Program Design and Data Structures, 20.0 c

Course code: 1DL201, Report code: DL201, 67%, DAG, NML,
week: 44 - 02 Semester: Autumn 2017
week: 03 - 11 Semester: Spring 2018

EXPLORATIONS

This information is not available in English. Now showing the Swedish version.

The problems and links presented here offer you a place to look for extra things to do with Haskell, for example, if you have finished a lab and are waiting to have your work reviewed, or you want to hone your Haskell skills at home.

LINKS

- If you are looking for problems to solve to hone your coding skills, a good place to start is 99 Haskell Problems.
- Another place with all kinds of problems of all levels is CodeWars.
- The Project Euler website has hundreds of problems to solve using programming. Most are fairly mathematical in nature and the level of difficulty ranges from straightforward to extreme.
- Lots of problems of different levels for you to try out are available at the Rosetta Code wiki. In fact, solutions (in many different languages) are also available on that site, if you prefer to see how someone else solves a problem.
- Problems in the realm of bioinformatics can be found on http://rosalind.info/. Some of these will require the ability to read input from either the terminal or a file. Have a look at http://learnyouahaskell.com/input-and-output to help you with that. We'll cover input and output after Christmas.
- Why not try your coding skills in a coding game. (Select the Haskell option, obviously.)

Exploration 1

Write functions to solve the following problems.

1. Compute one of the solutions to $ax^2 + bx + c = 0$.

   let quadsolve a b c = ...

   Find solutions to $10x^2 + 12x - 22 = 0$, $2x^2 + 4x - 4 = 0$→, and $x^2 - 12x - 4 = 0$→.

   Use Google to find the relevant formula and use the Haskell documentation to find the functions you need.

2. Compute the distance between two points (x_0,y_0) and (x_1,y_1).
   →
   let distance x0 y0 x1 y1 = ...

3. Pairs of values in Haskell are represented as (a,b). For example, (10, "Hello") is a pair of an integer and a string. We will use pairs to represent intervals: an (open) interval `(a,b)`, where `a` and `b` are numbers of some sort such that $a < b$, is the set of numbers $\{ x \mid a < x \text{ and } x < b \}$.

   Compute whether two intervals overlap.

let overlap (a,b) (c,d) =

How would you test your function (i.e., does it compute correct results)?

Exploration 2

Computer graphics are ubiquitous these days. They are at the heart of computer games, data visualisation, user interfaces, etc. There are many different computer graphics packages available for Haskell. One of the simpler packages is called *Gloss*. It can display drawings, animations and simulations all using a relatively simple interface. (On the other hand, Gloss is fairly limited in what it can do.)

From time to time you'll get the chance to play around with Gloss in lab assignments. You might even want to use it in your project later in the course.

Gloss is already installed on the student machines. If you wish to use Gloss on your **own machine**, issue the following commands (assuming that Haskell has been installed). **DO NOT RUN THESE COMMANDS ON THE DEPARTMENT'S MACHINES:**

```
> cabal update
> cabal install gloss-examples
> gloss-styrene
```

The first command updates the software for installing Haskell-related libraries. If that software is missing, you need to Google "installing cabal". The last command simply checks that it has been installed correctly by running one of the examples.

Gloss's web page is here: http://gloss.ouroborus.net/

The top-level documentation for Gloss is located here: https://hackage.haskell.org/package/gloss-1.9.4.1/docs/Graphics-Gloss.html.
This documentation provides some information to get you started, along with the top-level functions provided by Gloss. *Most of it can be ignored for now.*

A simple program using Gloss is the following:

```
import Graphics.Gloss
main = display (InWindow "Nice Window" (200, 200) (10, 10)) white (circle 80)
```

Gloss separates the code for specifying the shapes to draw from the code that does the drawing. In the above code, "circle 80" specifies a particular circle (which is of type Picture). The remainder of the code puts it on the screen.

Enter that code into a file, say, code.hs and save it. Compile the file using

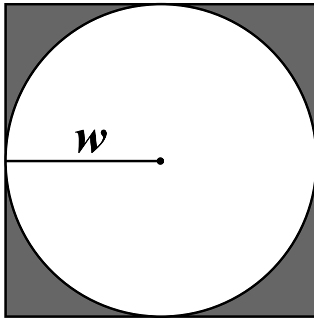
```
> ghc -threaded code.hs
```

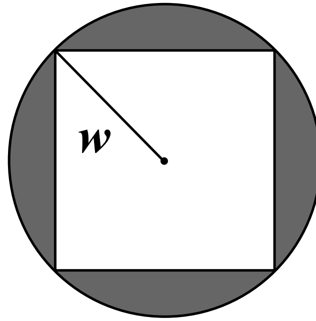
and run it using

```
> ./code
```

The documentation for picture elements (circles, etc) is located here: https://hackage.haskell.org/package/gloss-1.9.4.1/docs/Graphics-Gloss-Data-Picture.html

Using Gloss, write functions to draw the two following shapes:





Hint: The function line will allow you to draw a line through a sequence of points. For example,

line [(10, 20), (30, 40), (20, 50), (15, 35), (10, 20)]

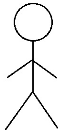
corresponds to a polygon going through the points mentioned. How can you display it on the screen?

The code

pictures [circle 20, circle 30]

corresponds to a Picture made from multiple other pictures.

Play around with Gloss. For instance, draw a stick figure on a piece of paper, work out the coordinates of the various parts of it, and use Gloss's picture constructors to construct the shapes that represent the stick figure. Then display the figure on your screen.



Find something else to draw and draw it!

Play around with other aspects of Gloss. Change the colour of your shapes. Make pretty patterns with multiple overlapping circles or squares using the translate and rotate functions.

A note regarding Gloss. It seems that it does not work from `ghci`. Haskell programs using Gloss should be written in a file and compiled using `ghc -threaded YourFile.hs`.

Exploration 3

This exploration will continue working with the graphics library Gloss. See the previous instructions for how to run Gloss.

QUESTION 1

The goal of this exercise is to compute a minimum bounding box for a polygon.

A polygon can be represented in two ways:

line [(10, 20), (30, 40), (20, 50), (15, 35), (10, 20)]

gives the outline of a polygon -- it requires that the first and last points are the same.

polygon [(10, 20), (30, 40), (20, 50), (15, 35)]

gives a filled polygon.

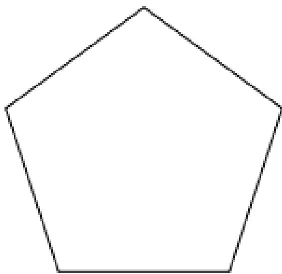
In both cases, the polygon is represented by its vertices -- a list of points.

A minumum bounding box for a shape is the smallest rectangle which contains all of the points of the shape inside it (the shape may touch the boundary of the
rectangle).

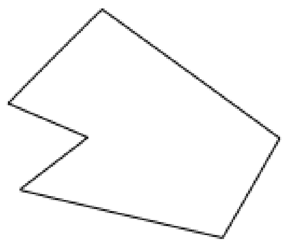
1. Given a polygon, represented as a list of points such as [(10, 20), (30, 40), (20, 50), (15, 35), (10, 20)], write a function to compute the bounding box for the polygon -- you need to think about how you will represent a rectangle.
2. Write code to display both the polygon and the bounding box.

QUESTION 2

A convex polygon is a polygon in which no line between two points on the boundary ever goes outside the polygon. Alternatively, all interior angles are less than or equal to 180 degrees. Here's a picture of a convex polygon:



Here's one that's not convex (concave):



Think about how you would compute the area of a convex polygon. How would you convert this into an algorithm?

QUESTION 3

Algorithms such as these are the basis of Computational Geometry and Computer Graphics. Both topics have a heavy visual component, but also require specialised algorithms to get quick results. If you are interested, you might want to look these topics up on the web and see what they are about.

Exploration 4

This exploration continues the line focussing on computational geometry/computer graphics. Previously, we explored general shapes and thought about how to take the area of a convex polygon. A polygon is made up of a sequence of points, where the first and last points coincide (so that the graphic library Gloss's function line can accurately render the polygon). It doesn't take long to realise that not every list of points defines a sensible polygon.

The goal of this exploration is to write functions that firstly check whether a polygon is valid and secondly whether a valid polygon is convex. These functions are useful to ensure that certain library functions work validly or that effient algorithms can be chosen. For instance, trying to take the area of an invalid polygon would produce nonsense. In addition, if you know that the polygon is convex, then an efficient algorithm can be used to test the area.

A 2D-point is given by the type

type Point = (Float, Float)   -- Float is like Double, but with less precision

A (possible) polygon is given by type

type Polygon = [Point]

Look at Lab 2 for instructions on how to render this to the screen using the functions line and display from the Gloss library.

1. Develop the following functions:

- valid :: Polygon -> Bool

  Given a list of points, determine whether it form a valid polygon. valid p is true whenever p is a valid polygon.

  Hint: Think about what conditions are required to for a list of points to be be a valid polygon. Try lots of examples and see which ones are valid and which are not, in order to find the conditions required.

- convex :: Polygon -> Boolean

  The aim of this function is to take a valid polynomial and test whether it is a convex polynomial.

  A precondition for this function is that the input is a valid polygon. This assumption simplifies the code you need to write.

  You may need to use Google to find out when a polynomial is convex. Develop an algorithm first before trying to code it up.

2. Think about testing. How would you test the correctness of the two functions above?

Exploration 5

This exploration briefly explores population dynamics, which is a field that studies the changes in size and age of populations.
The results of these explorations do not need to be shown to a teaching assistant.

Population dynamics uses simple (and not so simple) models of the populations of one or more different groups.

We'll explore a simple (but not too simple) model of the population of a single group. Later we'll look at more complicated models, including one that will appeal to fans of The Walking Dead.

The following simple model is called Discrete Time Logistical Model. This model is based on the following formula, which gives the population over different time steps:

$$N_{t+1} = N_t + r\,N_t\,(1 - N_t k)$$

N_t is the population at time t and N_{t+1} is population at the next time step. The model has two parameters that can be modified to see different system dynamics (= behaviour).

Parameter r is the **per capita growth rate**. This is the most interesting parameter to adapt, as it more wildly changes system dynamics. Parameter r can be any value, though it typically is positive and fairly small. As r gets larger, say 3, interesting system dynamics occurs. As it gets larger still, the system very quickly collapses. In fact, a small change in r can often result in large changes in the system; this means that the system exhibits chaotic behaviour.

Parameter k is the **system capacity**. After the population exceeds the capacity, the population will start to decrease.

The goal of this exploration is to play around with the given population dynamics model. This model has, essentially 3 parameters to vary r, k and the initial population N_0, but r gives the most interesting variation, so select some values of k and N_0 and vary r. By "play around with", we mean to select different values of the parameters to see which result in interesting changes to behaviour.

In order to interpret what each change of a parameter does, a good approach is to visualise the population data generated.

- Open a web browser at http://www.wolframalpha.com/.
- In the little text field in the middle of the page enter
  →`plot ([10000,19595,37447,68187,113576,163145,193509,199852,200001,200000])`
  →and press 'return'.
- You see a graph appear.

The following code is enough to get you started. In particular, tryModel can be run multiple times with different values of *r*. Note that tryModel will be run for 20 generations. Perhaps you need to increase that number to see interesting variations.

```
-- helper: ensures that population will never be negative
cutoff :: Double -> Double
cutoff n | n < 0 = 0
         | otherwise = n

-- this function computes the function above, thus running the simulation for one round
dtlDynamics :: Double -> Double -> Double -> Double
dtlDynamics r k n = cutoff $ n + r * n * (1 - n / k)

-- here are some sample models set up with given parameters r and k
modelA :: Double -> Double
modelA = dtlDynamics 1.01 200000

modelB :: Double -> Double
modelB = dtlDynamics 0.9 20000

modelC :: Double -> Double
modelC = dtlDynamics 0.3 20000

-- this function runs a model for a given number of generations
-- and given initial population
runModel :: (Double -> Double) -> Int -> Double -> [Integer]
runModel model generations init = map round $ take generations (iterate model init)

-- these will give you 20 generations of the models described above with different
-- initial populations
exampleA1 :: [Integer]
exampleA1 = runModel modelA 20 10000

exampleA2 :: [Integer]
exampleA2 = runModel modelA 20 500

exampleB1 :: [Integer]
exampleB1 = runModel modelB 20 10000

exampleB2 :: [Integer]
exampleB2 = runModel modelB 20 500

exampleC1 :: [Integer]
exampleC1 = runModel modelC 20 10000

exampleC2 :: [Integer]
exampleC2 = runModel modelC 20 500

exampleC3 :: [Integer]
exampleC3 = runModel modelC 20 30000
```

```
-- allows you to try different values of r
tryModel :: Double -> [Integer]
tryModel r = runModel (dtlDynamics r 20000) 20 500
```

Exploration 6

This exploration is again related to computational geometry. The Jordan curve theorem states, in essence, that any closed non-self-intersecting curve on a plane has an inside and an outside. That may seem obvious, but how would you prove it? Well, that's a topic for mathematicians. In an earlier lab, you may have written a function that tested whether a collection of points described a valid polygon (for example, when drawn using Gloss's `line` function). Let's assume that such a function exists. A valid polygon satisfies the Jordan curve theorem. This means that every point on the plane will be unambiguously either inside the polygon, outside the polygon, or on the boundary of the polygon.

The goal of this exploration is to write a function that will take a valid polygon and a point and tell you where the point lies in relation to the polygon. First start by understanding how to solve the problem using pen and paper. Then try to work out an algorithm, breaking the steps into smaller and smaller pieces. Then write the function → most likely this will take quite some time.

The function can return an element of the following data type:

```
data Location = Inside | Outside | Boundary
```

Write the function locatePoint poly p which takes a valid polygon poly a point p and returns the element of the Location datatype that indicates where the point is.

```
locatePoint :: Polygon -> Point -> Location
```

where

```
type Point = (Float, Float)
type Path = [Point]
type Polygon = Path -- with constraint that first and last points are the same.
```

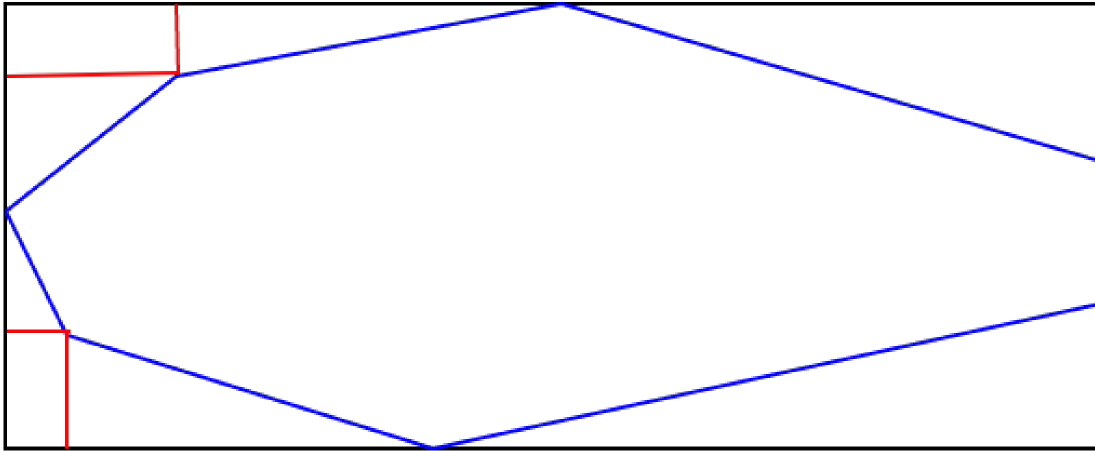
The best way to approach this is using pen and paper. Draw a shape and a point. Checking whether the point is on the boundary is fairly easy, so work out how to do that first. For the remainder, I offer only a hint. Draw a line from one edge of your page to the other that passes through your point. Do this for a number of shapes and different point positions. Can you see a pattern that could help you determine whether something is on the inside or outside?

You can test the correctness of your algorithm by drawing the polygon and point using the following code.

```
import Graphics.Gloss
mypoly = [(0,0),(20,20),(0,20),(0,0)]
mypoint = (15,10)
myshapes poly (x,y) = scale 3 3 $
  pictures [line poly, color red $ translate x y $ circle 0.5]
main = display (InWindow "Nice Window" (200, 200) (10, 10)) white
 (myshapes mypoly mypoint)
```

EXPLORATION 7

Write a function to compute the area of a convex polygon → an earlier exploration asked you to think of an algorithm for doing this. The following image is suggestive of what that algorithm should do. Here the polygon is in blue, the bounding box is black, and the red marks some of the triangles and squares of the bounding box that are not included in the polygon's area.



EXPLORATION 8

This exploration will explore animations using Gloss. Remember, that Gloss (probably) doesn't work in `ghci`. You will need to put your code in a file compile it using `ghc -threaded YourFile.hs`.

The Haskell library Gloss that we've seen in a number of these explorations supports animations. An animation is represented by a fairly simple type: Float -> Picture. The idea is that the Float argument represents the number of seconds (or part thereof) since the start of the animation and the return value is the picture to draw at that time point. The animation proceeds by repeatedly calling this function to get new pictures to draw over time.

The Gloss function for doing the animation is called animate. It has the following signature:

```
animate :: Display -> Color -> (Float -> Picture) -> IO ()
```

The Display is the display mode or the window in which to draw the animation, the Color argument is the background function, and the Float -> Picture argument describes the animation. The return value has type IO () which indicates that the animation performs Input/Output, which in this case is the rendering of the animation on the screen.

The following code includes some simple animations and some combinators (functions) for manipulating and combining animations. Each of the functions utilises some of the functionality in the Gloss library, such as scaling, rotating and translating pictures.

```
import Prelude hiding (until)
import Graphics.Gloss

-- simple animations
type Anime = Float -> Picture

-- rotates the text "I'm hungry"
hungry :: Anime
hungry t = rotate (t * 60) $ scale 0.25 0.25 $ text "I'm hungry"

-- rotates and grows a square
squarer :: Anime
squarer t = rotate (t * 60) $ scale t t $ line [(0,0), (30,0), (30,30), (0,30), (0,0)]

-- grows a circle
circler :: Anime
circler t = scale t (-t) $ Circle 50

-- displays the number of seconds
seconds :: Anime
seconds t = translate scale 0.25 0.25 $ text $ show t
```

```
-- stops an animation after a certain time
until :: Anime -> Time -> Anime
until f stopTime t = if t < stopTime then f t else Blank

-- runs two animations together
together :: Anime -> Anime -> Anime
together f g t = pictures [f t, g t]

-- runs for a certain time, then runs another animation
after :: Anime -> Time -> Anime -> Anime
after f stopTime g t = if t < stopTime then f t else g (t - stopTime)

-- a complex (but uninteresting) animation
anim = until (after (together hungry squarer) 5 (circler)) 20

-- the main function for running animations.
-- replace "anim" by your animation
-- "main" must be called "main"
-- compile this file using ghc -threaded FileName.hs
main = animate (InWindow "Animation" (600, 600) (10, 10)) white $ anim
```

Play around with this code, by changing the different animation functions to see what they do. Divising your own simple animations.

Consider the functions until, together and after which give different ways of manipulating and combining animations. Can you think of different ways of manipulating and combining animations? Code them up.

This style of animation is very close to the notion of *Functional Reactive Programming*. Google it and also f*unctional feactive animations.*

Exploration 9

Have you ever spent time solving logic puzzles like the following (From Thinking as Computation by Hector J. Levesque)?

> Chris, Sandy and Pat have distinct occupations and play distinct musical instruments. Their occupations are doctor, lawyer and engineer, and the instruments they play are piano, flute and violin. Also,
>
> 1. Chris is married to the doctor.
> 2. The lawyer plays the piano.
> 3. Chris is not the engineer.
> 4. Sandy is a patient of the violinist.
>
> Who plays the flute?

These can be a lot of fun and can really make you think. But who needs thinking when we've got computers. Am I right?

Simpler logic problems like this can easily be encoded in Haskell using list comprehensions, both to generate the possible answers and to filter incorrect ones using constraints.

1. Put the code below in a file, and run the function `main` to see the solution. Try to understand what is going on (ignore the printing code). The key thing to realise is that data values for only one of the dimensions (Person) are used, and other dimensions (Occupation, Musical Instrument) are just Haskell variables that hold such data values.

```
import Data.List(nub, delete)
```

```
data Person = Chris | Sandy | Pat deriving (Eq, Show)

people = [Chris, Sandy, Pat]

distinct3 a b c = a /= b && a /= c && b /= c

married x y = x /= y
patientOf x y = x /= y

solution = [(doctor, lawyer, engineer, piano, violin, flute) |
            -- the next lines generate candidates for each of the
            -- jobs and instruments
            doctor <- people,
            lawyer <- people,
            engineer <- people,
            piano <- people,
            violin <- people,
            flute <- people,

            -- the next two lines ensure that the jobs and the instruments
            -- correspond to distinct people
            distinct3 doctor lawyer engineer,
            distinct3 piano violin flute,

            -- 1. Chris is married to the doctor
            Chris `married` doctor,

            -- 2. The lawyer plays the piano.
            lawyer == piano,

            -- 3. Chris is not the engineer
            Chris /= engineer,

            -- 4. Sandy is a patient of the violinist
                   -- (hence, the violinist is a doctor)
            Sandy `patientOf` violin,
            violin == doctor]

-- prints the solution nicely
printSolution (doctor, lawyer, engineer, piano, violin, flute) =
    do
        putStrLn $ "Doctor is " ++ show doctor
        putStrLn $ "Lawyer is " ++ show lawyer
        putStrLn $ "Engineer is " ++ show engineer
        putStrLn $ "Pianist is " ++ show piano
        putStrLn $ "Violinist is " ++ show violin
        putStrLn $ "Flautist is " ++ show flute

-- run this
main = printSolution $ head solution
```

You can jump to question 4 and try to encode your own problem in Haskell, or look at the examples that follow for more inspiration and tricks.

2. Here is another example very similar to the first. It involves more complex constraints about which people are married or not, so the solution needs to generate all possible ways that they can be married or single (see function `couplings`). Here is the new set of constraints (with the same background story). Only the third constraint differs from above.

1. Chris is married to the doctor.
2. The lawyer plays the piano.
3. Pat is not married to the engineer. ****
4. Sandy is a patient of the violinist.

Here is the code solving this puzzle. Put it in a separate file, run it to solve the puzzle, and try to understand how it works (ignoring the code for printing).

```
import Data.List(nub, delete)

-- For more see: http://www.logic-puzzles.org/


data Person = Chris | Sandy | Pat deriving (Eq, Show)

people = [Chris, Sandy, Pat]

distinct3 a b c = a /= b && a /= c && b /= c


-- this problem is trickier, because it requires knowledge of
-- who is married and who is not, rather than just the fact that
-- if a is married to b, then a is not the same person as b,
-- as in the previous example
-- The function couplings below generates all possible ways
-- in which a group of people could be married (including being single)

patientOf x y = x /= y


-- couplings gives all ways of coupling up/marrying a list of persons,
-- including the cases when the person is not married
couplings :: [Person] -> [[(Person,Person)]]
couplings [] = [[]]
couplings (a:as) = couplings as ++
  concat [map ((a,p):) (couplings (delete p as)) | p <- as]


isMarried p [] = False
isMarried p ((a,b):cs) = if p == a || p == b then True else isMarried p cs

areMarried (p,q) [] = False
areMarried (p,q) ((a,b):cs) = if (p,q) == (a,b) || (q,p) == (a,b)
                          then True
                          else areMarried (p,q) cs

solution = [(doctor, lawyer, engineer, piano, violin, flute) |
          doctor <- people,
          lawyer <- people,
          engineer <- people,
          piano <- people,
          violin <- people,
          flute <- people,
          distinct3 doctor lawyer engineer,
          distinct3 piano violin flute,
          couples <- couplings people,
```

```
            areMarried (Chris, doctor) couples,
            lawyer == piano,
            not $ areMarried (Pat, engineer) couples,
            Sandy `patientOf` violin,
            violin == doctor]

printSolution (doctor, lawyer, engineer, piano, violin, flute) =
    do
        putStrLn $ "Doctor is " ++ show doctor
        putStrLn $ "Lawyer is " ++ show lawyer
        putStrLn $ "Engineer is " ++ show engineer
        putStrLn $ "Pianist is " ++ show piano
        putStrLn $ "Violinist is " ++ show violin
        putStrLn $ "Flautist is " ++ show flute

main = printSolution $ head solution
```

3. Here is a third example taken from http://www.logic-puzzles.org/:

"Against the Grain" offers hand-made wooden furniture at reasonable prices. Each item is made by an in-house employee. Using only the clues that follow, match each item to the employee who crafted it, and determine its price and the type of wood used to make it.

1. Of the $350 item and the item made of elm, one was crafted by Gwen and the other was crafted by Patsy.
2. Of the $275 piece and Patsy's item, one was made of walnut and the other was made of elm.
3. Rosemary's item is either the $350 piece or the item made of ash.
4. Patsy's piece doesn't cost $325.
5. The item made of walnut costs 25 dollars more than Hope's piece.
6. Latasha's item costs 25 dollars more than the piece made of oak.

Here is an approach to solving the problem:
A. Make a list of all the entities involved

> Woodworkers: Gwen, Hope, Latasha, Patsy, Rosemary
> Woods: ash, chestnut, elm, oak, walnut
> Amounts: $250, $275, $300, $325, $350

Let's refer to these (Woodworkers, Woods, Amounts) as the **dimensions**.

B. Select one dimension as the data values and represent the other dimensions as variables. The data values will be used to link the other dimensions together, thereby representing a solution. In this particular problem, using the value in dollars are the core dimension, because constraints on it can be phrased in Haskell using arithmetic.

C. Formulate constraints in terms of variables and data values from the selected dimension.

Put this example in a separate file, run it to solve the puzzle, and try to understand how it works (again ignoring the code for printing).

```
import Data.List(nub, deleteBy)

amounts = [250, 275, 300, 325, 350]

-- ensure that a b c d e are distinct values
distinct5 a b c d e = (length $ nub [a,b,c,d,e]) == 5

solution = [(gwen, hope, latasha, patsy, rosemary,
           ash, chestnut, elm, oak, walnut) |
```

```
          -- generate candidates for the people
          gwen <- amounts,
          hope <- amounts,
          latasha <- amounts,
          patsy <- amounts,
          rosemary <- amounts,

          -- ensure that they are distinct
          distinct5 gwen hope latasha patsy rosemary,

          -- generate candidates for the kind of wood
          ash <- amounts,
          chestnut <- amounts,
          elm <- amounts,
          oak <- amounts,
          walnut <- amounts,

          -- ensure that they are distinct
          distinct5 ash chestnut elm oak walnut,

          -- 1. Of the $350 item and the item made of elm, one was
          -- crafted by Gwen and the other was crafted by Patsy.
          (350 == gwen && elm == patsy) || (350 == patsy && elm == gwen),

          -- 2. Of the $275 piece and Patsy's item, one was made of
          -- walnutand the other was made of elm.
          (275 == walnut && patsy == elm) ||
          (275 == elm && patsy == walnut),

          -- 3. Rosemary's item is either the $350 piece or the item made
          -- of ash.
          (rosemary == 350 || rosemary == ash),
          ash /= 350,  -- is this required? implied?

          -- 4. Patsy's piece doesn't cost $325.
          patsy /= 325,

          -- 5. The item made of walnut costs 25 dollars more than
          -- Hope's piece.
          walnut == hope + 25,

          -- 6. Latasha's item costs 25 dollars more than the piece
          -- made of oak.
          latasha == oak + 25
          ]


-- some code to make the solution more comprehensible
printSolution (gwen, hope, latasha, patsy, rosemary,
               ash, chestnut, elm, oak, walnut) =
  let res = fixup [(gwen, "Gwen"), (hope, "Hope"), (latasha, "Latasha"),
                   (patsy, "Patsy"), (rosemary, "Rosemary")]
                  [(ash, "Ash"), (chestnut, "Chestnut"), (elm, "Elm"),
                   (oak, "Oak"), (walnut, "Walnut")]
  in
  do
    mapM_ printOne res
```

```
    putStrLn $ "=============================="
      where printOne (i, name, material) = putStrLn $ name ++ " made an " ++ material
                                 ++ " item costing $" ++ show i

-- run this to see the solution(s)
main = mapM_ printSolution solution

fixup :: [(Int, a)] -> [(Int, b)] -> [(Int, a, b)]
fixup [] _ = []
fixup ((i,a):as) bs = (i,a,b):fixup as (deleteBy (\(i,_) -> \(j,_) -> i == j) (i,b) bs)
  where Just b = lookup i bs
```

4. Go to http://www.logic-puzzles.org/ and generate a puzzle. Click on the link "**Solve a Logic Puzzle**" and request a puzzle on a small grid size of easy or moderate difficulty. Copy the text into a file and convert the problem into Haskell.

EXPLORATION 10

The goal of this exercise is to use **binary search trees** to compute an inverted index for a document. An inverted index is a mapping from the words that appear in the document to their location, which in this case will be their page number.

You will be provided with a binary search tree, the document data type, some small documents, and a function to print out the resulting index. Your task is to fill in the gaps.

A document will be a collection of pages, each consisting of a page number and a collection of words. For example, the document:

> page 1: good dog big dog
> page 2: good cat big cat
> page 3: cat dog cat dog

is represented as

```
doc0 = [(1, ["good", "dog", "big", "dog"]),
        (2, ["good", "cat", "big", "cat"]),
        (3, ["cat", "dog", "cat", "dog"])]
```

It's index is:

> big: 1, 2
> cat: 2, 3
> dog: 1, 3
> good: 1, 2

Think about how you can use the binary search tree to collect the information you need. How will you process a single word? A page? The whole document?

You will need to adapt the binary search tree structure below to include additional information. You will also need to adapt how information is inserted into the tree.

```
import Data.List(intercalate) -- to help with printing

-- a binary search tree, but it contains only keys
data BSTree k = Leaf | Branch (BSTree k) k (BSTree k)
  deriving Show

insert :: k -> BSTree k -> BSTree k
insert k Leaf = Branch Leaf k Leaf
insert k (Branch left k' right)
```

```
  | k == k' = Branch left k' right
  | k < k' = Branch (insert k left) k' right
  | otherwise = Branch left k' (insert k right)

inorder :: BSTree k -> [k]
inorder Leaf = []
inorder (Branch left k right) = inorder left ++ [k] ++ inorder right

-- the document types
type PageNumber = Integer
type Document = [(PageNumber, [String])]
type Index = [(String, [PageNumber])]


-- YOUR FUNCTION HERE (apologies for shouting)
invertedIndex :: Document -> Index
invertedIndex = undefined


-- for printing out an index included
printIndex :: Index -> IO ()
printIndex index = mapM_ putStr $ map f $ index
  where f (k, d) = k ++ ": " ++ intercalate ", " (map show d) ++ "\n"


-- example documents
doc0 = [(1, ["good", "dog", "big", "dog"]),
        (2, ["good", "cat", "big", "cat"]),
        (3, ["cat", "dog", "cat", "dog"])]

doc1 = [(1, words "how now brown cow"),
        (2, words "how is brown cow now"),
        (3, words "how are you brown cow these days"),
        (4, words "supercilious splendiferous cow")]

doc2 = zip [1..] $ map words $ lines "it was the best of times\nit was the worst of times\nit was the age of wisdom\nit was the age of foolishness\nit was the epoch of belief\nit was the epoch of incredulity\nit was the season of light\nit was the season of darkness\nit was the spring of hope\nit was the winter of despair"
```

EXPLORATION 11

The following code implements two type classes Encode and Decode that have functions that will encode any element of a type into a string of Os and 1s and decode a string of Os and 1s into an element of the type. The two key functions satisfy the property *decode $ encode x = x :: T*, meaning that if I encode a value, then decode the resulting bitstream I get the same value back.

A number of instances of the type classes have been defined for some common data types. Have a look at the code, run it, try to understand what's going on. There are some questions after the code.

```
import Data.Char(ord,chr)

data Bit = O | I deriving (Show, Eq)
type Bitstream = [Bit]

class Encode t where
   encode :: t -> Bitstream
```

```
class Decode t where
    decode :: Bitstream -> t
    decode v = case decode' v of (res, []) -> res
                                 _ -> error "Decode failure."

    decode' :: Bitstream -> (t, Bitstream)

-- Booleans
instance Encode Bool where
    encode True = [I]
    encode False = [O]

instance Decode Bool where
    decode' (I:as) = (True, as)
    decode' (O:as) = (False, as)


-- Lists
instance Encode a => Encode [a] where
    encode [] = [O]
    encode (a:as) = I:(encode a ++ encode as)

instance Decode a => Decode [a] where
    decode' (O:as) = ([], as)
    decode' (I:as) = let (hd,as') = decode' as in
                     let (tl,as'') = decode' as' in
                       (hd:tl, as'')

-- BTrees
data BTree a = Leaf | Branch a (BTree a) (BTree a)
  deriving (Show, Eq)

instance Encode a => Encode (BTree a) where
    encode Leaf = [O]
    encode (Branch a l r) = (I:encode a ++ encode l ++ encode r)

instance Decode a => Decode (BTree a) where
    decode' (O:as) = (Leaf, as)
    decode' (I:as) = let (a,as') = decode' as in
                     let (l,as'') = decode' as' in
                     let (r,as''') = decode' as'' in
                       (Branch a l r, as''')


-- Digits -- to help out for ints
data Digit = Digit Int deriving Show -- assumption that the integer is in 0 -- 9

instance Encode Digit where -- Huffman coding
    encode (Digit 0) = [I,I,I]
    encode (Digit 1) = [O,I,O]
    encode (Digit 2) = [O,O,O]
    encode (Digit 3) = [I,I,O,I]
    encode (Digit 4) = [I,O,I,O]
    encode (Digit 5) = [I,O,O,O]
    encode (Digit 6) = [O,I,I,I]
    encode (Digit 7) = [O,O,I,O]
```

```
    encode (Digit 8) = [I,O,I,I]
    encode (Digit 9) = [O,I,I,O]

instance Decode Digit where
    decode' (I:I:I:as) = (Digit 0, as)
    decode' (O:I:O:as) = (Digit 1, as)
    decode' (O:O:O:as) = (Digit 2, as)
    decode' (I:I:O:I:as) = (Digit 3, as)
    decode' (I:O:I:O:as) = (Digit 4, as)
    decode' (I:O:O:O:as) = (Digit 5, as)
    decode' (O:I:I:I:as) = (Digit 6, as)
    decode' (O:O:I:O:as) = (Digit 7, as)
    decode' (I:O:I:I:as) = (Digit 8, as)
    decode' (O:I:I:O:as) = (Digit 9, as)


-- helper functions
-- numToDigits n
-- PRE: n >= 0
numToDigits n = reverse $ numToDigits' n
    where
        numToDigits' 0 = []
        numToDigits' n = (Digit $ n `mod` 10) : numToDigits' (n `div` 10)

digitsToNum d = digitsToNum' 0 d
    where
        digitsToNum' acc [] = acc
        digitsToNum' acc ((Digit d): as) = digitsToNum' (acc * 10 + d) as


-- Ints
instance Encode Int where
     encode i | i < 0 = O:(encode $ numToDigits (-i))
              | otherwise = I:(encode $ numToDigits i)

instance Decode Int where
    decode' (s:ss) = let (a, as) = decode' ss
                     in (sign * digitsToNum a, as)
                     where sign = case s of O -> -1
                                            I -> 1


-- Char
instance Encode Char where
    encode c = encode (ord c)

instance Decode Char where
    decode' l = let (a, ls) = decode' l in
                (chr a, ls)

--

sample1 = [True, False, True, True]
sample2 = Branch True (Branch False Leaf Leaf) Leaf
sample3 = 24324324 :: Int
sample4 = "The quick brown dog jumped over the lazy fox."
```

```
test1 = decode $ encode sample1 :: [Bool]
test2 = decode $ encode sample2 :: BTree Bool
test3 = decode $ encode sample3 :: Int
test4 = decode $ encode sample4 :: String
```

Think about the following, in any particular order. There's a lot of stuff. I had fun preparing this part of the lab, so maybe you will too:

- The encoding of integers is odd. It was made up based some Huffman code I saw. Why couldn't I just encode them as their corresponding bitstrings? Try it using div and mod.
- Implementing decode' is a bit of a pain, because you have to careful deal with the part of the bitstream that is no used. Is it possible to implement decode directly? Or have some helper function to avoid having to deal with the end of the bitstream explicitly?
- What can happen if your encoding isn't sufficiently unique, meaning that decode cannot work out how to decode a bitstream? How can I ensure that the encoding is decodable?
- Why do I need to put the type annotation ':: T' in after doing a call to decode?
- Write type classes to support the following data types:
  - Integer
  - Tuples
  - data Suit = Club | Diamond | Heart | Spade
  - data Value = Two | Three | Four | Five | Six | Seven
    | Eight | Nine | Ten | Jack | Queen
    | King | Ace
  - type Card = (Suit, Value)
- You use QuickCheck to test whether your encode and decode function work properly.