We will now build a very simple model for wildfire propagation. So simple in fact, that we don't need to concern ourselves with its correctness. It's not correct.

In the model of our world will be a two dimensional grid made up of discrete cells with their own local state. We will then model the computation as an evolution rule, defined once for all grid cells. This rule may request the state of any cell in the previous generation.

To keep this challenging, we require that the grid be infinitely large, requiring the user to explicitly state the bounds of their computation, at which point the evolution rule will have to stop asking about other cells in the at a particular boundary. Without such a requirement, the number of grid cells to compute would exponentially grow as further generations ask about more and more cells. It's impossible to say what this growth factor would be without inspecting the evolution rule itself, as it will be free to request the state of any neighbouring cells from generation $n - 1$, which will need to be lazily computed and may in turn request information from generation $n - 2$, and so on.

To make such a computation feasible, we will require that the evolution rule for the system be pure, so that we can memoise generation n-1, and not have to re-calculate every previous generation at every step.

The first question we need to answer is – how will we model such a computation, where we have a rule that may require its own output at a previous generation. This happens to be a perfect use of the Store Comonad. Let us begin by defining our evolution rule.

First, some imports.

```
{-# LANGUAGE ViewPatterns #-}
{-# LANGUAGE RecordWildCards #-}
{-# LANGUAGE OverloadedStrings #-}
module Main where

import Control.Comonad.Store
import Data.MemoCombinators as Memo
import Linear.V2
import Data.Maybe
import Data.Map(Map)
import qualified Data.Map as Map
import qualified SDL
import qualified SDL.Cairo as Cairo
import qualified SDL.Cairo.Canvas as Canvas
import SDL.Cairo.Canvas (Canvas)
```

Recall that the monad State s a is a function from initial state s to final value a and final state s.

$s \to (a, s)$

These functions can be composed together to form a chain of computations that can be used to model a program's state.

If you flip some arrows around, this State monad becomes a Store Comonad. Observe that Store s a is some state s along with a function that computes an output based on that state.

$((s \to a), s)$

The Store is trivially a Functor, where $fmap\ g\ (Store\ f\ s) = Store\ (g\ .\ f)\ s$.

The Store is trivially a Comonad, were extract $(Store\ f\ s) = f\ s$, and $duplicate\ (Storefs) = Store\ (Store\ f)\ s$, an finally $extendf = fmap\ f\ .\ duplicate$.

We can define our evolution rule as a store algebra, where the configuration is the coordinate that we are computing. This algebra returns a new pixel state, and is able to look at any of the previous generation's pixel states with the provided Store. A the output of rule is uniquely determined by the function within Store that goes in (generation n-1's rule, which *could* be an entirely different rule), and the coordinates represented as a 2D vector.

Our PixelState will be the instantaneous average: temperature, and fuel remaining.

```
data PixelState = PixelState { temp :: !Double
                             , fuel :: !Double }
  deriving Show
```

The intuition here is that an algebra takes the entire lazily computed universe at a point in time, and computes the pixel for the next moment at a particular coordinate.

```
burn :: Store (V2 Int) PixelState -> PixelState
burn s =
```

To simulate the temperature of the fire, we will use very simple (broken) model of a constant flashpoint, heat output scaled by the logarithm of the fuel, and bound by a oxygen ceiling, heat loss proportional to the amount of heat, and heat transfer itself modelled by a "rolling", average, a sort of cellular blur filter.

```
    us { temp = new_temp
       , fuel = new_fuel}
  where
```

We are wherever the this store's coordinates points, at the previous generation.

```
    us = extract s
```

These are our eight neigbouring cells.

```
    V2 px py = pos s
    them = [ peek (V2 (px + dx) (py + dy)) s
           | dx <- [-1..1], dy <- [-1..1]
           , dx /= 0 || dy /= 0
           ]
```

If we're burning, we burn proportional to the amount of fuel, and limited by oxygen.

```
    our_temp = if flashpoint then temp us + burn_speed else temp us
    new_fuel = if flashpoint then fuel us - burn_speed else fuel us
    burn_speed = max 0 $ min 200 $ fuel us / 2
```

We now need to incorporate this heat with our surrounding cells.

```
    ambient_temp = sum (fmap temp them) / fromIntegral (length them)
    new_temp = max 0 $ our_temp + ((ambient_temp - our_temp) * heat_transfer) - heat_loss
```

Some magic numbers.

```
    heat_transfer = 0.1
    heat_loss = temp us / 100
    flashpoint = temp us > 500
```

With our governing rule defined, we need a base case for the computation. This will be our starting generation 0. The initial world is full of emptyPixels, aside from entries in a seed map.

```
emptyPixel :: PixelState
emptyPixel = PixelState { temp = 30, fuel = 4000 }

gen0 :: Store (V2 Int) PixelState
gen0 = store (\v2 -> fromMaybe emptyPixel (Map.lookup v2 seed)) (V2 0 0)
  where
      seed = Map.fromList [ (V2 20 20, emptyPixel { temp = 6000 })
                          , (V2 25 25, emptyPixel { temp = 5000 })
                          , (V2 40 40, emptyPixel { temp = 5000 })
                          , (V2 37 40, emptyPixel { temp = 6000 })
                          ]
```

By passing gen0 to to the burn algebra, we can now compute a single pixel of the next generation. If we were to iterate this twice, then gen_2 would request the output of neighbouring pixels at gen_1, which would in turn look up its neighbouring pixels in gen_0. As this process is iterated, we can see how we only compute the pixels relevant to our computation, and on demand.

This could get out of hand very quickly, and there is no point computing the entire universe to learn about 100x100 pixels, so, we bound our computation with a base-case that does not consult its neighbours, the bounds of our system can be thought of as a surrounding box of constant pixels.

```
bound :: (V2 Int, V2 Int) -> a -> (Store (V2 Int) a -> a ) -> Store (V2 Int) a -> a
bound (V2 bx by, V2 tx ty) def f s@(pos -> V2 x y)
  | x < bx || y < by || x > tx || x > ty = def
  | otherwise = f s
```

Using data-memocombinators, we can memoise pure functions without dealing with IO. If finer grained control of what we memoise is required, a transformer over might be cleaner.

```
memoise :: Store (V2 Int) a -> Store (V2 Int) a
memoise (runStore -> (f, s)) = store (Memo.wrap to from memoPair f) s
  where
      to (x,y) = V2 x y
      from (V2 x y) = (x,y)
      memoPair = Memo.pair Memo.integral Memo.integral
```

Now we define our simulation parametrised by a store algebra, and an initial store. We feed result of successive uniform applications the algebra over the entire universe back into itself. simulation alg base !! 10 will retrieve the tenth generation. Only this tenth generation store is peeked into will the computation actually take place, when all nine previous generations are calculated bottom up.

```
simulation :: (Store (V2 Int) a -> a) -> Store (V2 Int) a -> [Store (V2 Int) a]
simulation f base = iterate (extend f . memoise) base
```

As suggested, we'll need something to peek into a store. Given a bounding box, we evaluate a store at every point within.

```
computeWithin :: (V2 Int, V2 Int) -> Store (V2 Int) a -> [(V2 Int, a)]
computeWithin (V2 x0 y0, V2 xn yn) w = fmap (\s -> (s, peek s w)) coords
  where
    coords = do
      dx <- [x0..xn]
      dy <- [y0..yn]
      return $ V2 dx dy
```

Putting it all together now.

```
main :: IO ()
main = do
```

We create a lazy value, frames, which is a list of "windows" into our universe.

```
    let bounds = (V2 10 10, V2 50 50)
    let results = simulation (bound bounds emptyPixel burn) gen0
    let frames = fmap (computeWithin bounds) results
```

Now for the quick and nasty rendering of our simulation, we will render a cairo texture within SDL. Get a window and show it.

```
    SDL.initialize [SDL.InitVideo]
    window <- SDL.createWindow "Burning the world" SDL.defaultWindow
    SDL.showWindow window
```

Get a renderer to put our texture on.

```
    renderer <- SDL.createRenderer window (-1) SDL.defaultRenderer  { SDL.rendererType = SDI
```

Associate our cairo texture with the renderer.

```
    texture <- Cairo.createCairoTexture' renderer window
    loop renderer texture frames
  where

    loop renderer texture (frame:frames) = do
```

A rendering loop like any other rendering loop. Clear the buffer.

```
        SDL.clear renderer
```

Draw on the buffer.

```
        Canvas.withCanvas texture $ do
          mapM_ (uncurry drawPixel) frame
```

Swap the buffers.

```
    SDL.copy renderer texture Nothing Nothing
    SDL.present renderer
    loop renderer texture frames
```

We represent each pixel by a circle, bigger and redder circles are hotter.

```
drawPixel :: V2 Int -> PixelState -> Canvas ()
drawPixel coords PixelState{..} = do
  let heat_scaled = max 1 (2000 / temp)
  let redness = round $ 255 / heat_scaled
  Canvas.fill (Canvas.red redness)
  Canvas.circle (fmap ((*10) . fromIntegral) coords) (10 / heat_scaled)
```