Aufgabenblatt 6

Aufgabe 1

a)

Attacken

=> Attacke;Attacken

=> Aktion;Attacken

=> Schelle;Attacken

=> Schelle;Attacke;Attacken

=> Schelle;Aktion;Attacken

=> Schelle;Schlag auf Schulter;Attacken

=> Schelle;Schlag auf Schulter;Attacke;Drehung Gradzahl;Attacken

=> Schelle;Schlag auf Schulter;Aktion;Drehung Gradzahl;Attacken

=> Schelle;Schlag auf Schulter;Schelle;Drehung Gradzahl;Attacken

=> Schelle;Schlag auf Schulter;Schelle;Drehung 90;Attacken

=> Schelle;Schlag auf Schulter;Schelle;Drehung 90;Attacke

=> Schelle;Schlag auf Schulter;Schelle;Drehung 90;Dampfhammer

b)

Attacken

=> Attacke;Drehung Gradzahl;Attacken

=> Attacke;Drehung Gradzahl;Aktion;Attacken

=> Attacke;Drehung Gradzahl;Aktion;Aktion;Attacken

=> Attacke;Drehung Gradzahl;Aktion;Aktion;Aktion

=> Attacke;Drehung Gradzahl;Aktion;Aktion;Dampfhammer

=> Attacke;Drehung Gradzahl;Aktion;Schlag auf Schulter;Dampfhammer

=> Attacke;Drehung Gradzahl;Schelle;Schlag auf Schulter;Dampfhammer

=> Attacke;Drehung 180;Schelle;Schlag auf Schulter;Dampfhammer

=> Dampfhammer;Drehung 180;Schelle;Schlag auf Schulter;Dampfhammer

```haskell
--
================================================================================
-- Grundlagen der Programmierung 2
-- Lars Gröber
-- Aufgabenblatt 6
--
================================================================================

  module Blatt6_LarsGroeber where
  import Prelude hiding ((<*>),(*>),(<*))
  import Data.Char
  import Data.List
  import Data.List.Split
  import CombParser

  --
================================================================================
  -- =
  -- = Aufgabe 1
  -- =

  --
  -- c)
  --
  tupleToList :: ([a], [a]) -> [a]
  tupleToList (x1,x2) = x1 ++ x2
  semi = token ";"
  gradzahl = token "45" <|> token "90" <|> token "135" <|> token "180"
  aktion = token "Schelle" <|> token "Schlag auf Schulter" <|> token "Schlag
auf Bauch"
  attacke = token "Dampfhammer" <|> aktion <|>  (aktion <*> (semi <*> attacke
<@ tupleToList) <@ tupleToList)
  attacken = attacke <|> (attacke <*> (semi <*> attacken <@ tupleToList) <@
tupleToList) <|>
     ((attacke <*> (semi <*> (token "Drehung " <*> gradzahl <@ tupleToList) <@
tupleToList) <@ tupleToList) <*> semi <@ tupleToList) <*> attacken <@
tupleToList

  parseAttacken :: Parser Char String
  -- Parser für die Grammatik L
  parseAttacken = attacken

  {-
  Testfälle:
  parseAttacken "Schelle" `shouldBe` [("", "Schelle")]
  parseAttacken "Schelle;Schelle;Dampfhammer" `shouldBe`
[(";Schelle;Dampfhammer","Schelle"),(";Dampfhammer","Schelle;Schelle"),
    ("","Schelle;Schelle;Dampfhammer"),(";Dampfhammer","Schelle;Schelle"),
    ("","Schelle;Schelle;Dampfhammer"),("","Schelle;Schelle;Dampfhammer"),
    ("","Schelle;Schelle;Dampfhammer")]
  -}

  --
  -- d)
  --

  combineTuple :: (CTuple, CTuple) -> CTuple
  combineTuple ((a1, b1, c1, d1), (a2, b2, c2, d2)) = (a1 + a2, b1 + b2, c1 +
c2, d1 + d2)

```

```haskell
  aktion2 = token "Schelle" <@ const (0,1,0,0) <|> token "Schlag auf Schulter" <@ const (0,0,1,0) <|> token "Schlag auf Bauch" <@ const (0, 0, 0, 1)
  attacke2 = token "Dampfhammer" <@ const (1,0,0,0) <|> aktion2 <|>  (aktion2 <*> (semi *> attacke2) <@ combineTuple)
  attacken2 = attacke2 <|> (attacke2 <*> (semi *> attacken2) <@ combineTuple) <|>
     ((attacke2 <*> (semi *> ((token "Drehung ") <*> gradzahl <@ const (0,0,0,0))) <@ combineTuple) <* semi) <*> attacken2 <@ combineTuple

  type CTuple = (Integer, Integer, Integer, Integer)

  parseAttackenC :: Parser Char (Integer,Integer,Integer,Integer)
  -- Parser, der die Anzahl an Dampfhammern, Schelle und Schläge zurückgibt.
  parseAttackenC = attacken2

  {-
  Testfälle:
  parseAttackenC "Schelle;Dampfhammer;Drehung 45;Dampfhammer" `shouldBe` [(";Dampfhammer;Drehung 45;Dampfhammer",(0,1,0,0)),
           (";Drehung 45;Dampfhammer",(1,1,0,0)),
           (";Drehung 45;Dampfhammer",(1,1,0,0)),
           ("",(2,1,0,0)),
           ("",(2,1,0,0))]
  parseAttackenC "Schelle" `shouldBe` [("",(0,1,0,0))]
  parseAttackenC "" `shouldBe` []
  parseAttackenC "Schelle;Dampfhammer" `shouldBe` [(";Dampfhammer",(0,1,0,0)),("",(1,1,0,0)),("",(1,1,0,0))]
  -}

  --
  -- e)
  --

  compareC :: Integer -> Integer -> String
  compareC b t | b < t = "Terence"
               | t < b = "Bud"
               | otherwise = ""

  compareResults :: CTuple -> CTuple -> (String,String,String,String)
  compareResults (b1, b2, b3, b4) (t1, t2, t3, t4) = (compareC b1 t1, compareC b2 t2, compareC b3 t3, compareC b4 t4)

  vergleiche :: String -> String ->  (String,String,String,String)
  -- Überprüft, welche Aktionen Bud oder Terence häufiger ausgeführt haben.
  vergleiche b t = compareResults (snd (last (parseAttackenC b))) (snd (last (parseAttackenC t)))

  {-
  Testfälle:
  vergleiche "Schelle;Schlag auf Bauch;Dampfhammer" "Schelle;Schelle;Schelle;Schlag auf Bauch" `shouldBe` ("Bud","Terence","","")
  vergleiche "Schelle" "Schelle;Schelle;Schelle;Schlag auf Bauch" `shouldBe` ("","Terence","","Terence")
  vergleiche "Dampfhammer;Dampfhammer" "Schelle;Schelle;Schelle;Schlag auf Bauch" `shouldBe` ("Bud","Terence","","Terence")
  -}
```