready,geblubber) = blubber True xs
    blubber r Nil = (r,Nil)
    blubber r xs@(x:>Nil) = (r,xs)
    blubber r ys@(x1:>x2:>xs)
      |x1>x2 = (r2,x2:>bs2)
      |otherwise = (r3,x1:>bs3)
      where
        (r2,bs2) = blubber False (x1:>xs)
        (r3,bs3) = blubber r (x2:>xs)
```

Haskell

Erläuterung

Bei Listen mit mindestens zwei Elementen wird geschaut, ob die ersten beiden Elemente in falscher Reihenfolge sind. Es wird dann nach eventueller Tauschung die rekursion auf die Restliste angewendet.

Frage: Längstes Listenelement

Schreiben Sie für eine Standardliste von Elementen, die eine Länge haben, eine Funktion, die das längste Element in der Argumentliste selektiert. Haskell

```haskell
      module Li where
laengstesElement :: Foldable t => [t a] -> t a
laengstesElement xss = head xss
```

Haskell

Lösung

Haskell

```haskell
module Li where

laengstesElement :: Foldable t => [t a] -> t a
laengstesElement (xs:xss)
  = foldl (\x y-> if (length x >= length y) then x else y) xs xss
```

Haskell

Erläuterung

Die Lösung verwendet die standard Faltungsfunktion. Gefaltet wird über die Operatorfunktion, die von zwei Elementen das mit der größeren Länge selektiert.

Frage: Jedes zweite Listenelement

Schreiben Sie für eine Standardliste eine Funktion, die die Liste aus jedem zweiten Element der Argumentliste erzeugt. Haskell

```haskell
module Li where
jedesZweite :: [a] -> [a]
jedesZweite xs = xs
```

Haskell

Lösung

Haskell

```haskell
module Li where

jedesZweite :: [a] -> [a]
jedesZweite [] = []
jedesZweite xs@[x] = xs
jedesZweite (x1:x2:xs) = x1:jedesZweite xs
```

Haskell

Erläuterung

Das *pattern matching* unterscheidet leere Listen, einelementige Listen und Listen von mindestens zwei Elementen. Im letzten Fall erfolgt ein rekursiver Aufruf.

Frage: Verdoppelung jedes Listenelements

Schreiben Sie für die Standardlisten eine Funktion, die jedes Element in der Liste ein zweites mal einfügt. Aus `[1,5,432,9]` soll also `[1,1,5,5,432,432,9,9]` werden. Haskell

```haskell
      module Li where
doppelAlle :: [a] -> [a]
doppelAlle xs = xs
```

Haskell

Lösung

Haskell

```haskell
module Li where

doppelAlle :: [a] -> [a]
doppelAlle = concatMap (\x->[x,x])
```

Haskell

Erläuterung

Die Lösung verwendet die Standardfunktion `concatMap`, die erst alle Element zu einer zweielementigen Liste macht, und diese Teillisten dann aneinander fügt.

Frage: Element eines Baumes

Schreiben Sie für eine binäre Baumstruktur eine Funktion, die die Elemente in *inorder* in einer Liste zurück gibt. Haskell

```haskell
      module BT where
data BT a = Nil
  |Br (BT a) a (BT a)  deriving (Show,Eq)

elemente :: BT t -> [t]
elemente _ = []
```

Haskell

Lösung

Haskell

```haskell
        module BT where
data BT a = Nil
  |Br (BT a) a (BT a)  deriving (Show,Eq)

elemente :: BT t -> [t]
elemente Nil = []
elemente (Br l e r) = elemente l++e:elemente r
```

Haskell

Erläuterung

Der leere Baum erzielt eine leere Ergebnisliste. Ansonsten kommt das Wurzelelement in die Mitte zwischen den Ergebnis der Funktion auf dem linken und auf dem rechten Kindbaum.

Frage: Blattelemente eines Binärbaums

Schreiben Sie für eine binäre Baumstruktur eine Funktion, die die Liste aller Blattelemente von links nach rechts erzeugt. Haskell

```haskell
      module BT where
data BT a = Nil
  |Br (BT a) a (BT a)  deriving (Show,Eq)

blattElemente :: BT t -> [t]
blattElemente _ = []
```

Haskell

Lösung

Haskell

```haskell
        module BT where
data BT a = Nil
  |Br (BT a) a (BT a)  deriving (Show,Eq)

blattElemente :: BT t -> [t]
blattElemente Nil = []
blattElemente (Br Nil e Nil) = [e]
blattElemente (Br l _ r) = blattElemente l++blattElemente r
```

Haskell

Erläuterung

Leere Bäume haben keine Blattelemente. Blätter haben links und rechts leere Bäume als Kinder. Ansonsten sind die Blätter des linken Teilbaums und des rechten Teilbaums aneinander zu fügen.

Frage: Suche im binären Baum

Schreiben Sie für eine binäre Baumstruktur eine Funktion, die testet, ob ein Element mit einer bestimmten Eigenschaft enthalten ist. Haskell

```haskell
      module BT where
data BT a = Nil
  |Br (BT a) a (BT a)  deriving (Show,Eq)

enthaeltMit :: (t->Bool) -> BT t -> Bool
enthaeltMit _ _ = False
```

Haskell

Lösung

Haskell

```haskell
        module BT where
data BT a = Nil
  |Br (BT a) a (BT a)  deriving (Show,Eq)

enthaeltMit :: (t->Bool) -> BT t -> Bool
enthaeltMit _ Nil = False
enthaeltMit p (Br l e r)
   = p e || enthaeltMit p l || enthaeltMit p r
```

Haskell

Erläuterung

Leere Bäume enthalten gar kein Element. Ansonsten wird das Wurzelelement getestet und gegebenenfalls in den Kindern rekursiv weiter gesucht.

Frage: Pfad zu einem Element im Binärbaum

Schreiben Sie für eine binäre Baumstruktur eine Funktion, die als Liste den Pfad aller Elemente von der Wurzel zum ersten Auftreten eines bestimmten Elements im Baum berechnet. Ist das Element nicht im Baum, so sei das Ergebnis die leere Liste. Haskell

```haskell
      module BT where
data BT a = Nil
  |Br (BT a) a (BT a)  deriving (Show,Eq)

pfadZu :: Eq t => t -> BT t -> [t]
pfadZu _ _ = []
```

Haskell

Lösung

Haskell

```haskell
        module BT where
data BT a = Nil
  |Br (BT a) a (BT a)  deriving (Show,Eq)

pfadZu :: Eq t => t -> BT t -> [t]
pfadZu _ Nil = []
pfadZu x (Br l e r)
  |x==e = [e]
  |not$null pl = e:pl
  |not$null pr = e:pr
  |otherwise = []
  where
    pl = pfadZu x l
    pr = pfadZu x r
```

Haskell

Erläuterung

Für den leeren Baum ist das Ergebnis die leere Liste. Ist das gesuchte Element an der Wurzel, so besteht das Ergebnis aus der Liste mit dem einen Wurzelelement. Ansonsten wird erst beim linken Kind, dann beim rechten Kind geschaut, ob es da deb gesuchten Pfad gibt. wenn ja, hängt sich die Wurzel an diesen Pfad.