

2.md

class: center, middle

Functional Programming

2. Countdown

Frank C Langbein

frank@langbein.org

Version 1.0

Countdown

- British game show since 1982 (>>5,000 episodes)
 - Based on French version: "Des Chiffre et Des Lettre" (since 1965, >>20,000 episodes)
- The **countdown problem**
 - Given a **set of numbers** and a **set of arithmetic operators**,
 - **construct an expression** to calculate a **given value**
- **Rules** (abstracted from the TV series for simplicity)
 - All numbers are **natural numbers**
 - Each can be used **at most once**
- For example:
 - Numbers: 1 3 7 10 25 50
 - Operators: + - * /
 - Value: 765 - there are 780 solutions (e.g. $(25-10) * (50+1)$)
 - Value: 831 - there is no solution

Expressions

- Firstly, need to **represent expressions** (a tree)
- Introduce **new type** for operators

```
data Op = Add | Sub | Mul | Div deriving (Show)
      -- to display it
```

- **Apply** operator

```
apply :: Op -> Int -> Int -> Int
apply Add x y = x + y
apply Sub x y = x - y
apply Mul x y = x * y
apply Div x y = x `div` y
```

- Decide if an operator can be applied, is **valid**

```
valid :: Op -> Int -> Int -> Bool
valid Add _ _ = True -- Placeholder _
```

```
valid Sub x y = x > y
valid Mul _ _ = True
valid Div x y = x `mod` y == 0
```

Evaluate Expressions

- Expressions type (with constructors `Val` and `App`)

```
data Expr = Val Int | App Op Expr Expr deriving (Show)
```

- Function to **evaluate** an expression

```
eval :: Expr -> [Int]
eval (Val n) = [n | n > 0]
eval (App o l r) = [apply o x y | x <- eval l,
                                y <- eval r,
                                valid o x y]
```

- For Example

```
> eval $ App Mul (Val 10) (Val 2)
```

- Anything after '\$' takes precedence over anything before

```
> putStrLn (show $ 1 + 1)
> putStrLn $ show (1 + 1)
> putStrLn $ show $ 1 + 1
```

Extract Values from Expressions

- Return a list of all values in an expression

```
values :: Expr -> [Int]
values (Val n) = [n]
values (App _ l r) = values l ++ values r
```

- For example

```
values $ App Mul (Val 10) (Val 2)
```

Brute Force Solver

- Decide if an expression is a solution for a given list of source numbers and a target number (**formal problem definition**)

```
solution :: Expr -> [Int] -> Int -> Bool
solution e ns n = elem (values e) (choices ns)
                && eval e == [n]
```

- Create all expressions and test if they solve the problem:**

- Construct all expressions
- Test an expression to filter

- Return a list of all expressions that solve a countdown problem

```
solutions :: [Int] -> Int -> [Expr]
solutions ns n = [e | ns' <- choices ns, -- all number sequences
                     e <- exprs ns',    -- all expressions
                     eval e == [n]]     -- filter
```

- All solutions: `> solutions [1,3,7,10,25,50] 765`
- First solution: `> solutions [1,3,7,10,25,50] 765 !! 1`

Combinatorics

- Generate all sub-sequences of a list, including all orderings and all possibilities of including and excluding each element of the list

```
subs :: [a] -> [[a]]
subs [] = [ [] ]
subs (x:xs) = yss ++ map (x:) yss
              where yss = subs xs
```

- Create all possible ways of inserting a new element into a list

```
interleave :: a -> [a] -> [[a]]
interleave x [] = [[x]]
interleave x (y:ys) = (x:y:ys) : map (y:) (interleave x ys)
```

- Create all permutations of a list

```
perms :: [a] -> [[a]]
perms [] = [ [] ]
perms (x:xs) = concat (map (interleave x) (perms xs))
```

Choices

- All possible ways of selecting zero or more elements in any order from a list

◦ E.g. `choices[1,2] = [[],[1],[2],[1,2],[2,1]]`

```
choices :: [a] -> [[a]]
choices = concat . map perms . subs
```

- Note, $(f \cdot g) x = f g x$ (composing functions)
- But $f \$ x = f x$ (change precedence)

- So `choices [...]` gives all possible combinations of numbers, but without operators

All Expressions

- Return a list of all possible expressions whose values are precisely a given list of numbers

```
exprs :: [Int] -> [Expr]
exprs [] = []
exprs [n] = [Val n]
exprs ns = [e | (ls,rs) <- split ns, -- split into left / right numbers
                l <- exprs ls,      -- left expression
                r <- exprs rs,      -- right expression
                e <- combine l r]    -- combine left and right
```

- This is the key brute force solver function

Split and Combine

- Split list into all possible left/right pairs
 - E.g. `split [1,2,3,4] = [[1],[2,3,4]], ([1,2],[3,4]), ([1,2,3],[4])`

```
split :: [a] -> [[a],[a]]
split [] = []
split [_] = []
split (x:xs) = ([x],xs) : [(x:ls,rs) | (ls,rs) <- split xs]
```

- Combine two expressions with all possible operators

```
combine :: Expr -> Expr -> [Expr]
combine l r = [App o l r | o <- [Add,Sub,Mul,Div]]
```

First Solver

- This is everything we need for the function defined earlier

```
solutions :: [Int] -> Int -> [Expr]
solutions ns n = [e | ns' <- choices ns, -- all number sequences
                      e <- exprs ns',   -- all expressions
                      eval e == [n]]    -- filter
```

- All solutions: `> solutions [1,3,7,10,25,50] 765`
- First solution: `> solutions [1,3,7,10,25,50] 765 !! 1`
- How efficient is this?
 - Note, this test ~33M expressions
 - Test in the labs...

Improvements?

- Many of the expressions that are considered will typically be **invalid**
 - They fail to evaluate
 - For the example, only about 5M of ther 33M expressions are valid
- **Combining generation with evaluation** would reject invalid expressions earlier
- We seek to define a function that **fuses together** the generation and evaluation of expressions

Fusing two Functions

- Type to represent **fused results**
 - Valid expressions and their values
- ```
type Result = (Expr,Int)
```
- Generate result pairs (valid expression,value) for a list of numbers

```
results :: [Int] -> [Result]
results [] = []
results [n] = [(Val n,n) | n > 0]
results ns = [res | (ls,rs) <- split ns,
 lx <- results ls,
 ry <- results rs,
 res <- combine' lx ry]
```

where we **combine expressions and their values**

```
combine' :: Result -> Result -> [Result]
combine' (l,x) (r,y) = [(App o l r, apply o x y) | o <- [Add,Sub,Mul,Div],
 valid o x y]
```

## Second Solver

```
solutions' :: [Int] -> Int -> [Expr]
solutions' ns n = [e | ns' <- choices ns, -- all number sequences
 (e,m) <- results ns', -- only all valid expressions
 m == n]
```

- This should be faster!
  - Test in the labs...

## Can we do better?

- Many expressions will be **essentially the same** using simple arithmetic properties, e.g.
  - $x * y = y * x$
  - $x * 1 = x$
- Exploiting such properties would considerably reduce the search and solution spaces.
- This can be done in `valid` :

```
valid' :: Op -> Int -> Int -> Bool
valid' Add x y = x <= y
valid' Sub x y = x > y
valid' Mul x y = x /= 1 && y /= 1 && x <= y
valid' Div x y = y /= 1 && x `mod` y == 0
```

- \* Create a third solver with this in the labs