Program Design

Stuart Paton

February 1, 2013

1 Introduction

In this document I will discuss my design and modelling considerations for my project's implementation by thorough discussion of each task chosen.

2 Internals

To construct accurate models of each Catalan structure I have, in the file "Internal.hs", created a type class called Catalan which is defined as follows:

```
class Catalan a where
        cons :: a -> a -> a
        decons :: a -> (a,a)
```

In this type class, I have two operators: *cons* and *decons* which are the operators to construct, or compose and deconstruct (or decompose) respectively for each structure.

3 Dyck Paths

To model Dyck paths, I created a module which I named **CatalanStructures.DyckPath**. To represent a Dyck Path I have a list of up-steps and down-steps. Each step is represented by the algebraic data type Step which uses the encoding U for an up-step and D for a down-step. A full Dyck path is represented as a list of steps shown below in the type synonym DyckPath.

```
data Step = U | D deriving (Eq, Show)
type DyckPath = [Step]
```

Next I created an instance of Catalan for the type DyckPath. It is constructed as follows:

```
instance Catalan DyckPath where
        cons alpha beta = mkIndec alpha ++ beta
        decons gamma = stripMaybe $ decompose gamma
```

The function *mkIndec* takes a Dyck path, alpha, and makes an indecomposable DyckPath by prepending a U to the start of alpha then a D to the end of alpha. Then to fully compose our Dyck path with a given alpha and beta we just append beta to mkIndec alpha as is shown above.
Decomposing a Dyck path is the hardest task faced in the design and implementation of the model of a Dyck path. To do this we make a function called *decompose* with takes in a parameter gamma, where gamma is a full Dyck path. Our function decompose looks like the following:

```
decompose :: DyckPath -> Maybe (DyckPath, DyckPath)
decompose [] = Nothing
decompose xs@(U:xt) = Just (map fst (init ys), map fst zs)
                where
                  0:ht = height xs
                  (ys, zs) = span(\(_, h) -> h > 0) $ zip xt ht
```

This function starts off by taking a Dyck path and mapping each element of the Dyck path to the height of each element in the half, except from the first element which is disregarded. This is shown by 0:ht. In order to obtain (ys, zs) we use the span function which splits the list into our alpha and beta lists disgregarding the down-step which is appended to alpha. To finish off we apply the following to *Just*. We map the first element of the pair to all the elements of ys except the first, and we then map the first element of the pair to zs.
For the height function, it is defined as follows:

```
height :: DyckPath -> [Int]
height = scanl (+) 0 . map dy
        where
        dy U = 1
        dy D = -1
```

As this is in $O(n)$ time instead of the next example which is in $O(n^2)$ time it is more efficient as it repeatedly adds the partal sums starting from 0 to each element which was mapped to their dy values. The next example is the $O(n^2)$ version.

```
height :: DyckPath -> [Int]
height = map sum . inits .map dy
          where
            dy U = 1
            dy D = -1
```

Here we start by mapping our encodings of U and D to create a list of 1's and -1's. By applying this to the function inits, it creates a list of partial sums which we then fully add together using "map sum" and we have the height of each element.