Gedaechtnisprotokoll SWTPP 2. Test: Programmierparadigmen

27.02.2017

Part 2: Funktionale Programmierung in Haskell

Gegeben sind folgende Typen:

```
data Zutat  = Tomate | Fleisch | Salat
data Bun    = Weizen | Roggen
data Sosse  = Ketchup Sosse | Mayo Sosse | Keine
data Burger = Burger Bun [Zutat] Sosse
```

Sie duerfen annehmen, dass Zutat bereits eine Instanz von Eq ist.

Aufgabe 1

Verfollstaendigen sie folgende Statements so, dass am Ende in **dw** ein Doppel Wopper mit 2 mal Fleisch, ein mal Tomate, ohne Salat einmal Ketchup & einmal Mayo sowie ein Weizen Bun enthalten ist.

```
let zt =                                    in    -- Zutaten

  let so =                                    in  -- Sosse

    let dw =
```

.

Aufgabe 2

Vervollstaendigen Sie folgende Instanziierung der Typen Eq:

```
instance Eq Bun where

      (==)
```

.

Aufgabe 3

Implementieren Sie die Funktion **anzahlZutat**, welche die Anzahl einer bestimmten Zutat in einer gegebenen Zutatenliste zurueck gibt. Verwenden Sie Tail-Recursion!

```
anzahlZutat :: Int -> Zutat -> [Zutat] -> Int
```

.

Aufgabe 4

Implementieren Sie die Funktion **hatZutat**, welche *true* zurueck gibt falls gegebene Zustat in der Zutatenliste enthalten ist, sonst *false*. Verwenden sie **anzahlZutat**, Currying und Partial Application.

```
hatZutat :: Zutat -> [Zutat] -> Bool

hatZutat =
```

.

Aufgabe 5

Implementieren Sie die Funktion **istVegetarisch**, welche *true* zurueck gibt, falls in Gegebener Zutatenliste kein Fleisch enthalten ist, sonst *false* Verwenden Sie Currying und Partial Application.

```
istVegetarisch :: [Zutat] -> Bool

istVegetarisch =
```

.

Aufgabe 6

Implementieren Sie sie Funktion **zuSosse**, welche eine Zutatenliste nimmt und eine Sosse zurueck gibt, in der fuer jedes Fleisch in den Zutaten einmal Ketchup enthalten ist und fuer jede Tomate oder Salat einmal Mayo. Verwenden Sie das Listenfunktional *foldr*.

```
zuSosse :: [Zutat] ->

zuSosse = foldr
```

.

Aufgabe 7

Implementieren Sie die Funktion **paar**, welche eine unendliche Liste nimmt und immer zwei aufeinander folgende Elemente in ein neue Liste tut und diese Subslisten in einer unendlichen Liste zureuck gibt.

Anwendungsbeispiel:

```
λ> paar [1,2,3,4,5,6,...]
  [[1,2],[3,4],[5,6],...]
```

.

```
paar :: [a] -> [[a]]

paar          =
```

.

Aufgabe 8

Implementieren Sie die Funktion **stapel**, welche eine unendliche Liste von Zutaten zurueck gibt, wobei jede dritte Zutat eine Tomate sein soll, jede zweite Zutat, die noch keine Tomate ist Salat, und der Rest Fleisch. (Elegant ausgedrueckt /s)
Verwenden Sie Listenfunktionale!

```
stapel :: [Zutat]

stapel =
```

.

Aufgabe 9

Implementieren Sie nun die Funktion **burgerProduktion**, welche eine unendliche Liste von Burgern produziert. Verwenden sie **stapel** und **paar** um die Zutaten zu generieren, sowei **zuSosse** um die Sosse zu waehlen. Des weiteren sollen Burger mit Fleisch ein Weizen-Bun erhaten, die anderen ein Roggen-Bun.
Verwenden Sie ihre bereits geschriebenen Funktionen.

```
burgerProduktion :: [Burger]

burgerProduktion =
```

.

Part 3: Logische Programmierung in Prolog

Sei folgende Datenbasis gegeben:

```
startNode(a).
node(a).
node(b).
node(c).

edge(a,b).
edge(b,c).
edge(a,c).
```

Aufgabe 1

Implementieren Sie das Praedikat **reach**, welches wahr ist, falls vom ersten Knoten ueber die gerichteten Kanten der zweite Knoten erreicht werden kann. Dabei soll **reach** auch auf einer zyklischen Datenbasis korrekt funktionieren.
Beispiel:

```
?- reach(a, c, []).
   true.

?- reach(c, b, []).
   false.
```

.

```
reach(                    ) :-
```

.

Aufgabe 2

Implementieren Sie das Praedikat **sink**, welches wahr ist, falls der Knoten eine Senke ist, also keine Kanten von ihm ausgehen.
Beispiel:

```
?- sink(c).
   true.

?- sink(a).
   false.
```

.

```
sink( X ) :-
```

.

Aufgabe 3

Implementieren Sie das Praedikat **mirror**, welches wie folgt arbeitet:

```
?- mirror([1,2,3], [], X).
   X = [1,2,3,3,2,1]
```

Also in der hinteren Liste die erste List und danach die selbe Liste nochmal umgekehrt. Verwenden Sie dabei keine mitgelieferten Listen-Praedikate, wie append, usw.

```
mirror(                          ) :-
```

.