Do you know how to print a diamond?

I'm not sure. What do you mean by *print a diamond*?

I mean printing letters in a certain fashion so that the sequence of letters looks like a diamond.
Like this:

```
  A
 B B
C   C
 B B
  A
```

This was a diamond made with letters from A to C. Can you print a diamond with letters from A to F?

I suppose.

```
     A
    B B
   C   C
  D     D
 E       E
F         F
 E       E
  D     D
   C   C
    B B
     A
```

That is right. Let's write a program that prints diamonds like that.

Okay. How do we tell the computer you want a diamond printed?

We write a small *haskell* program that given an uppercase letter as an argument, calls a function that computes a diamond, and then prints the result of that function.

Here is a program which does that. Of course, the diamond function does not compute anything, it returns fixed values instead.

```
import System.Environment (getArgs)

diamond 'A' = ["A"]
diamond 'B' = [" A ","B B"," A "]

main = getArgs >>=
   putStr ∘ unlines ∘ diamond ∘ head ∘ head
```

Main.hs

The program prints a diamond for letters A or B only:

```
runhaskell Main.hs A ↩
A

runhaskell Main.hs B ↩
 A
B B
 A

runhaskell Main.hs C ↩
Main.hs: Main.hs:(3,1)-(4,33):
Non-exhaustive patterns in function diamond
```

So let's replace the diamond function in that program with a better one, sitting in its own module.

```
import System.Environment (getArgs)
import Diamond (diamond)

main = getArgs >>=
   putStr ∘ unlines ∘ diamond ∘ head ∘ head
```

We can create the module, but for now diamond is undefined.

```
module Diamond
where

diamond :: Char → [String]
diamond _ = undefined
```

Diamond.hs

Well then let's define it. What is the first thing we can say about diamond?

Well, for one thing, it cannot be empty.

Ok let's describe that in a new program.

```
import Test.Hspec (hspec, describe, it)
import Test.QuickCheck (forAll, choose)
import Diamond (diamond)

main = hspec $ do
    describe "a diamond" $ do
        let letter = choose ('A','Z')

        it "is not empty" $ forAll letter $
            not ∘ null ∘ concat ∘ diamond
```

Specs.hs

Can you write an implementation for which that property holds?

Very easily.

```
module Diamond
where

diamond :: Char → [String]
diamond _ = [" "]
```

What other property should we describe?

The upper left corner of a diamond from A to the letter *l* is made with a diagonal from A to *l*.

How do you delimit the upper left corner of the diamond?

If *n* is the length of the suite from A to *l*, then taking the *n* first characters of the *n* first lines should do the trick.

```
ul n = map (take n)∘(take n)
```

How would you check that a list of Strings contains a diagonal from A to *l*?

If *n* is the length of the list then the character at position $n - 1$ in line 0 should be A; the character at position $n - 2$ in line 1 should be B, and so on.
All the other positions should be filled with space.

	0	1	2	3	4
0					A
1				B	
2			C		
3		D			
4	E				

Let's represent that property:

```
it "contains a diagonal made of letters" $
    forAll letter $ λ l →
        let ls = ['A'..l]
            n  = length ls
            d  = diamond l
            ul = (map (take n)∘(take n)) d
            m  = n − 1
            diag j i | i == (m−j) = ul !! j !! i == ls !! j
                     | otherwise  = ul !! j !! i == ' '
        in and [diag r c | r <- [0..m], c <- [0..m]]
```

Ok, now there is some work to do:

```
module Diamond
where

diamond :: Char → [String]
diamond l = map pattern [0..n]
    where
    pattern i = (sp (n−i)) ++ [ls !! i] ++ (sp i)
    sp n = replicate n ' '
    ls = ['A'..l]
    n = length ls − 1
```

Another interesting property of the diamond is that flipping it horizontally yields the same value.

```
it "is symmetric horizontally" $ do
    forAll letter $ λ l →
        diamond l == reverse (diamond l)
```

Indeed. Maybe we could just mirror our pattern:

```
module Diamond
where

diamond :: Char → [String]
diamond l = mirror (map pattern [0..n])
    where
    pattern i = (sp (n−i)) ++ [ls !! i] ++ (sp i)
    sp n = replicate n ' '
    ls = ['A'.. l]
    n = length ls − 1

    mirror xs = xs ++ reverse xs
```

Well, guess what: flipping the diamond vertically also yields the same result.

```
it "is symmetric vertically" $ do
forAll letter $ λ l →
    diamond l == map reverse (diamond l)
```

Ok, then we will mirror each line as well.

```
module Diamond
where

diamond :: Char → [String]
diamond l = mirror (map (mirror∘pattern) [0.. n])
    where
    pattern i = (sp (n−i)) ++ [ls !! i] ++ (sp i)
    sp n = replicate n ' '
    ls = ['A'.. l]
    n = length ls − 1

    mirror xs = xs ++ reverse xs
```

Are we done with our diamond function?

Not yet: it's still lacking a property, as can be seen from this visual check:

```
runhaskell Main.hs D ↩
   AA
  B  B
 C    C
D      D
D      D
 C    C
  B  B
   AA
```

I see. The width and height of a diamond should be an odd number.

```
it "has an odd height and width" $ do
forAll letter $ λ l →
    odd (length (diamond l))
    && odd (maximum (map length (diamond l)))
```

The solution is to remove an element in the mirroring process:

```
module Diamond
where

diamond :: Char → [String]
diamond l = mirror (map (mirror∘pattern) [0.. n])
    where
    pattern i = (sp (n−i)) ++ [ls !! i] ++ (sp i)
    sp n = replicate n ' '
    ls = ['A'.. l]
    n = length ls − 1

    mirror xs = xs ++ tail (reverse xs)
```

Perfect!

```
runhaskell Main.hs D ↩
   A
  B B
 C   C
D     D
 C   C
  B B
   A
```

Are we done?

I don't think so. Here's a sabotaged version of the function:

```
module Diamond
where

diamond :: Char → [String]
diamond l = mirror  (map (mir∘pattern)  [0.. n])
    where
    pattern  i  = (sp (n−i))  ++ [ls !! i]  ++ (sp  i)
    sp n =  replicate  n  ' '
    ls  = [ 'A' .. l ]
    n = length  ls  − 1

    mir  xs = xs ++ " "  ++ ( tail  (reverse (xs ++ " ")))
    mirror  xs = xs ++ ( tail  (reverse xs))
```

Does it pass all the checks?

It does. Why wouldn't that be the case?

Look at what it yields:

```
runhaskell Main.hs D ↩
   A A
  B   B
 C     C
D       D
 C     C
  B   B
   A A
```

I see. We can prevent this by adding a property that states the precise height and width of a diamond.

We know what that is: if a diamond is made with n letters, then its height equals its width equals $2n-1$.

Okay. Let's write a property.

```
it  "has an height = width = N∗2−1" $ do
forAll   letter  $ λ l →
    let  d = diamond l
         height  = length  d
         width    = sum (map length d) ‘div‘  height
    in  height  ≡  width
    && height ≡  length  [' A '.. l ]  ∗  2 − 1
```

Ok, now the flawed version makes the checks fail. We are done.

Well maybe we can refactor this code a bit. This pattern function could be improved:

```
    pattern  i  = (sp  (n−i))  ++ [ls !! i ]  ++ (sp  i )
```

We don't need to compute each line this way. We could use list functions instead. You can try them using *ghci*. Try the inits :: [a] → [[a]] function for instance.

OK.

```
import Data.List ↩
inits "hello" ↩
["","h","he","hel","hell","hello"]
```

Now write a function that given a number n, returns space strings of size $0,1,\ldots,n-1$.

Easy:

```
let spaces n = take n (inits (repeat ' ')) ↩
spaces 5 ↩
[""," ","  ","   ","    "]
```

Now using
zipWith :: (a → b → c) → [a] → [b] → [c], and (:), you can insert each letter of the list into each space string.
Then you can concat that to the same space strings list, reversed.

I see

```
zipWith (:) ['A'..'E'] (spaces 5) ↩
["A","B ","C  ","D   ","E    "]
zipWith (++) (reverse (spaces 5)) it ↩
["    A","   B ","  C  "," D   ","E    "]
Prelude Data.List> putStrLn (unlines it) ↩
    A
   B
  C
 D
E
```

I works!!

Let's refactor the diamond function.

Ok.

```
module Diamond
where
import Data.List ( inits )

diamond :: Char → [String]
diamond l = mirror (map mirror diagonal)
    where
    diagonal = (reverse spaces)<+>(letters<:>spaces)
     letters  = ['A' .. l]
    n         = length letters
    spaces    = take n ( inits (repeat ' '))
    (<+>)     = zipWith (++)
    (<:>)     = zipWith (:)
    mirror l  = l ++ ( tail (reverse l))
```

And now we are done.

Let’s print a big diamond.

```
runhaskell Main.hs Z ↩
                         A
                        B B
                       C   C
                      D     D
                     E       E
                    F         F
                   G           G
                  H             H
                 I               I
                J                 J
               K                   K
              L                     L
             M                       M
            N                         N
           O                           O
          P                             P
         Q                               Q
        R                                 R
       S                                   S
      T                                     T
     U                                       U
    V                                         V
   W                                           W
  X                                             X
 Y                                               Y
Z                                                 Z
 Y                                               Y
  X                                             X
   W                                           W
    V                                         V
     U                                       U
      T                                     T
       S                                   S
        R                                 R
         Q                               Q
          P                             P
           O                           O
            N                         N
             M                       M
              L                     L
               K                   K
                J                 J
                 I               I
                  H             H
                   G           G
                    F         F
                     E       E
                      D     D
                       C   C
                        B B
                         A
```