# Countdown!

## 8 of of 10 cats

Christian Höner zu Siederdissen
christian.hoener.zu.siederdissen@uni-jena.de

Theoretische Bioinformatik, Bioinformatik Uni Jena

Nov $10^{th}$, 2022

Countdown! (Video)

# Spielregeln

- es werden zufaellig 6 “Kombinations”-Zahlen gezogen (positiv, Mehrfachziehungen moeglich)
- es wird eine Zielzahl gezogen (positive, typisch bis 1 000)
- Kombiniere bis zu 6 der Zahlen in einem legalen arithmetischen Ausdruck
- Klammern sind erlaubt
- $+, -, \times, \div$ ist erlaubt
- (alle Zwischenergebnisse muessen positiv sein)
- Divisionen muessen aufgehen

# Beispiel

- $1, 3, 7, 10, 25, 50$
- Ziel: 831
- $7 + (1 + 10) \times (25 + 50) = 832$

Anzahl legaler Lsg nach Anzahl Zahlen:
1, 6, 96, 2652, 95157, 4724692

# Ziel

- wir implementieren das Spiel Countdown!
- erst ziemlich rudimentaer
- spaeter lernen wir dann Monaden daran kennen
- die Anzahl moeglicher Loesungen steigt exponentiell mit den Zahlen, “beste” Loesung ist geschickt zu finden
- Parsing

# Der Haskell-Kopf

```
1  {-# Language GeneralizedNewtypeDeriving #-}
2  {-# Language ScopedTypeVariables #-}
3
4  import Data.List (sort)
5  import Data.Char (isDigit)
6  import Text.Printf (printf)
```

# Strukturen arithmetischer Ausdruecke

```
 1  data Expr
 2    = Num Int
 3    | App Op Expr Expr
 4    deriving (Show,Eq)
 5
 6  data Op = Add | Sub | Mul | Div
 7    deriving (Show,Eq,Bounded,Enum)
 8
 9  newtype Value = Value {getValue :: Int}
10    deriving (Show,Eq,Ord,Enum,Real,Num,Integral)
```

```
 1  value :: Expr -> Value
 2  value (Num k) = Value k
 3  value (App o l r) = applyOp o (value l) (value r)
 4
 5
 6
 7  applyOp :: Op -> Value -> Value -> Value
 8  applyOp Add (Value l) (Value r) = Value (l + r)
 9  applyOp Sub (Value l) (Value r) = Value (l - r)
10  applyOp Mul (Value l) (Value r) = Value (l * r)
11  applyOp Div (Value l) (Value r) = Value (l `div` r)
```

```
1  legal :: Op -> Value -> Value -> Bool
2  legal Add l r = True
3  legal Sub l r = r < l
4  legal Mul l r = True
5  legal Div l r = l `mod` r == 0
```

# Erzeugen aller nichtleeren Teillisten einer Liste

```
 1  subseqs :: [a] -> [[a]]
 2
 3  subseqs [x] = [[x]]
 4
 5  -- erst alle suffixlisten
 6  -- dann nur x
 7  -- dann alle suffixlisten mit x vorne dran
 8  subseqs (x:xs) = xss ++ [x] : map (x:) xss
 9    where xss = subseqs xs
10
11  -- xs ++ ys ist teuer in xs
```

# Erstellen aller legalen Expr-Trees

Jeder Expr-Tree ist direkt mit seinem `Value` kombiniert

```
1  mkExprs :: [Int] -> [(Expr, Value)]
2
3  -- das ist die naechste Aufgabe
4  -- (bzw. Aufgabe fuer zu Hause)
```