PROGRAMMIERUNG

ÜBUNG 3: ZEICHENKETTEN & BÄUME

Eric Kunze
eric.kunze@tu-dresden.de

TU Dresden, 27.04.2022

letzte Änderung:
27.04.2022, 12:52

Aufgabe 1

Zeichen & Zeichenketten

ZEICHEN & ZEICHENKETTEN

Zeichen

- Datentyp `Char`
- Eingabe in einfachen Anführungszeichen
- z.B. `'a'`, `'e'`, `'3'`

ZEICHEN & ZEICHENKETTEN

Zeichen

- Datentyp `Char`
- Eingabe in einfachen Anführungszeichen
- z.B. `'a'`, `'e'`, `'3'`

Zeichenketten

- Datentyp `String = [Char]`
- Eingabe in doppelten Anführungszeichen
- z.B. `"hallo"`, `"welt"`
- Konkatenation von Zeichenketten:

```
"hallo " ++ "welt" = "hallo welt"
```

AUFGABE 1 – TEIL (A)

Präfix – Test

```
isPrefix :: String -> String -> Bool
```

AUFGABE 1 – TEIL (A)

Präfix – Test

```
isPrefix :: String -> String -> Bool
```

```haskell
isPrefix :: String -> String -> Bool
isPrefix [] _ = True
isPrefix _ [] = False
isPrefix (p:ps) (c:cs) = p == c && isPrefix ps cs
```

AUFGABE 1 – TEIL (B)

Vorkommen eines Patterns zählen

```
countPattern :: String -> String -> Int
```

AUFGABE 1 – TEIL (B)

Vorkommen eines Patterns zählen

```
countPattern :: String -> String -> Int
```

```haskell
countPattern :: String -> String -> Int
countPattern "" "" = 1
countPattern _  "" = 0
countPattern xs yys@(y:ys)
  | isPrefix xs yys = 1 + countPattern xs ys
  | otherwise       = countPattern xs ys
```

Aufgabe 2

Algebraische Datentypen

ALGEBRAISCHE DATENTYPEN

- Ziel: problemspezifische Datenkonstruktoren
- z.B. in *C*: Aufzählungstypen
- funktionale Programmierung: algebraische Datentypen

Aufbau:

```
data Typename
  = Con1 t11 ... t1k1
  | Con2 t21 ... t2k2
  | ...
  | Conr tr1 ... trkr
```

- `Typename` ist ein Typkonstruktor
- `Con1, ... Conr` sind Datenkonstruktoren
- `tij` sind Typnamen

Konstruktoren beginnen mit Großbuchstaben.

ALGEBRAISCHE DATENTYPEN – BEISPIELE

```haskell
data Season = Spring | Summer | Autumn | Winter
```

ALGEBRAISCHE DATENTYPEN – BEISPIELE

```haskell
data Season = Spring | Summer | Autumn | Winter

goSkiing :: Season -> Bool
goSkiing Winter = True
goSkiing _      = False
```

ALGEBRAISCHE DATENTYPEN – BEISPIELE

```haskell
data Season = Spring | Summer | Autumn | Winter

goSkiing :: Season -> Bool
goSkiing Winter = True
goSkiing _      = False

data Weather = Sunny Int Int Bool | Cloudy Float
                                  | Rainy String Int
```

ALGEBRAISCHE DATENTYPEN – BEISPIELE

```haskell
data Season = Spring | Summer | Autumn | Winter

goSkiing :: Season -> Bool
goSkiing Winter = True
goSkiing _      = False

data Weather = Sunny Int Int Bool | Cloudy Float
                                  | Rainy String Int

data Bool =  True | False
```

ALGEBRAISCHE DATENTYPEN – BEISPIELE

```haskell
data Season = Spring | Summer | Autumn | Winter

goSkiing :: Season -> Bool
goSkiing Winter = True
goSkiing _      = False

data Weather = Sunny Int Int Bool | Cloudy Float
                                  | Rainy String Int

data Bool =  True | False

data BinTree = Branch Int BinTree BinTree | Nil
```

- Konstruktor `Branch`: erzeugt Knoten mit Beschriftung und zwei Kindern ⇝ *Rekursionsfall*
- Konstruktor `Nil`: erzeugt leeren Baum ⇝ *Basisfall*

ALGEBRAISCHE DATENTYPEN – REKURSION

- Basisfall: nulläre Wertkonstruktoren (z.B. `Nil`)
- Rekursionsfall: Konstruktoren mit Stelligkeit > 0 (z.B. `Branch`)

ALGEBRAISCHE DATENTYPEN – REKURSION

- Basisfall: nulläre Wertkonstruktoren (z.B. `Nil`)
- Rekursionsfall: Konstruktoren mit Stelligkeit > 0 (z.B. `Branch`)

Man **erzeugt** konkrete Bäume, indem man `Int` und `BinTree` aus der Typdefinition durch konkrete Werte ersetzt, z.B. `Int` durch `3` und `BinTree` durch `Nil` oder `Branch ...`.

ALGEBRAISCHE DATENTYPEN – REKURSION

- Basisfall: nulläre Wertkonstruktoren (z.B. `Nil`)
- Rekursionsfall: Konstruktoren mit Stelligkeit > 0 (z.B. `Branch`)

Man **erzeugt** konkrete Bäume, indem man `Int` und `BinTree` aus der Typdefinition durch konkrete Werte ersetzt, z.B. `Int` durch `3` und `BinTree` durch `Nil` oder `Branch ...`.

Pattern Matching funktioniert weiterhin; man nutzt dafür die Wertkonstruktoren (hier: `Branch` und `Nil`):

```
1 foo :: BinTree -> ...
2 foo Nil = ...
3 foo (Branch x l r) = ...
```

ALGEBRAISCHE DATENTYPEN – REKURSION

- Basisfall: nulläre Wertkonstruktoren (z.B. `Nil`)
- Rekursionsfall: Konstruktoren mit Stelligkeit > 0 (z.B. `Branch`)

Man **erzeugt** konkrete Bäume, indem man `Int` und `BinTree` aus der Typdefinition durch konkrete Werte ersetzt, z.B. `Int` durch `3` und `BinTree` durch `Nil` oder `Branch ...`.

Pattern Matching funktioniert weiterhin; man nutzt dafür die Wertkonstruktoren (hier: `Branch` und `Nil`):

```
1 foo :: BinTree -> ...
2 foo Nil = ...
3 foo (Branch x l r) = ...
```

Um in GHCi eine **Ausgabe** der Bäume zu erhalten, muss `deriving Show` hinter die Typdefinition geschrieben werden.

AUFGABE 2 – TEIL (A)

```haskell
data BinTree = Branch Int BinTree BinTree | Nil
```

AUFGABE 2 – TEIL (A)

```
data BinTree = Branch Int BinTree BinTree | Nil
```

Ein Beispielbaum:

```
mytree :: BinTree
mytree = Branch 0
          ( Nil )
          ( Branch 3
            ( Branch 1 Nil Nil )
            ( Branch 5 Nil Nil )
          )
```

... erfüllt die Suchbaumeigenschaft.

AUFGABE 2 – TEIL (B)

Test auf Baum-Gleichheit

AUFGABE 2 – TEIL (B)

Test auf Baum-Gleichheit

```
1 data BinTree = Branch Int BinTree BinTree | Nil
2 equal :: BinTree -> BinTree -> Bool
```

AUFGABE 2 – TEIL (B)

Test auf Baum-Gleichheit

```
1 data BinTree = Branch Int BinTree BinTree | Nil
2 equal :: BinTree -> BinTree -> Bool
```

```
1 equal :: BinTree -> BinTree -> Bool
2 equal Nil                Nil                = True
3 equal Nil                (Branch y l2 r2)   = False
4 equal (Branch x l1 r1)   Nil                = False
5 equal (Branch x l1 r1)   (Branch y l2 r2)
6   = (x == y) && (equal l1 l2) && (equal r1 r2)
```

AUFGABE 2 – TEIL (C)

Einfügen von Schlüsseln in einen Binärbaum

```
1 data BinTree = Branch Int BinTree BinTree | Nil
2 insert :: BinTree -> [Int] -> BinTree
```

AUFGABE 2 – TEIL (C)

Einfügen von Schlüsseln in einen Binärbaum

```haskell
data BinTree = Branch Int BinTree BinTree | Nil
insert :: BinTree -> [Int] -> BinTree
```

```haskell
insert :: BinTree -> [Int] -> BinTree
insert t     [] = t
insert t (n:ns) = insert t' ns
  where t' = insertSingle t n
        insertSingle Nil            n = Branch n Nil Nil
        insertSingle (Branch x l r) n
          | n < x     = Branch x (insertSingle l n) r
          | otherwise = Branch x l (insertSingle r n)
```

AUFGABE 2 – TEIL (D)

Levelorder-Traversierung

```haskell
data BinTree = Branch Int BinTree BinTree | Nil
unwind :: BinTree -> [Int]
```

AUFGABE 2 – TEIL (D)

Levelorder-Traversierung

```
data BinTree = Branch Int BinTree BinTree | Nil
unwind :: BinTree -> [Int]
```

Idee: Nutze Liste von Bäumen als Zwischenspeicher (Hilfsfunktion) und hänge Kindbäume hinten an diese Liste an

AUFGABE 2 – TEIL (D)

Levelorder-Traversierung

```haskell
data BinTree = Branch Int BinTree BinTree | Nil
unwind :: BinTree -> [Int]
```

Idee: Nutze Liste von Bäumen als Zwischenspeicher (Hilfsfunktion) und hänge Kindbäume hinten an diese Liste an

```haskell
unwind :: BinTree -> [Int]
unwind t = go [t]
 where
  go []                    = []
  go ( Nil             : ts) = go ts
  go ((Branch x l r) : ts) = x : go (ts ++ [l,r])
```

ENDE

Fragen?