Countdown!

8 of of 10 cats

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

Theoretische Bioinformatik, Bioinformatik Uni Jena

Nov 17^{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  -- Blaetter sind Zahlen
4  mkExprs [x] = [(Num x, Value x)]
5
6  -- finde alle Paare sortierte Mergelisten
7  mkExprs xs = [ ev | (ys,zs) <- unmerges xs
8  -- erstelle alle Expressions pro Mergeliste
9                    , l <- mkExprs ys, r <- mkExprs zs
10 -- kombiniere l,r mit jeweils allen legalen Operationen
11                   , ev <- combine l r ]
```

Mergelisten

```
 1  unmerges :: Show a => [a] -> [([a],[a])]
 2  -- beide Varianten an Paaren
 3  unmerges [x,y] = [([x],[y]), ([y],[x])]
 4
 5  -- singleton Listen
 6  -- alle Teillisten ohne x mit x dann an beide Kandidaten
 7  unmerges (x:xs) = [([x],xs),(xs,[x])]
 8            ++ concatMap (add x) (unmerges xs)
 9    where add x (ys,zs) = [(x:ys,zs),(ys,x:zs)]
10
11  merge :: Ord a => [a] -> [a] -> [a]
12  merge [] rs = rs
13  merge ls [] = ls
14  merge (l:ls) (r:rs) | l <= r = l : merge ls (r:rs)
15                      | r < l  = r : merge (l:ls) rs
16
17  -- unmerges [1..3]
18  -- [([1],[2,3]),([2,3],[1]),([1,2],[3])
19  -- ,([2],[1,3]),([1,3],[2]),([3],[1,2])]
```

Kombiniere zwei Teilbaeume

```
1  combine :: (Expr,Value) -> (Expr,Value) -> [(Expr,Value)]
2  combine (l,v) (r,w) =
3    [ (App op l r, applyOp op v w)
4    | op <- ops, legal op v w
5    ]
6    where ops = [Add,Sub,Mul,Div] -- [minBound..maxBound]
```

Loesungen testen

Welche `Expr` ist am naechsten an "`nearest 831`" dran?

```
1  nearest :: (Eq b, Ord b, Num b) => b -> [(a,b)] -> (a,b)
2  nearest n ((e,v):evs)
3  -- direkt eine Loesung gefunden?
4    | d == 0 = (e,v)
5  -- nein? Suche starten, mit Abstand d
6    | otherwise = search n d (e,v) evs
7    where d = abs (n-v)
```

Loesungen testen

Suche solange nach weiteren Loesungen bis die Entfernung 0 ist oder keine weiteren Lsg existieren

```
search :: (Ord a, Num a) => a->a->(b,a) -> [(b,a)] -> (b,a)
-- es gibt nur suboptimale Lsg
search n d ev [] = ev
-- teste naechste Lsg
search n d ev ((e,v):evs)
-- optimal
  | d' == 0 = (e,v)
-- besser
  | d' < d  = search n d' (e,v) evs
-- schlechter
  | d' >= d = search n d ev evs
  where d' = abs (n-v)
```

Tokenizing

Tokens zerlegen eine Eingabe in "atomare" Bausteine auf denen wir leichter arbeiten koennen

```
1  data Token
2    = TNum Int
3    | TOp Op
4    | TLeft
5    | TRight
6    deriving (Show,Eq)
```

Tokenizer

```
1  tokenize :: String -> [Token]
2  tokenize [] = []
3  tokenize (x:xs)
4  -- eine Zahl
5    | isDigit x = let (ls,rs) = span isDigit xs
6                  in  TNum (read (x:ls)) : tokenize rs
7  -- Operator
8    | x `elem` "+-*/" = TOp (parseOp x) : tokenize xs
9  -- Klammer links
10   | x == '(' = TLeft : tokenize xs
11 -- Klammer rechts
12   | x == ')' = TRight : tokenize xs
13
14
15 parseOp :: Char -> Op
16 parseOp '+' = Add
17 ...
```

Von Token zu Expr

Parser wie pSumPNP erwarten eine Liste von Token und geben eine Expr zurueck, sowie eine Restliste von *nicht bearbeiteten* Token

```
1  token2Expr :: [Token] -> Expr
2  token2Expr xs = case pSumPNP xs
3    of Just (expr,[]) -> expr
4       Nothing -> error (show xs)
```

Wir bauen den Parser jetzt aber mal bottom-up auf

hoechste "Precedence": Zahlen und Klammern

```
 1  pNumParen :: [Token] -> Maybe (Expr,[Token])
 2
 3  -- eine Zahl zu parsen ist einfach
 4  pNumParen (TNum n:xs) = Just (Num n, xs)
 5
 6  -- bei linker Klammer: auf Rest den kompletten Parser
 7  -- rekursiv laufen lassen
 8  pNumParen (TLeft:xs) = case pSumPNP xs of
 9  -- alles bis zur schliessenden Klammer + Rest
10    Just (expr, TRight:ys) -> Just (expr, ys)
11  -- misses clothing bracket
12    Just _ -> Nothing
13    Nothing -> Nothing
14  pNumParen _ = Nothing
```

Produkt + Division

```
 1  pProdNP :: [Token] -> Maybe (Expr,[Token])
 2
 3  -- von links Zahl oder Klammer lesen
 4  pProdNP xs = case pNumParen xs of
 5
 6  -- es geht mit Mul weiter
 7    Just (el, TOp Mul:ys) -> case pProdNP ys of
 8
 9  -- zweites Argument ebenso erfolgreich geparsed
10      Just (er, zs) -> Just (App Mul el er, zs)
11      Nothing -> Nothing
12
13  -- analog fuer Division
14    Just (el, TOp Div:ys) -> case pProdNP ys of
15      Just (er, zs) -> Just (App Div el er, zs)
16      Nothing -> Nothing
17    res -> res
```

```
1  pSumPNP :: [Token] -> Maybe (Expr,[Token])
2
3  -- erstmal testen ob Mul,(), oder Zahl
4  pSumPNP xs = case pProdNP xs of
5
6  -- Danach folgt Add Token
7    Just (el, TOp Add:ys) -> case pSumPNP ys of
8
9  -- zweites Argument ebenso erfolgreich geparsed
10     Just (er, zs) -> Just (App Add el er, zs)
11     Nothing -> Nothing
12
13 -- analog fuer Subtraktion
14   Just (el, TOp Sub:ys) -> case pSumPNP ys of
15     Just (er, zs) -> Just (App Sub el er, zs)
16     Nothing -> Nothing
17   res -> res
```

Run!

```
 1  main = do
 2    print "Give␣me␣numbers"
 3    ns :: [Int] <- sort . fmap read . words <$> getLine
 4    print "Give␣me␣a␣target"
 5    tgt :: Int <- fmap read getLine
 6    printf "%s␣closest␣to␣%d␣with:\n"
 7      (unwords $ fmap show ns) tgt
 8    suggest :: String <- getLine
 9    let (expr,nst) = nearest (Value tgt)
10        . concatMap mkExprs $ subseqs ns
11        myexp = token2Expr $ tokenize suggest
12        myval = value myexp
13    printf "You␣have:␣%d\n" (getValue myval)
14    printf "Optimal␣is:␣%d\n" (getValue nst)
```

Und nun?

- Verbessern von `mkExprs` (Memoization)
- Bessere Parser (Monaden!)
- Fehlerbehebung (Monaden!)
- Ein- und Ausgabe (Monaden!)
- “gegeneinander spielen” (. . . Monaden!)