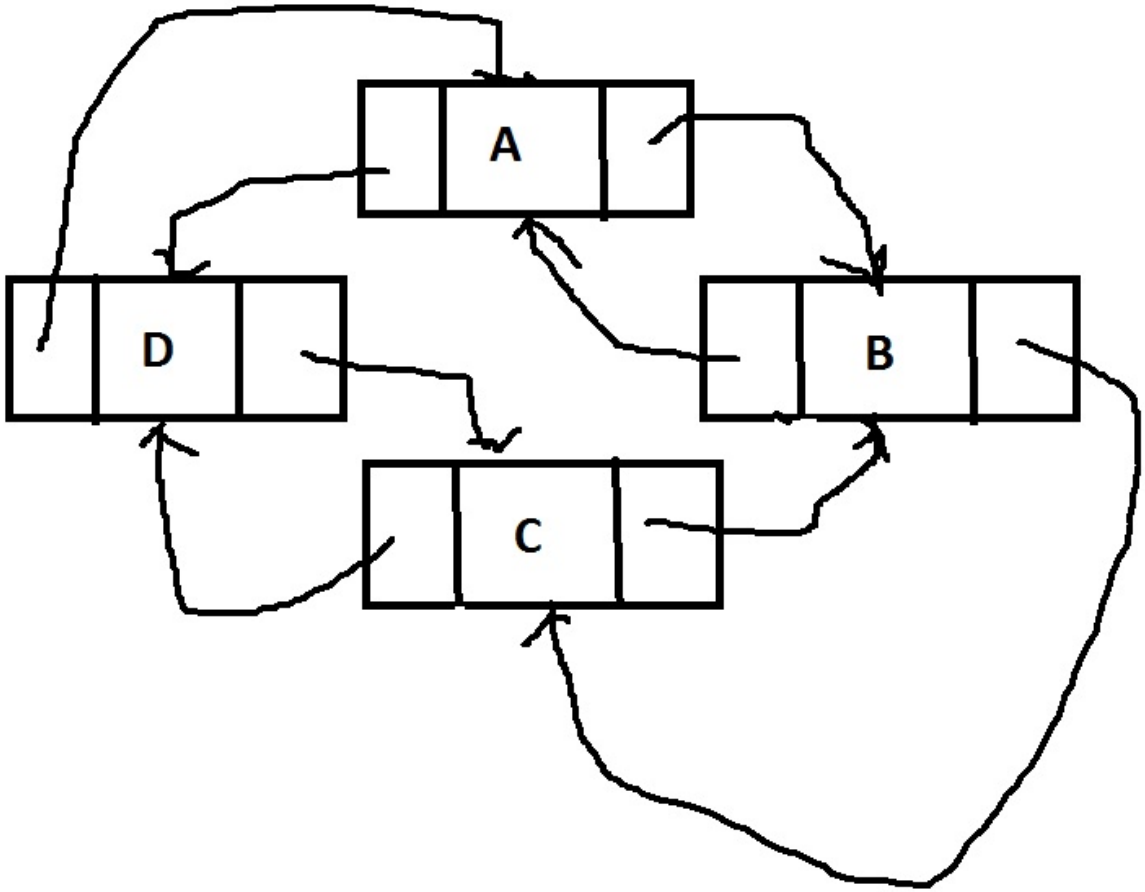
How to Link a List

Richard S. Bird

Department of Computer Science, Oxford University

Fun in the Afternoon, Feb 28th, 2012

Circular, doubly-linked lists



From haskellwiki/Tying_the_Knot

"It twists your brain a bit the first few times you do it, but it works fine."

```
data DList a = Nil | Cons (DList a) a (DList a)

mkCDlist :: [a] -> DList a
mkCDlist [] = Nil
mkCDlist as = first
          where (first,last) = go last as first

go :: DList a -> [a] -> DList a -> (DList a,DList a)
go prev []     next = (next,prev)
go prev (a:as) next = (this,last)
          where this = Cons prev a rest
                (rest,last) = go this as next
```

How I reacted to this code

I don't understand it.

I feel stupid.

I read the accompanying notes, but I still don't understand it.

I feel even more stupid.

Maybe I can derive it?

Maybe I can derive a better program?

Preliminaries: The Naming of Parts

```
elem :: DList a -> a
elem (Cons p a n) = a

prev, next :: DList a -> DList a
prev (Cons p a n) = p
next (Cons p a n) = n
```

Specification

What is the specification of a circular, doubly-linked list `x` whose elements are a given finite nonempty list `as`?

Answer: that there exists a finite list `xs` of dlists such that:

```
x = head xs &&
map elem xs = as &&
map prev xs = rotr xs &&
map next xs = rotl xs

where

rotr xs = [last xs] ++ init xs
rotl xs = tail xs ++ [head xs]
```

Specification

What is the specification of a circular, doubly-linked list `x` whose elements are a given finite nonempty list `as`?

Answer: that there exists a finite list `xs` of dlists such that:

```
x = head xs &&
map elem xs = as &&
map prev xs = rotr xs &&
map next xs = rotl xs

where

rotr xs = [last xs] ++ init xs
rotl xs = tail xs ++ [head xs]
```

Specification

What is the specification of a circular, doubly-linked list `x` whose elements are a given finite nonempty list `as`?

Answer: that there exists a finite list `xs` of dlists such that:

```
x = head xs &&
map elem xs = as &&
map prev xs = rotr xs &&
map next xs = rotl xs
```

where

```
rotr xs = [last xs] ++ init xs
rotl xs = tail xs ++ [head xs]
```

Does this specify a unique dlist for a given list?

(i) Take `as = "ABCD"` and `xs = [x1,x2,x3,x4]` where

```
x1 = Cons x4 'A' x2
x2 = Cons x1 'B' x3
x3 = Cons x2 'C' x4
x4 = Cons x3 'D' x1
```

(ii) Take `as = "ABCD"` and `xs = [x1,x2,x3,x4]` where

```
x1 = Cons x4 'A' x2
x2 = Cons x1 'B' x3
x3 = Cons x2 'C' x4
x4 = Cons x3 'D' x5
x5 = Cons x4 'A' x2
```

Also

(iii) Take as = "SOSO" and xs = [x1,x2,x3,x4] where

```
x1 = Cons x4 'S' x2
x2 = Cons x1 'O' x3
x3 = Cons x2 'S' x4
x4 = Cons x3 'O' x1
```

(iv) Take as = "SOSO" and xs = [x1,x2,x1,x2] where

```
x1 = Cons x2 'S' x2
x2 = Cons x1 'O' x1
```

Consequence

Define

```
cells :: DList a -> [DList a]
cells Nil = []
cells x   = x:cells (next x)
```

Then our specification is equivalent to

```
x = head xs &&
map elem xs = as &&
map prev xs = rotr xs &&
map next xs = rotl xs
where xs = take (length as) (cells x)
```

An easy lemma

Lemma. Let `xs = take n (cells x)`. Then

```
xs = zipWith3 Cons (map prev xs)
                   (map elem xs)
                   (map next xs)
```

Proof: By induction on `n`.

Corollary. If `x = mkCDlist as`, then

```
x = head xs where
  xs = zipWith3 Cons (rotr xs) as (rotl xs)
```

A glorious finish

Oh, does that mean

```
mkCDlist :: [a] -> DList a
mkCDlist [] = Nil
mkCDlist as = head xs
   where xs = zipWith3 Cons (rotr xs) as (rotl xs)   ?
```

Yes, provided we make `zipWith3` sufficiently lazy:

```
zipWith3 f ~(a:as) (b:bs) ~(c:cs)
       = f a b c:zipWith3 f as bs cs
zipWith3 f _      _      _      = []
```

A glorious finish

Oh, does that mean

```
mkCDlist :: [a] -> DList a
mkCDlist [] = Nil
mkCDlist as = head xs
   where xs = zipWith3 Cons (rotr xs) as (rotl xs)   ?
```

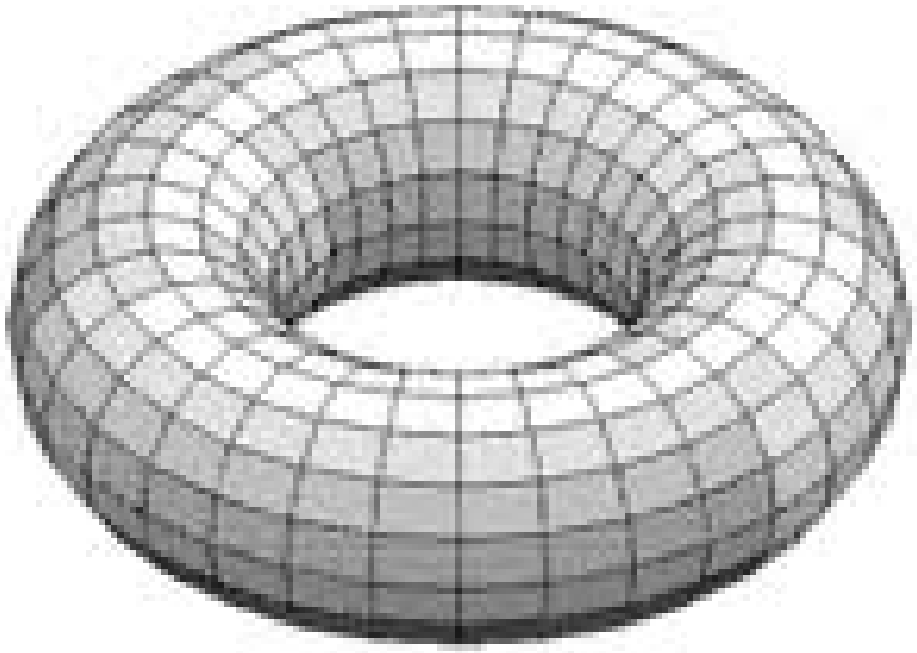
Yes, provided we make `zipWith3` sufficiently lazy:

```
zipWith3 f ~(a:as) (b:bs) ~(c:cs)
       = f a b c:zipWith3 f as bs cs
zipWith3 f _       _      _       = []
```

How I feel now

Much better, thank you.

Geraint's challenge: Build a torus



In more detail

Let `ass` be a list of lists representing a matrix. Build a torus `t = torus ass`, where

```
data Torus a = Cell {value :: a,
                     up,down,left,right :: Torus a}
```

You have to be able to tweet the answer!

```
mkTorus ass = head (head xss)
  where
  xss = zipWith5 (zipWith5 Cell)
        ass
        (rotr xss)
        (rotl xss)
        (map rotr xss)
        (map rotl xss)
```

In more detail

Let ass be a list of lists representing a matrix. Build a torus t = torus ass, where

```
data Torus a = Cell {value :: a,
                     up,down,left,right :: Torus a}
```

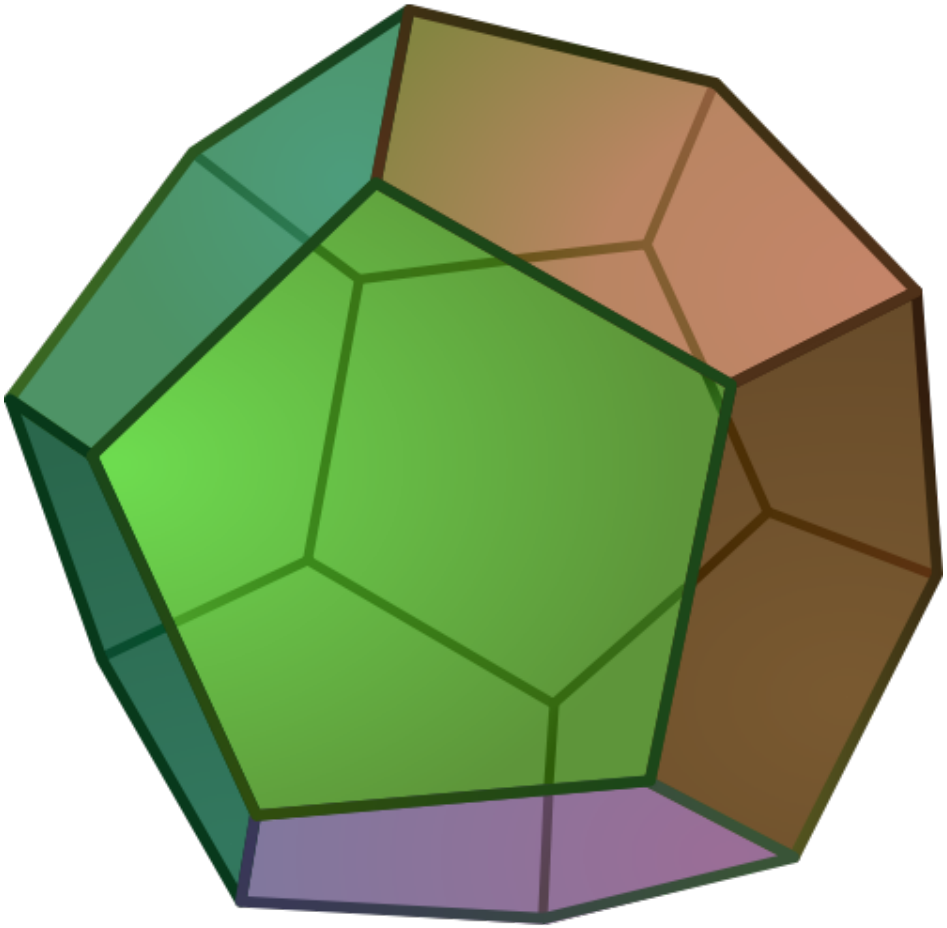
You have to be able to tweet the answer!

```
mkTorus ass = head (head xss)
  where
  xss = zipWith5 (zipWith5 Cell)
        ass
        (rotr xss)
        (rotl xss)
        (map rotr xss)
        (map rotl xss)
```

Building a torus, take 2

```
mkTorus ass = xa!(0,0)
  where
  xa = array ((0,0),(m-1,n-1)) ivs
  m = length ass
  n = length (head ass)
  ivs = [entry (i,j) a
        | (i,as) <- zip [0..m-1] ass,
          (j,a) <- zip [0..n-1] as]
  entry (i,j) a
      = ((i,j),Cell a (xa!u) (xa!d) (xa!l) (xa!r))
        where u = ((i-1) `mod` m,j)
              d = ((i+1) `mod` m,j)
              l = (i,(j-1) `mod` n)
              r = (i,(j+1) `mod` n)
```

The next challenge!



Acknowledgements

The real truth is that the specification I originally came up with for a circular, doubly-linked list was not as neat as the one above. I did manage to derive the original wiki code, and talked about it one Friday at a research meeting.
It was during the meeting that the "glorious finish" was discovered by Geraint Jones, Ralf Hinze and Jeremy Gibbons, and it only remained for me to reverse engineer the calculation.