G53IDS - Interim Report

Embedded Domain Specific Language for
Describing Recipes in Haskell

James Burton - 4251529 - psyjb6

December 7, 2017

Contents

1 Introduction

Consider the following recipe to make a cup of tea:

```
- Boil some water
- Pour over a teabag
- Wait for 3 minutes
- Remove the teabag
- Add milk (optional)
```

This is a very simple but useful recipe that many people will perform, in some cases, many times a day over their lives. What we can realise by looking at this recipe is that it actually consists of many smaller recipes, such as boiling water and combining tea with milk, performed in a certain order. This raises the question, to what extent does the order matter and to what extent can we rearrange things in order to make the recipe more efficient? No doubt you have done this, maybe subconsciously, while cooking at home. Furthermore which steps can be done concurrently in the event that multiple people are cooking e.g. in a professional kitchen with a full brigade?

Perhaps closer to computer science, we could also ask, how could we instruct a robot to do this? After doing some research on robotic chefs it appears that not a huge number exist. There is one home cooking robot [1] which uses motion capture in order to learn recipes. In my opinion this is rather restrictive. It presumes that the human performs the recipe in the optimal manner and it would be very difficult to model a brigade system in this way. In reality there is a limited set of fundamental actions that one becomes able to perform when learning to cook. Recipes can then be performed using a sequence of these actions. Representing recipes like this would allow us to take a robot programmed to perform each of the fundamental actions and tell it how to cook literally anything. This is the same principle as taking code written in a high level language and compiling it down into a sequence of low level actions in assembly language.

So we've established that if we can structure recipes in a more formal way then we will have a great amount of freedom in terms of how we process them whether it be optimisation for a human chef or full on automation. My contributions to this, some not completed as of this point, are the following:

- Define a set of combinators as an EDSL in Haskell and show that they can be used to describe a wide variety of recipes (Section 2).

- The combinators describe a recipe but we then need to know how to execute a recipe as a sequence of the fundamental actions mentioned above. Fortunately, as described above, recipes are just a combination of simpler recipes meaning that we can recursively define the actions necessary to perform complex recipes as long as we have manually defined which action are required for each combinator (Section 3).
- We now have a way to describe recipes and a way to show the sequence of actions to complete a recipe but now we need to apply them to something. Using the operational semantics of recipes we can optimise and schedule recipes for a given kitchen system. We can then express this in several ways including printing steps, drawing the cooking process as a graph or simulating the recipe within the given kitchen setup (Section 4).

2 Describing Recipes

In this section I shall outline the EDSL for describing recipes. The EDSL is implemented in the functional programming language called Haskell which has been used for many EDSLs in the past [2]. Michael Snoyman, creator of Yesod (a Haskell Web Framework), stated many advantages of using Haskell for EDSLs among which were that the type system helps catch mistakes and Haskell allows us to overload almost any syntax [3].

2.1 A Cup of Tea

Consider our cup of tea example from earlier. We can start by defining all of our ingredients:

```
milk, teabag, water :: Recipe
milk = Ingredient "milk"
teabag = Ingredient "teabag"
water = Ingredient "water"
```

The next step is to start describing what we want to transform those ingredients into for example:

```
boilingWater, blackTea :: Recipe
boilingWater = heat (Deg 100) water
blackTea = (teabag >< boilingWater) >>> Wait 5
```

We've introduced three things here. `heat` is simply a function meaning heat the given recipe to the given temperature. `><` is our combine operator and simply means mix the given recipes together. Finally `>>>` is our

sequencing operator meaning do the first recipe then the second. The types for these are as follows:

```
heat :: Temperature -> Recipe -> Recipe
(><) :: Recipe -> Recipe -> Recipe
(>>>) :: Recipe -> Recipe -> Recipe
```

We can now use the above to define our cup of tea recipe as follows:

```
cupOfTea :: Recipe
cupOfTea = blackTea >< milk
```

Now you may notice that we haven't mentioned preparation of ingredients, for example measuring how much of them to use. While experimenting with the basic representation of a recipe and the different ways to interpret it we thought it would be beneficial to keep the combinators as simple and abstract as possible. This means that, for now, we will be describing recipes "cooking show" style where we presume everything is conveniently measured and prepared in a bowl next to the chef.

2.2 Currying - But Not What You Think

In this section I shall introduce the full set of combinators we are currently working with and proceed to build up a more complex recipe.

`ingredient :: String -> Recipe`
The recipe `(ingredient s)` simply represents an ingredient with the name s.

`heat :: Temperature -> Recipe -> Recipe`
`heat t r` means to heat the recipe r to the temperature t.

`wait :: Time -> Recipe`
`wait t` simply means do nothing for t amount of time. The decision to for `wait` to not include a recipe as an argument was very deliberate and will be discussed later in this subsection.

`(><) :: Recipe -> Recipe -> Recipe`
The recipe `r1 >< r2` is the combination of r1 and r2. The details of the method you use to combine the recipes are not yet captured.

`(>>>) :: Recipe -> Recipe -> Recipe`
`r1 >>> r2` represents the sequence of r1 and r2 i.e. perform recipe r1 followed by recipe r2.

Figure 1: Combinators for defining recipes

For the sake of conciseness I shalln't bother listing all the ingredient declarations here as they are relatively straightforward.

We could just create a complex recipe outright however, this would be rather verbose and is likely to repeat a lot of code which somewhat goes against the core principles of DSLs and Haskell. Therefore we shall begin

by defining some extra combinators made from our fundamental combinators.

Marinating is a frequently used recipe so let's write a function for it:

```
marinate :: Recipe -> [Recipe] -> Time -> Recipe
marinate r' [] t     = r' >>> wait t
marinate r' [r] t    = (r' >< r) >>> wait t
marinate r' (r:rs) t = marinate r' [foldr (><) r rs] t
```

`marinate r' rs t` means marinate `r'` in a marinade composed of the list of recipes `rs` for an amount of time `t`.

Due to the compositional nature of our combinators it is easy to create new ones like `marinate` thus allowing us to avoid repeating common sequences of recipes. There is no need to adjust any of our functions that work on recipes to work with `marinate` because it's just a combination of the combinators we already support.

A huge number of recipes start by heating olive oil in a pan then adding something. We define the following combinator for this:

```
preheatOil :: Temperature -> Recipe -> Recipe
preheatOil t r = oil >< r
    where oil = heat t oliveOil
```

That is `preheatOil Medium onion` means heat some olive oil on medium heat then add some onion.

I'm now going to explain the reasoning behind `wait` not taking a recipe as an argument. What this allows us to do is use `wait` in multiple ways. `wait t >< r` means perform recipe r for t amount of time whereas `r >>> wait t` means perform recipe r then wait for t. Using this we can define another combinator to allow us to heat a recipe at a given temperature but also for a certain time:

```
heatFor :: Temperature -> Recipe -> Time -> Recipe
heatFor temp r time = (heat temp r) >< (wait time)
```

So let's get to that more complex recipe I mentioned earlier, Chicken Jalfrezi!

```
spicedChicken :: Recipe
spicedChicken = marinate chicken
    [cumin, coriander, turmeric]

chickenAndPeppers :: Recipe
```

```
chickenAndPeppers = heatFor Medium
    (cookedChicken >< redPeppers) 10
    where cookedChicken = heatFor Medium
                          spicedChicken 10

jalfreziSauce :: Recipe
jalfreziSauce = heatFor Medium
                (sauceBase >< spices) 10
    where
        onionMix     = onion >< garlic
                     >< greenChilli
        cookedOnions = heatFor Medium
                       (preheatOil Medium onionMix) 5
        spices       = cumin >< coriander
                       >< turmeric >< garamMasala
        sauceBase    = cookedOnions >< water
                       >< tinnedTomatoes >< spices

chickenJalfrezi :: Recipe
chickenJalfrezi = heatFor Medium chickenJalfrezi' 5
    where chickenJalfrezi' = chickenAndPeppers
                             >< jalfreziSauce
                             >< cherryTomatoes
```

And there we have it. Quite a large recipe expressed formally with multiple reusable components. As the project continues, the set of combinators will continue to evolve as we run in to issues. Some of these issues have already been discovered and will be discussed later (Section 3.3).

2.3 Conditionals

Many recipes have some sort of condition attached to them which dictate whether or not they are completed. This could be reaching a certain temperature or a certain amount of time elapsing. This functionality is not currently captured by our combinators so we need to introduce the concept of a conditional recipe. The current plan is to implement another constructor:

```
forall a. Eq a => Recipe `Until` a
```

That is for any type that supports equality, allow us to use it as a condition for our recipe so `r `Until` c` would mean "perform recipe r until conditon c evaluates to true".

2.4 Moving Forward

At the time of writing this interim report, the combinators we have are decent but not perfect. There are several crucial pieces of information not captured such as measurements and, as mentioned above, conditionals and optional recipes are not yet implemented.

As the project continues we will naturally be provided with an opportunity to scrutinise the set of combinators we are working with by analysing any issues we run into.

3 Meaning of a Recipe

Our set of combinators allow us to describe a recipe, but what does a recipe actually mean? What are the different ways it can be interpreted? In this section we will discuss these possible interpretations.

3.1 Ingredient Analysis

The end result of all recipes is just the result of transforming and combining the initial ingredients in some way. Extracting a list of ingredients from one of our recipes would be really simple as `Ingredient` is it's own constructor in our recipe type. Once we have this list we can process it in a variety of ways.

One such interpretation could be to evaluate the cost of the recipe and create a shopping list of sorts. Even the cost of a recipe has multiple interpretations. For example the recipe may only use 50g of flour however, if we don't already have some flour then we would have to buy an entire bag so we may wish to use that cost instead.

If we had access to some sort of database with details about ingredients we could work out the nutritional values of a recipe, which diets it supports and the range of flavours present in the dish. This could then possibly be used to recommend susbstitutes to meet certain requirements or, going back to our shopping list idea, in the event that an item is out of stock.

3.2 Executing a Recipe

How do we actually perform a recipe? We need to translate our declarative recipe into a set of actions. Again if we can make these compositional then we will only need to change our definition in the event of a new fundamental combinator.

If we consider our description of a recipe as a tree then the actions we can choose to complete first are the leaves of the tree. This means that we have a set of actions that we can possibly perform at any given time and this set of actions becomes smaller as we move to the root of the tree i.e. as time progresses.

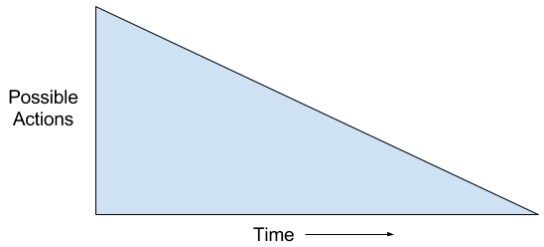

Figure 2: Number of possible actions over time.

We can model this relationship using denotational semantics by translating our description of a recipe into a recipe process. A recipe process is a mapping of time to a recipe action `RP = Time -> RA`. A recipe action is a list of the possible actions you could be doing, `RA = [Action]` where actions are fundamental things such as getting an ingredient or stirring something. The tree structure of our recipes makes denotational semantics possible, due to their compositional nature, which also means we can make use of `fold` in Haskell to evaluate our recipes or to create a stack of steps [8].

$$
\begin{aligned}
[\![Ingredient\ s]\!] &= Node\ (Get\ (Ingredient\ s))\ [] \\
[\![Heat\ t\ r]\!] &= Node\ (PlaceInHeat\ r)\ [Node\ (Preheat\ t)\ [], [\![r]\!]] \\
[\![Wait\ t]\!] &= Node\ (DoNothing\ t)\ [] \\
[\![Combine\ r1\ r2]\!] &= Node\ Mix\ [[\![r1]\!], [\![r2]\!]] \\
[\![Sequence\ r1\ r2]\!] &= Node\ (DoNothing\ 0)\ [[\![r1]\!], [\![r2]\!]]
\end{aligned}
$$

Figure 3: Denotational semantics for recipes.

Now you may notice that the definition `Sequence` is a bit strange. The `DoNothing 0` action is just a bit of a hack so that parts of the project discussed below could be started. The issue leading to the need for this hack is discussed in Section 3.3.

The current approach to this is to first translate the recipe into a tree of actions. The tree data type is as follows:

```
data Tree a = Empty | Node a [Tree a]
```

Then we find all the possible ways the tree could be topologically sorted which gives us all the possible orderings of the actions that maintain the natural ordering of the tree, thus not changing the recipe. The type at this point would be `[[Action]]`, transposing this list of lists would then allow us to define our recipe process as `xs !! t` where `xs` is the transposed list of topological sorts and `t` is the current time. Note that this is a simplified notion of time in that `t == 0` would mean start the first action, `t == 1` the second action and so on.

3.3 Some Issues

One issue with our tree type above is that the root node will naturally contain no action, except in the unlikely event that there is only one possible first action. Rather than adjusting the data type, the solution to this could be to add some sort of placeholder `Init` action. This is not only a solution to the problem but could also prove very useful in the event that we are dealing with robots. Depending on the model we are given (Section 4) it may be necessary to do some sort of initialisation before beginning the actual actions for the recipe.

Secondly, while writing the denotational semantics it became apparent that there were some issues with the fundamental combinators. Consider the following recipe `(r1 >< r2) >>> (r3 >< r4)`, that is "combine r1 and r2, then combine r3 and r4". This would translate into the following tree of actions:

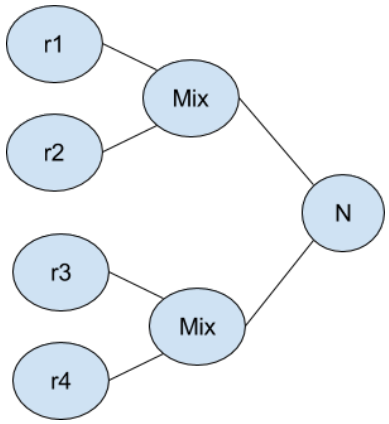

Figure 4: Action tree for the above recipe.

The issue is, what should the action at node N be? The cause of this issue is that we have added meaning to a recipe that doesn't exist in the form of the sequencing operator. This contradicts the statement that "a DSL should capture the semantics of the domain and nothing more" [2]. Sequencing of steps only matters in a recipe if the steps interact with each other. This is automatically captured by the tree like nature of our combinators. What has happened in this recipe is we have tried to enforce sequencing on two unrelated recipes.

One could argue that this is an issue that should be ruled out by a type checker. As SPJ said "we can't expect abstract syntax to rule out all bad programs, that's what type checkers are for" [7]. On the other hand I would argue that what is being talked about here is the prevention of programs that make no sense for example subtracting an integer from a string. In our case the recipe should make sense and as mentioned there is evidence of a more fundamental issue with the DSL design therefore it is appropriate to directly address and fix this issue.

It may seem as though we can just remove the sequencing combinator to fix this issue however, sequencing or combining our wait combinator with

a recipe is our way of determining the difference between "do r then wait" and "do r for x amount of time". This highlights another issue which is that our combine combinator has too many purposes. In the presentation that accompanied the financial contracts language it was stated that combinators should have one responsibility [6]. Our solution is to provide this functionality with our concept of conditional recipes (Section 2.3).

4 Concrete Implementation

If we want to have an effective method of scheduling and optimising these actions then it would be useful to know more about the environment in which they are being performed. We will do this by providing a model in the form of a kitchen data type that can be used to store relevant information. Overall the structure of the project is now looking something like the following:

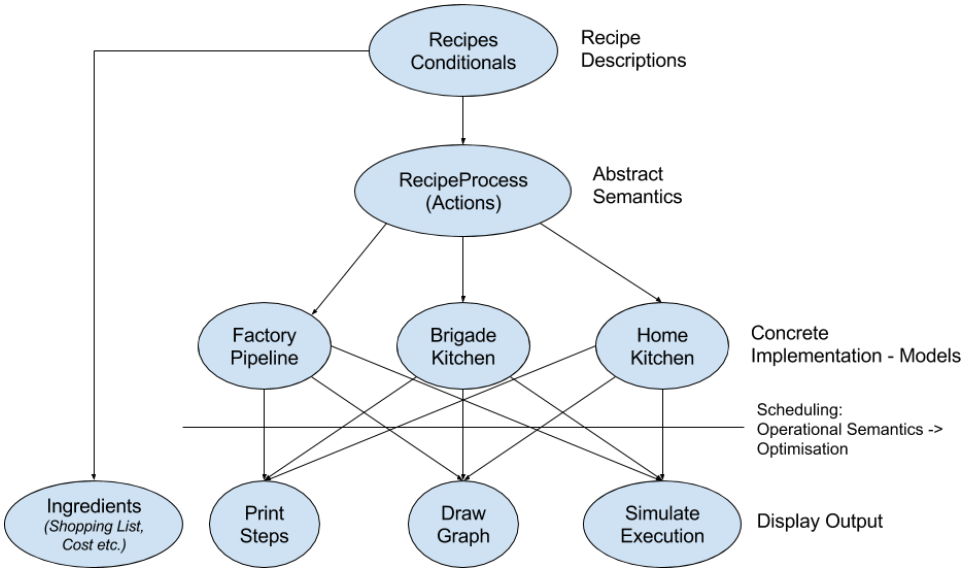

Figure 5: Components of the recipe system.

4.1 Kitchens

In a home kitchen we only have a small set of "paths of execution", which we'll refer to as stations, "stations" for example one oven, one human etc.

Each of these stations is only capable of performing certain actions. The oven can only heat something whereas the human has to put that thing into the oven. In a brigade system we would have many stations each responsible for various things. The kitchen data type will also need to contain functions to call in order to get any relevant information to be used in our conditionals such as the current temperature of an oven or the current time. At this point in the project this data type has not been designed except for the basic concept as just detailed.

4.2 Optimisations, Graphs and Simulations

So we've described a recipe in a declarative sense, compiled it into a set of actions and modelled the environment in which we're cooking. Now we can start looking at how to process and display this information in a useful way to the user. The first would be to optimise the recipe and schedule it. By this we mean trying to perform as many tasks concurrently as possible. We could also run through the process of this schedule either in the form of a simulation or actually controlling some automated system. It may also be sufficient to just print out the schedule either as a list of steps for each station or displayed as a tree or graph of some sort.

5 Progress

This section focuses on the schedule of the project and the progress made so far.

5.1 Project Management

The original project plan that was submitted with the project proposal is included below:

1. Create project proposal and email to supervisor (deadline 13th October).
2. Submit completed ethics clearance form to supervisor (deadline 23rd October).
3. Amend project proposal based on feedback from supervisor and submit amended version (deadline 23rd October).
4. Read [2, 4, 5] to gain a better understanding of the DSL design process.

5. Boil down, if you'll forgive the pun, a few recipes into their components and recreate the recipes in terms of a set of combinators. Try to create other more complex recipes and repeat until a satisfactory set of combinators is found.
6. Define algebraic properties of the combinators.
7. Consider various representations of recipes and the combinators e.g. as functions or as data structures.
8. Write and submit interim report (deadline 8th December).
9. Mainly revise for other modules over Christmas and during exams.
10. Once the core components of the DSL are in place consider specifics such as temperature being centigrade, farenheit or gas mark.
11. Create system to output recipes from the DSL into a format similar to that of a regular recipe.
12. Create parser to read recipe formatted text into the DSL.
13. Experiment with various applications of the DSL such as finding steps that can be executed concurrently.
14. Outline contents for final report.
15. Write and submit final report (deadline 24th April).

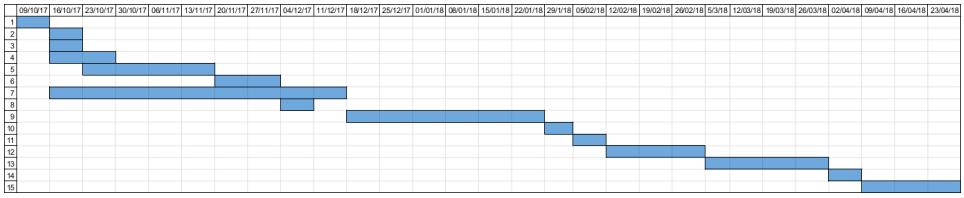

Figure 6: Project plan gantt chart from the project proposal.

There has been some adjustment to the original project schedule. Initially I thought that I would spend the first half of the project designing the combinators from Section 2.2 however, my supervisor suggested it would be better to quickly define an initial set of combinators then use them to design and develop the rest of the project. This was in order to prevent being stuck refining the combinators forever and not having anything else

to show. Refining the combinators while developing other sections is also useful because issues can often be brought to light that would otherwise have gone unnoticed.

Below is an updated plan for the remainder of the project:

1. Continue to refine and adjust the set of combinators and set of actions as other areas of the project are developed.
2. Finish implementation of denotational semantics and recipe processes.
3. Catch up on other coursework.
4. Mainly revise for other modules over Christmas and during exams.
5. Create kitchen data type for concrete implementation.
6. Create systems to print recipes in a variety of ways.
7. Define operational semantics to see if any optimisations are possible then work on the scheduling of a recipe.
8. Once the core components of the DSL are in place consider specifics such as units of temperature and allowing the measuring of ingredients and recipes.
9. Outline contents for final report.
10. Write and submit final report (deadline 24th April).

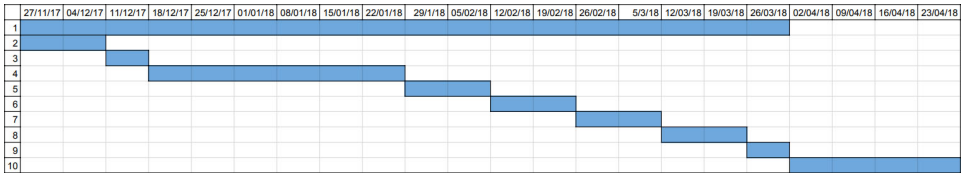

Figure 7: Updated project plan gantt chart.

5.2 Contributions and Reflections

Currently the project is progressing well. While certain areas such as the concrete implementation have not been developed or fully designed yet, there is at least a firm idea of the direction the project will go over the remainder of its duration. The scrum development methodology seems to be effective in allowing flexibility. For example quite a bit of time was spent thinking of how to represent the semantics of a recipe and I am now in the process of implementing that system.

As far as I am aware there is no exsiting EDSL for describing recipes or manipulating recipes. While DSLs and various forms of semantics are certainly not new, exactly how to apply them to recipes is relatively unexplored. The contributions made so far are as follows:

- A small set of combinators that can be used to describe a wide variety of recipes.
- A partial implementation of recipe processes which detail how to actually perform a recipe.
- A plan to model kitchens as a set of stations in order to schedule or simulate a recipe in a certain environment.

These will be further developed and expanded on over the remainder of the project.

6 Related Work

As stated in the previous section, a recipe DSL is something that has not been done before. However, the various componenets such as the concept of a DSL have been around for a long time. A DSL that has some similarities to the recipe DSL is that for financial contracts [4]. Reading that paper provided a lot of inspiration and guidance regarding how to break down recipes and proceed with the design of the language. There has also been research done into DSLs in general for example [2] and [3].

References

[1] The Guardian. 2015. *Future of food: how we cook*. https://www.theguardian.com/technology/2015/sep/13/future-of-food-how-we-cook

[2] Paul Hudak. Domain Specific Languages. Department of Computer Science, Yale University, December 15, 1997.

[3] Michael Snoynman. O'Reilly Webcast: Designing Domain Specific Languages with Haskell. January 4, 2013. https://www.youtube.com/watch?v=8k_SU1t50M8

[4] Simon Peyton Jones, Microsoft Research, Cambridge. Jean-Marc Eber, LexiFi Technologies, Paris. Julian Seward, University of Glasgow. Composing contracts: an adventure in financial engineering. August 17, 2000.

[5] John Hughes. The Design of a Pretty-printing Library. Chalmers Teniska Hogskola, Goteborg, Sweden. 1995.

[6] Simon Peyton Jones, Microsoft Research, Cambridge. Jean-Marc Eber, LexiFi Technologies, Paris. Julian Seward, University of Glasgow. Composing contracts: an adventure in financial engineering (PowerPoint Slides). August 17, 2000. https://www.microsoft.com/en-us/research/publication/composing-contracts-an-adventure-in-financial-engineering/

[7] Simon Peyton Jones. Into the Core - Squeezing Haskell into Nine Constructors. September 14, 2016. https://www.youtube.com/watch?v=uR_VzYxvbxg

[8] Graham Hutton. Fold and Unfold for Program Semantics. Department of Computer Science, University of Nottingham. September 1998.

A Recipe.Recipe

```
module Recipe.Recipe where

import Data.List
import Recipe.Tree

-------------------------------------
-- RECIPE DEFINITION
-------------------------------------

data Recipe = Ingredient String
            | Heat Temperature Recipe
            | Wait Int
            | Combine Recipe Recipe
            | Sequence Recipe Recipe
            -- | forall a. Eq a => Recipe `Until` a
            deriving Show

type Quantity = Int
type Time = Int

data Temperature = Deg Int | Low | Medium | High
    deriving (Show, Eq)

-- measure :: Quantity -> Recipe -> Recipe
-- measure = Measure

heat :: Temperature -> Recipe -> Recipe
heat = Heat

(><) :: Recipe -> Recipe -> Recipe
(><) = Combine

-- r1 then r2
(>>>) :: Recipe -> Recipe -> Recipe
(>>>) = Sequence

wait :: Quantity -> Recipe
wait = Wait

-------------------------------------
-- RECIPE SEMANTICS
-------------------------------------
```

```
filterJust :: [Maybe a] -> [a]
filterJust xs = [x | (Just x) <- xs]

sortRecipe :: Recipe -> [Action]
sortRecipe r = filterJust mas
    where
        t = labelTree $ expand r
        ls = kahn t
        mas = map (\l -> findLabel l t) ls

-- Translate Recipe into a tree of actions
expand :: Recipe -> Tree Action
expand r@(Ingredient s)   = Node (Get r) []
expand r@(Heat t r')      =
    Node (PlaceInHeat r') [Node (Preheat t) [], expand r']
expand r@(Wait t)         = Node (DoNothing t) []
expand r@(Combine r1 r2)  =
    Node (Mix r1 r2) [expand r1, expand r2]
expand r@(Sequence r1 r2) =
    Node (DoNothing 0) [expand r1, expand r2]

type RP = Time -> RA
type RA = [Action]

data Action = Get Recipe
    -- Heat
    | Preheat Temperature
    | Refrigerate Recipe
    | PlaceInHeat Recipe
    | LeaveRoomTemp Recipe
    | Freeze Recipe
    -- Wait
    | DoNothing Time
    -- Combine
    | PlaceAbove Recipe Recipe
    | PlaceIn Recipe Recipe
    | PourOver Recipe Recipe
    | Mix Recipe Recipe
    --
    | Init
    deriving Show

-------------------------------------
-- CONCRETE IMPLEMENTATION
```

```
-------------------------------------

-- TODO

-------------------------------------
-- UTILITY FUNCTIONS
-------------------------------------

-- Create a list of ingredients used in a recipe
getIngredients :: Recipe -> [String]
getIngredients (Ingredient s)   = [s]
getIngredients (Heat _ r)       = getIngredients r
getIngredients (Combine r1 r2)  =
    getIngredients r1 ++ getIngredients r2
getIngredients (Wait _)         = []
getIngredients (Sequence r1 r2) =
    getIngredients r1 ++ getIngredients r2
```

B Recipe.Tree

```
module Recipe.Tree where

import Control.Monad.State

data Tree a = Empty | Node a [Tree a]
    deriving Show

testT :: Tree Int
testT = Node 1 [Node 2 [Node 3 []], Node 4 []]

instance Functor Tree where
    -- fmap :: (a -> b) -> fa -> fb
    fmap _ Empty       = Empty
    fmap f (Node a ts) = Node (f a) (map (fmap f) ts)

type Label = Int

-- Label tree from 0 breadth-first
labelTree :: Tree a -> Tree (Label, a)
labelTree Empty = Empty
labelTree t = evalState (labelTree' t) 0

labelTree' :: Tree a -> State Label (Tree (Label, a))
labelTree' Empty       = return Empty
labelTree' (Node a ts) = do
```

```
    l <- get
    put (l + 1)
    ts' <- mapM labelTree' ts
    return $ Node (l, a) ts'

-- Returns children of a node
children :: Tree a -> [Tree a]
children Empty       = []
children (Node _ ts) = ts

-- True if node has no children
leaf :: Tree a -> Bool
leaf (Node _ (t:ts)) = False
leaf _               = True

-- Calculate the degree of a node in the tree
calcDegree :: Tree a -> Int
calcDegree Empty       = 0
calcDegree (Node _ ts) = length ts

-- Produce a list of nodes with a degree of zero
zeroDegree :: Tree a -> [Tree a]
zeroDegree n =
    if calcDegree n == 0
        then (n:ns)
        else ns
    where ns = concatMap zeroDegree (children n)

-- Removes any node with the given label from the tree
-- Children of that node are also removed
removeNode :: Label -> Tree (Label, a) -> Tree (Label, a)
removeNode _ Empty       = Empty
removeNode l (Node a ts) = Node a [t | t <- ts', getLabel t /= l]
    where ts' = map (removeNode l) ts

-- Get the parent node of the node with the
-- given label in the given tree, Empty if no parent
getParent :: Label -> Tree (Label, a) -> Tree (Label, a)
getParent _ Empty       = Empty
getParent _ (Node _ []) = Empty
getParent l n@(Node (l', _) ts)
    | l == l'   = Empty
    | otherwise = if True `elem` (map (\t -> getLabel t == l) ts)
                  then n
                  else head $ map (getParent l) ts
```

```
-- Get label of a given node
getLabel :: Tree (Label, a) -> Label
getLabel Empty           = -1
getLabel (Node (l, _) _) = l

-- Producs a list of Labels of nodes sorted topologically
kahn :: Tree (Label, a) -> [Label]
kahn t = kahn' t [] (map getLabel $ zeroDegree t)

-- ns = labels of sorted nodes
-- zs = labels of zero degree nodes
-- Presumes values stored in nodes are unique
kahn' :: Tree (Label, a) -> [Label] -> [Label] -> [Label]
kahn' _ ns []      = ns
kahn' t ns (z:zs)  = kahn' t' ns' zs'
    where
        -- take a zero degree node z and add to tail of sorted nodes
        ns' = ns ++ [z]
        -- get parent of z
        parentZ = getParent z t
        pLabel = getLabel parentZ
        -- remove z
        t' = removeNode z t
        -- if no other edges to parent then insert into sorted nodes
        zOnlyChild = length (children parentZ) <= 1
        pzNotEmpty = pLabel /= -1
        zs' = if zOnlyChild && pzNotEmpty
                then zs ++ [pLabel]
                else zs

-- Given a label and a tree, gives the first value found
-- with the corresponding label, if not found then Nothing
findLabel :: Label -> Tree (Label, a) -> Maybe a
findLabel _ Empty             = Nothing
findLabel l (Node (l', a) ts) =
    if l == l'
        then Just a
        else case [x | x@(Just _) <- xs] of
            []     -> Nothing
            (x:xs) -> x
    where xs = map (findLabel l) ts
```

C Recipe.Printer

```
module Recipe.Printer where

import           Recipe.Recipe

-- data Recipe = Ingredient String
--             | Heat Temperature Recipe
--             | Combine Recipe Recipe
--             | Wait Time
--             | Sequence Recipe Recipe
--             deriving (Show)

-------------------------------------
-- PRINTING RECIPES
-------------------------------------

toString :: Recipe -> String
toString (Ingredient s)   = s
toString (Heat t r)       =
    "Heat (" ++ toString r ++ ") to " ++ show t
toString (Combine r1 r2)  =
    "Mix (" ++ toString r1 ++ ") with (" ++ toString r2 ++ ")"
toString (Wait t)         = "Wait for " ++ show t
toString (Sequence r1 r2) =
    "Do (" ++ toString r1 ++ ") then (" ++ toString r2 ++ ")"

printRecipe :: Recipe -> IO ()
printRecipe x@(Ingredient _)   = putStrLn $ toString x
printRecipe x@(Heat _ r)       = printRecipe r
                                    >> putStrLn (toString x)
printRecipe x@(Combine r1 r2)  = printRecipe r1
                                    >> printRecipe r2
                                    >> putStrLn (toString x)
printRecipe x@(Wait _)         = putStrLn $ toString x
printRecipe x@(Sequence r1 r2) = printRecipe r1
                                    >> putStrLn (toString x)
                                    >> printRecipe r2

-------------------------------------
-- PRINTING INGREDIENTS
-------------------------------------

-- Print the list of ingredients in a recipe
printIngredients :: Recipe -> IO ()
printIngredients r = mapM_ putStrLn (getIngredients r)
```

```
-------------------------------------
-- PRICE
-------------------------------------

type Price = Float
type PricedItem = (String, Price)
type PriceList = [PricedItem]

class Priced a where
    findCost :: a -> PriceList -> Price

instance Priced Recipe where
    findCost (Ingredient s) ps =
        case prices of
            [] -> 0
            _  -> head prices
        where prices = [p | (x, p) <- ps, x == s]
    findCost (Heat _ r) ps       = findCost r ps
    findCost (Combine r1 r2) ps  =
        findCost r1 ps + findCost r2 ps
    findCost (Wait _) _          = 0
    findCost (Sequence r1 r2) ps =
        findCost r1 ps + findCost r2 ps

testList :: PriceList
testList = [ ("milk", 1.00)
           , ("teabag", 6.70)
           ]

-- Need to amend this once we have the measurement constructor
-- We don't need a full pack of teabags to make a cup of tea
```

D Recipe.Demo

```
module Recipe.Demo where

import Recipe.Recipe
import Recipe.Printer

-------------------------------------
-- TEST RECIPES
-------------------------------------

-- Cup of Tea
```

```
milk, teabag, water :: Recipe
milk = Ingredient "milk"
teabag = Ingredient "teabag"
water = Ingredient "water"

boilingWater, blackTea :: Recipe
boilingWater = heat (Deg 100) water
blackTea = (teabag >< boilingWater) >>> Wait 5

cupOfTea :: Recipe
cupOfTea = blackTea >< milk

-- Chicken Jalfrezi

oliveOil, chicken, cumin, coriander, turmeric :: Recipe
redPeppers, garamMasala, tinnedTomatoes, onion :: Recipe
garlic, greenChilli, cherryTomatoes :: Recipe
chicken = Ingredient "chicken"
cumin = Ingredient "cumin"
coriander = Ingredient "coriander"
turmeric = Ingredient "turmeric"
redPeppers = Ingredient "red peppers"
garamMasala = Ingredient "garam masala"
tinnedTomatoes = Ingredient "tinned tomatoes"
onion = Ingredient "onion"
garlic = Ingredient "garlic"
greenChilli = Ingredient "green chilli"
cherryTomatoes = Ingredient "cherry tomatoes"
oliveOil = Ingredient "olive oil"

spicedChicken :: Recipe
spicedChicken = marinate chicken
    [cumin, coriander, turmeric] 30

chickenAndPeppers :: Recipe
chickenAndPeppers = heatFor Medium
    (cookedChicken >< redPeppers) 10
    where cookedChicken = heatFor Medium
                            spicedChicken 10

jalfreziSauce :: Recipe
jalfreziSauce = heatFor Medium
                (sauceBase >< spices) 10
    where
        onionMix     = onion >< garlic >< greenChilli
```

```
        cookedOnions = heatFor Medium
                         (preheatOil Medium onionMix) 5
        spices       = cumin >< coriander
                         >< turmeric >< garamMasala
        sauceBase    = cookedOnions >< water
                         >< tinnedTomatoes >< spices

chickenJalfrezi :: Recipe
chickenJalfrezi = heatFor Medium chickenJalfrezi' 5
    where chickenJalfrezi' = chickenAndPeppers
                               >< jalfreziSauce
                               >< cherryTomatoes

-------------------------------------
-- CUSTOM COMBINATORS
-------------------------------------

marinate :: Recipe -> [Recipe] -> Time -> Recipe
marinate r' [] t     = r' >>> wait t
marinate r' [r] t    = (r' >< r) >>> wait t
marinate r' (r:rs) t = marinate r' [foldr (><) r rs] t

preheatOil :: Temperature -> Recipe -> Recipe
preheatOil t r = oil >< r
    where oil = heat t oliveOil

heatFor :: Temperature -> Recipe -> Time -> Recipe
heatFor temp r time = (heat temp r) >< (wait time)
```