Countdown!

8 out of 10 cats

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

Theoretische Bioinformatik, Bioinformatik Uni Jena

26. Oktober, 2023

Countdown! (Video)

Spielregeln

- Es werden zufaellig 6 “Kombinations”-Zahlen gezogen (positiv, Mehrfachziehungen moeglich, oft auch: 4 kleine, 2 grosse Zahlen)
- Es wird eine Zielzahl gezogen (positiv, typisch bis 1 000)
- Kombiniere bis zu 6 der Zahlen in einem legalen arithmetischen Ausdruck
- Klammern sind erlaubt
- $+, -, \times, \div$ sind erlaubt
- Divisionen muessen aufgehen
- *Optional:* Alle Zwischenergebnisse müssen positiv sein

Beispiel

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

Anzahl legaler Lsg nach Anzahl Zahlen:

Zahlen	Kombinationen	...
1		
2		
3		
4		
5		
6		

Beispiel

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

Anzahl legaler Lsg nach Anzahl Zahlen:

Zahlen	Kombinationen	...
1	1	a
2		
3		
4		
5		
6		

Beispiel

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

Anzahl legaler Lsg nach Anzahl Zahlen:

Zahlen	Kombinationen	...
1	1	a
2	6	$a + b,\ a - b,\ a \times b,\ a \div b,\ b - a,\ b \div a$
3		
4		
5		
6		

Beispiel

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

Anzahl legaler Lsg nach Anzahl Zahlen:

Zahlen	Kombinationen	...
1	1	a
2	6	$a + b,\ a - b,\ a \times b,\ a \div b,\ b - a,\ b \div a$
3	96	
4		
5		
6		

Beispiel

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

Anzahl legaler Lsg nach Anzahl Zahlen:

Zahlen	Kombinationen	...
1	1	a
2	6	$a + b,\ a - b,\ a \times b,\ a \div b,\ b - a,\ b \div a$
3	96	
4	2.652	
5		
6		

Beispiel

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

Anzahl legaler Lsg nach Anzahl Zahlen:

Zahlen	Kombinationen	...
1	1	a
2	6	$a + b,\ a - b,\ a \times b,\ a \div b,\ b - a,\ b \div a$
3	96	
4	2.652	
5	95.157	
6		

Beispiel

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

Anzahl legaler Lsg nach Anzahl Zahlen:

Zahlen	Kombinationen	...
1	1	a
2	6	$a + b,\ a - b,\ a \times b,\ a \div b,\ b - a,\ b \div a$
3	96	
4	2.652	
5	95.157	
6	4.724.692	

Ziel: Wir implementieren das Spiel Countdown!

Zu lösende Probleme:

1. Interne Repräsentation von Kandidaten: Datentypen
2. Kandidat evaluieren: Funktionen, Pattern Matching, Rekursion
3. Kandidaten aus Liste von Zahlen generieren: Rekursion
4. 6 + 1 Zahl zufällig ziehen: IO, Monaden
5. Parsing der Nutzereingabe: Parser, Monaden
6. Bestmögliche Lösung, Anzahl Lösungen
7. Ideen dem exponentiellen Anstieg zu begegnen?

Die effiziente Behandlung von exponentiell großen Suchräumen findet man in der Bioinformatik häufig als Problem: zB. RNA-Faltung.

Haskell Module

Module gruppieren Haskell-Code nach Anwenderwunsch.
`module Main` ist speziell und bezeichnet das Modul welches die `main`-Funktion enthält.

```
1  -- | The countdown game
2
3  module Countdown where
```

Typ- und Datenkonstruktor für Operatoren

- `data Op` Typkonstruktor mit Keyword `data`
- `Add | Sub | ...`: Datenkonstruktoren
- `deriving`: ignorieren wir für den Moment!

```
1  -- | The countdown game
2
3  module Countdown where
4
5  -- | Datentyp for die legalen Operatoren
6
7  data Op = Add | Sub | Mul | Div
8    deriving (Show,Eq,Read,Bounded,Enum)
```

Rekursive Datentypen

Wie können wir einen Aufrufbaum für $7 + (1 + 10) \times (25 + 50) = 832$ erzeugen?

- `data Expr`: Typkonstruktor mit Keyword `data`

Rekursive Datentypen

Wie können wir einen Aufrufbaum für $7 + (1 + 10) \times (25 + 50) = 832$ erzeugen?

- `data Expr`: Typkonstruktor mit Keyword `data`
- `Zahl Int`: Datenkonstruktor `Zahl :: Int -> Expr`

Rekursive Datentypen

Wie können wir einen Aufrufbaum für $7 + (1 + 10) \times (25 + 50) = 832$ erzeugen?

- `data Expr`: Typkonstruktor mit Keyword `data`
- `Zahl Int`: Datenkonstruktor `Zahl :: Int -> Expr`
- `App Op Expr Expr`: Datenkonstruktor
  `App :: Op -> Expr -> Expr -> Expr`

Rekursive Datentypen

Wie können wir einen Aufrufbaum für $7 + (1 + 10) \times (25 + 50) = 832$ erzeugen?

- `data Expr`: Typkonstruktor mit Keyword `data`
- `Zahl Int`: Datenkonstruktor `Zahl :: Int -> Expr`
- `App Op Expr Expr`: Datenkonstruktor
  `App :: Op -> Expr -> Expr -> Expr`
- `deriving` ignorieren

```
1  -- | Die Form von Countdown Ausdruecken
2
3  data Expr
4    = Zahl Int
5    | App Op Expr Expr
6    deriving (Show,Read,Eq)
```

Beispiel: Aufrufbaum

$$7 + (1 + 10) \times (25 + 50) = 832$$

```
1  test02 :: Expr
2  test02 =
3    App
4      Add
5      (Zahl 7)
6      (App Mul
7        (App Add (Zahl 1) (Zahl 10))
8        (App Add (Zahl 25) (Zahl 50))
9      )
```

Typ-Aliase

- Typ-Aliase geben neue Namen für den gleichen Typ.
- `type A = Int` und `type B = Int` geben `A` $\equiv$ `B`
- Wir können Aliase nur nutzen um den Code besser lesbar zu machen, *nicht* um dem Compiler mitzuteilen das bestimmte Typen unterschiedlich sein sollen.
- Lösung: entweder `data Wert = Wert Int` oder `newtype Wert = Wert Int`

```
1  -- | Alias fuer den Wert eines Countdowns
2
3  type Wert = Int
```

Anwendung von Operationen auf Werte

- Funktionsdefn: `appOp :: Op -> Wert -> Wert -> Wert`

```
1  -- | Gegeben Op und zwei Werte, rechne deren Ergebnis
2
3  appOp :: Op -> Wert -> Wert -> Wert
```

Anwendung von Operationen auf Werte

- Funktionsdefn: `appOp :: Op -> Wert -> Wert -> Wert`
- Pattern Matching: `appOp Add ...` "matched" auf Datenkonstruktoren

```
1  -- | Gegeben Op und zwei Werte, rechne deren Ergebnis
2
3  appOp :: Op -> Wert -> Wert -> Wert

4  appOp Add l r = l + r
5  appOp Sub l r = l - r
6  appOp Mul l r = l * r
7  appOp Div l r = l `div` r
```

Anwendung von Operationen auf Werte

- Funktionsdefn: `appOp :: Op -> Wert -> Wert -> Wert`
- Pattern Matching: `appOp Add ...` "matched" auf Datenkonstruktoren
- Wir garantieren hier nicht das `Sub` aund `Div` "legale" Ergebnisse liefern

```
1  -- | Gegeben Op und zwei Werte, rechne deren Ergebnis
2
3  appOp :: Op -> Wert -> Wert -> Wert

4  appOp Add l r = l + r
5  appOp Sub l r = l - r
6  appOp Mul l r = l * r
7  appOp Div l r = l `div` r
```

Einen kompletten Ausdruck auswerten

- `Expr`'s sind rekursiv aufgebaut und sollen auf `Werte` reduziert werden
- wie `Zahl` behandeln?
- wie `App` behandeln?

Einen kompletten Ausdruck auswerten

- `Expr`'s sind rekursiv aufgebaut und sollen auf `Werte` reduziert werden
- wie `Zahl` behandeln?
- wie `App` behandeln?

```
1  -- | Ausdruck auswerten
2
3  auswerten :: Expr -> Wert
```

Einen kompletten Ausdruck auswerten

- `Expr`'s sind rekursiv aufgebaut und sollen auf `Werte` reduziert werden
- wie `Zahl` behandeln?
- wie `App` behandeln?

```
1  -- | Ausdruck auswerten
2
3  auswerten :: Expr -> Wert

4  auswerten (Zahl k) = k
```

Einen kompletten Ausdruck auswerten

- `Expr`'s sind rekursiv aufgebaut und sollen auf `Werte` reduziert werden
- wie `Zahl` behandeln?
- wie `App` behandeln?

```
1  -- | Ausdruck auswerten
2
3  auswerten :: Expr -> Wert

4  auswerten (Zahl k) = k

5  auswerten (App o l r) = appOp o (auswerten l) (auswerten
       r)
```

Beispiel

```
1  test01 :: Wert
2  test01 = auswerten (App Add (Zahl 1) (Zahl 2))
```

Beispiel: `test01 [Enter]` $\Rightarrow$ `3`

Wann ist ein Ausdruck legal?

- Es werden zufaellig 6 “Kombinations”-Zahlen gezogen (positiv, Mehrfachziehungen moeglich, oft auch: 4 kleine, 2 grosse Zahlen)
- Es wird eine Zielzahl gezogen (positiv, typisch bis 1 000)
- Kombiniere bis zu 6 der Zahlen in einem legalen arithmetischen Ausdruck
- Klammern sind erlaubt
- $+, -, \times, \div$ sind erlaubt
- Divisionen muessen aufgehen
- *Optional:* Alle Zwischenergebnisse müssen positiv sein

Legalitätstest auf Operationen

```
1  -- | Ist eine Operation legal?
```

Legalitätstest auf Operationen

```
1  -- | Ist eine Operation legal?

2  legal :: Op -> Wert -> Wert -> Bool
```

Legalitätstest auf Operationen

```
1  -- | Ist eine Operation legal?
2  legal :: Op -> Wert -> Wert -> Bool
3  legal Add l r = True
```

Legalitätstest auf Operationen

```
1  -- | Ist eine Operation legal?
2  legal :: Op -> Wert -> Wert -> Bool
3  legal Add l r = True
4  legal Sub l r = r < l
```

Legalitätstest auf Operationen

```
1  -- | Ist eine Operation legal?
2  legal :: Op -> Wert -> Wert -> Bool
3  legal Add l r = True
4  legal Sub l r = r < l
5  legal Mul l r = True
```

Legalitätstest auf Operationen

```
1  -- | Ist eine Operation legal?
2  legal :: Op -> Wert -> Wert -> Bool
3  legal Add l r = True
4  legal Sub l r = r < l
5  legal Mul l r = True
6  legal Div l r = l `mod` r == 0
```

Erzeugen aller nichtleeren Teillisten einer Liste

Kombiniere bis zu 6 der Zahlen in einem legalen arithmetischen Ausdruck
. . . das heisst in beliebiger Reihenfolge, oder?

```
1  -- | All moeglichen Teilsequenzen einer Liste
2
3  subseqs :: [a] -> [[a]]
```

Erzeugen aller nichtleeren Teillisten einer Liste

Kombiniere bis zu 6 der Zahlen in einem legalen arithmetischen Ausdruck
...das heisst in beliebiger Reihenfolge, oder?

```
1  -- | All moeglichen Teilsequenzen einer Liste
2
3  subseqs :: [a] -> [[a]]

4  subseqs [x] = [[x]]
```

Erzeugen aller nichtleeren Teillisten einer Liste

Kombiniere bis zu 6 der Zahlen in einem legalen arithmetischen Ausdruck
...das heisst in beliebiger Reihenfolge, oder?

```
1  -- | All moeglichen Teilsequenzen einer Liste
2
3  subseqs :: [a] -> [[a]]

4  subseqs [x] = [[x]]

5  subseqs (x:xs) = xss ++ [x] : map (x:) xss
6    where xss = subseqs xs
```

Nicht-leeres Splitten einer Liste

```
1  -- | Given ordered @xs@, create two ordered sublists,
       which when merged are equal to @xs@.
2  --
3  -- let (ys,zs) = unmerges xs
4  -- forall y in ys . z in zs : merge y z == xs
5
6  unmerges :: Show a => [a] -> [([a],[a])]
```

Nicht-leeres Splitten einer Liste

```
1  -- | Given ordered @xs@, create two ordered sublists,
       which when merged are equal to @xs@.
2  --
3  -- let (ys,zs) = unmerges xs
4  -- forall y in ys . z in zs : merge y z == xs
5
6  unmerges :: Show a => [a] -> [([a],[a])]
```

```
1  unmerges [x,y] = [([x],[y]), ([y],[x])]
```

Nicht-leeres Splitten einer Liste

```
1  -- | Given ordered @xs@, create two ordered sublists,
       which when merged are equal to @xs@.
2  --
3  -- let (ys,zs) = unmerges xs
4  -- forall y in ys . z in zs : merge y z == xs
5
6  unmerges :: Show a => [a] -> [([a],[a])]
```

```
1  unmerges [x,y] = [([x],[y]), ([y],[x])]
```

```
1  unmerges (x:xs) = [([x],xs),(xs,[x])] ++ concatMap (add x
       ) (unmerges xs)
2    where add x (ys,zs) = [(x:ys,zs),(ys,x:zs)]
3  unmerges burn = error $ show burn
```

Einfügen der `Op`

```
1  -- | Combine two expression trees with all possible,
       legal operations
2
3  combine :: (Expr,Int) -> (Expr,Int) -> [(Expr,Int)]
4  combine (l,v) (r,w) = [(App op l r, appOp op v w) | op <-
       ops, legal op v w]
5    where ops = [Add,Sub,Mul,Div] -- [minBound..maxBound]
```

Erstellen aller legalen Expr-Trees

```
1  -- | Generate all possible expressions for our set of
       numbers
2
3  mkExprs :: [Int] -> [(Expr, Int)]
4  mkExprs [x] = [(Zahl x, x)]
5  mkExprs xs = [ ev | (ys,zs) <- unmerges xs
6                    , l <- mkExprs ys, r <- mkExprs zs
7                    , ev <- combine l r ]
```