2017 Haskell January Test
Decision Trees

This test comprises three parts and the maximum mark is 30. The **2017 Haskell Programming Prize** will be awarded for the best overall solution.

Credit will be awarded throughout for clarity, conciseness, *useful* commenting and the appropriate use of Haskell's various language features and predefined functions.

WARNING: The examples and test cases here are not guaranteed to exercise all aspects of your code. You should therefore define your own tests to complement the ones provided.

1 Introduction

A *decision tree* is a data structure for classifying data that is organised as a collection of *records*. Each record specifies the values of a given set of *attributes*, together with a *classification* for that record. A path from the root of a decision tree to each leaf encodes a sequence of attribute tests that, when applied to an arbitrary record in the data set, correctly identifies the class to which that record belongs. Once a tree has been built from so-called "training" data, the idea is to use the tree to classify previously-unseen data. The tree thus provides a mechanism for "learning" rules from data that can then be usefully applied in other contexts. Decision trees are extensively used in decision analysis and machine learning.

1.1 Example

To illustrate the idea, Table 1 shows a data set that describes the weather conditions on each of 14 fly-fishing days, together with a classification of what the fishing was like on each of those days[1]. The data is presented in the form of a table with four "input" attributes, `outlook`, `temp` (temperature), `humidity`, `wind`, and one "output" attribute, `result`, which serves to classify the data. The `result` attribute has two possible values, `bad` and `good`, and the values associated with the other attributes are:

Attribute	Values
`outlook`	`sunny`, `overcast`, `rainy`
`temp`	`hot`, `mild`, `cool`
`humidity`	`high`, `normal`
`wind`	`windy`, `calm`

Figures 1 and 2 show two possible decision trees for the data in Table 1. Each internal node (rectangle) denotes a test on an attribute and the arcs emanating from the node are labelled with the possible outcomes of the test, i.e. the possible values of the attribute. The leaves of the tree denote the classification (`good` or `bad`). A path through the tree thus represents a series of attribute tests so, for example, the combination `outlook=rainy`, `temp=mild`, `humidity=high`, `wind=windy` in Figure 1 leads to the classification `bad`, which reflects the information in the last row of Table 1.

The special `null` leaf is used when a path represents a set of attribute values that does not appear in the corresponding data set. For example, the combination `outlook=rainy`, `temp=hot` does not appear anywhere in Table 1, hence the `null` at the end of the corresponding path.

Notice that the tree in Figure 2 is very much simpler than that of Figure 1 yet is able to classify the same data correctly without reference to the `temp` attribute. Notice also that in both trees the test `outlook=overcast` leads immediately to the classification `good` without having to test any other attributes. We'll see later that the size and structure of the tree depends on the order in which the attributes are selected when building it.

[1]This is the classic and often-cited data set described in the seminal paper by Ross Quinlan (*J. R. Quinlan. Induction of decision trees. Mach. Learn., 1(1):81106, March 1986.*), but adapted here for a much more exciting application!

outlook	temp	humidity	wind	result
sunny	hot	high	calm	bad
sunny	hot	high	windy	bad
overcast	hot	high	calm	good
rainy	mild	high	calm	good
rainy	cool	normal	calm	good
rainy	cool	normal	windy	bad
overcast	cool	normal	windy	good
sunny	mild	high	calm	bad
sunny	cool	normal	calm	good
rainy	mild	normal	calm	good
sunny	mild	normal	windy	good
overcast	mild	high	windy	good
overcast	hot	normal	calm	good
rainy	mild	high	windy	bad

Table 1: Sample data set and classification

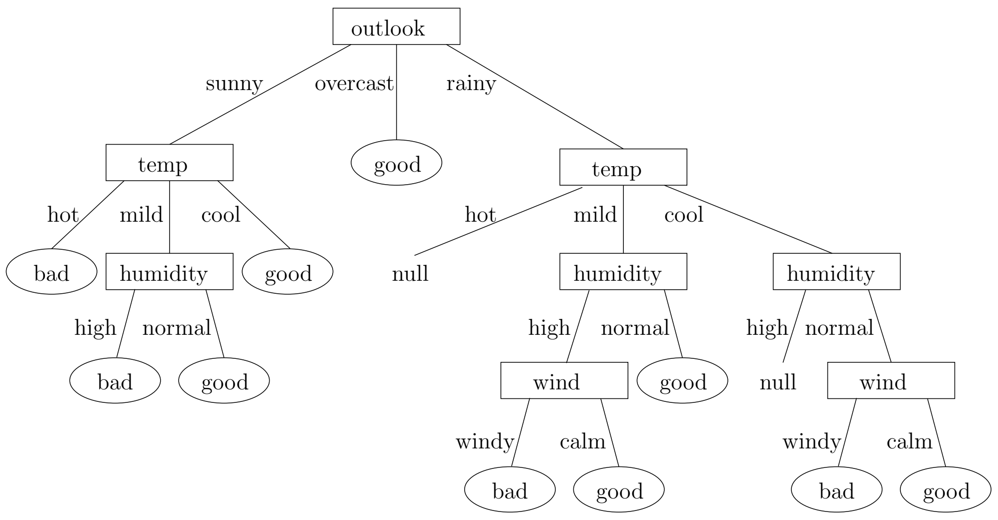

Figure 1: Complex decision tree for Table 1

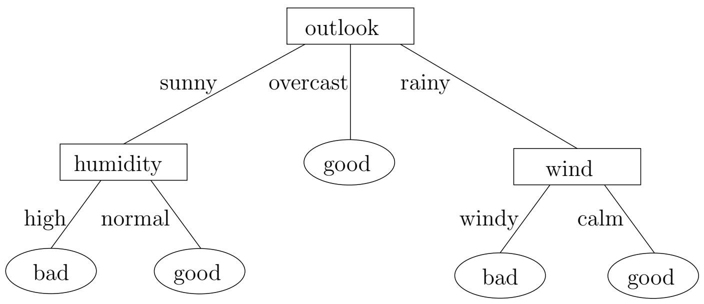

Figure 2: Optimised decision tree for Table 1

2 Representation

Data sets and decision trees can be represented in Haskell using the following type synonyms and data types, where we assume that all attribute names and values are `String`s:

```
type AttName = String

type AttValue = String

type Attribute = (AttName, [AttValue])

type Header = [Attribute]

type Row = [AttValue]

type DataSet = (Header, [Row])

data DecisionTree = Null |
                    Leaf AttValue |
                    Node AttName [(AttValue, DecisionTree)]
                  deriving (Eq, Show)
```

An `Attribute` comprises its name (equivalent to a `String`) and the possible values it can take (equivalent to a `[String]`). A data set (`DataSet`) consists of a header (`Header`) and a data *table* comprising a list of rows (`[Row]`). Each row contains the values of both the input attributes (the data "record" in the above sense) and its classification. The header specifies the attributes referenced in each row of the data set, in order. A universal assumption is that the values in each row are consistent with the corresponding attribute in the header. To illustrate this, the representation of the data set of Table 1 is shown below:

```
fishingData :: DataSet
fishingData
  = (header, table)

header :: Header
table  :: [Row]
header
  =  [outlook,    temp,   humidity, wind,    result]
table
  = [["sunny",    "hot",  "high",   "calm",  "bad" ],
     ["sunny",    "hot",  "high",   "windy", "bad" ],
     ["overcast", "hot",  "high",   "calm",  "good"],
     ["rainy",    "mild", "high",   "calm",  "good"],
     ["rainy",    "cool", "normal", "calm",  "good"],
     ["rainy",    "cool", "normal", "windy", "bad" ],
     ["overcast", "cool", "normal", "windy", "good"],
     ["sunny",    "mild", "high",   "calm",  "bad" ],
     ["sunny",    "cool", "normal", "calm",  "good"],
     ["rainy",    "mild", "normal", "calm",  "good"],
     ["sunny",    "mild", "normal", "windy", "good"],
     ["overcast", "mild", "high",   "windy", "good"],
     ["overcast", "hot",  "normal", "calm",  "good"],
     ["rainy",    "mild", "high",   "windy", "bad" ]]
```

Each row in the table has exactly five attribute values: four representing the data record, and one representing the classification. We'll see later that, in general, the classification 'column' can appear anywhere in a table, although in this example it happens to be the last column. Note that the header comprises a list of `Attributes`, rather than a list of `Strings`, because it is important to know the set of values that are allowed to appear in each column of the table. As an example, the attribute `temp` is declared in the template as follows:

```
temp :: Attribute
temp
  = ("temp", ["hot", "mild", "cool"])
```

The template includes the above definitions and also those of the remaining attributes (`outlook`, `humidity`, `wind` and `result`).

At this point you are in a position to answer the questions in Part I. You might wish to complete these before reading on.

3 Evaluating a tree

A decision tree (`DecisionTree`) can be *evaluated* with respect to the information in a given row of data (`Row`) by using the attribute values in that row to traverse the tree from its root to a leaf. As an example, consider the Haskell representation of the first row of Table 1:

```
*Main> table !! 0
["sunny","hot","high","calm","bad"]
```

These values can be used to trace a path through a corresponding decision tree that leads to a leaf labelled `bad`. It is important to appreciate the following:

- In order to know which row value(s) to use it is essential to know the corresponding header, because the internal nodes of the tree refer to attribute *names* which need to mapped to attribute *values*; the attribute values are used to determine which branch to take at each node in the tree. In the example here the header is:

  ```
  [outlook, temp, humidity, wind, result]
  ```

  which tells us that `"sunny"` is a value of the `outlook` attribute, `"hot"` is a value of `temp`, `"high"` is a value of `humidity` and `"calm"` is a value of `wind`.

- The order in which we need to look up the attribute values may, in general, be different to the order in which the attributes appear in the header. For example, in Figure 2 the left branch requires the attributes to be tested in the order `outlook` followed by `humidity`, which is different to the attribute order in the corresponding header (`outlook` followed by `temp`).

- The classifying attribute (`result` in the example) and its associated values aren't needed to evaluate a tree, but we can safely leave them in both the header and row data. This is because there will never be an internal node in a tree labelled with that attribute.

At this point you are in a position to answer the questions in Part II. You might wish to complete these before reading on.

4 Building a tree

A decision tree for a data set is built by *partitioning* the data based on a particular attribute and then recursively building a subtree for each partition. For example, suppose we choose to partition the data in Table 1 using the `outlook` attribute. This splits the table into three sub-tables, one for each possible value of `outlook`, as shown in Table 2. Notice that each partition represents a valid data set (`DataSet`), but with the `outlook` attribute removed from both the sub-table and its header.

In the Haskell representation, a `Partition` comprises the value of the attribute used to partition the data (`sunny`, `overcast` and `rainy` respectively for the three partitions shown in Table 2), together with the `DataSet` for that partition. The following type synonym is defined in the template:

```
type Partition = [(AttValue, DataSet)]
```

The `outlook` attribute forms the root node (`Node`) of the resulting decision tree and the `DataSets` in the three partitions are used to build three *child* subtrees of that root node recursively: one for the attribute value `sunny`, one for `overcast` and one for `rainy`. Note that in the Haskell representation each child is represented by a pair comprising the value of the attribute (`AttValue`) and the corresponding sub-tree (`DecisionTree`). This is mirrored in Figures 1 and 2, where the arcs are labelled with the attribute value associated with each sub-tree.

The above recursive building process terminates when either every row in the given data set has the same value for the classification attribute or the data set is empty, i.e. its corresponding table (type `[Row]`) is `[]`. For example, when we build the child sub-tree corresponding to the `overcast` sub-table in Table 2 (second sub-table) we find that each row has the same classification, `good`. In this case we return `Leaf "good"`, as reflected in Figures 1 and 2. If the data set is empty, we return `Null`.

4.1 Choosing the next attribute for partitioning

The structure of the tree generated by the recursive algorithm above is determined by the choice of attribute used at each stage to perform the partitioning. For example, Figure 1 uses the attributes in the order that they appear in the header, i.e. first `outlook`, then `temp` then `humidity` then `wind`. On the other hand, Figure 2 chooses the next attribute on the basis of *information gain*, as will be detailed in Section 5.4. The two tree building variants differ *only* in the choice of next attribute, so when you come to code your tree building function

temp	humidity	wind	result
hot	high	calm	bad
hot	high	windy	bad
mild	high	calm	bad
cool	normal	calm	good
mild	normal	windy	good

temp	humidity	wind	result
hot	high	calm	good
cool	normal	windy	good
mild	high	windy	good
hot	normal	calm	good

temp	humidity	wind	result
mild	high	calm	good
cool	normal	calm	good
cool	normal	windy	bad
mild	normal	calm	good
mild	high	windy	bad

Table 2: Partitions for the `sunny`, `overcast` and `rainy` values of `outlook`

the attribute selection function will be a *parameter* of that function, i.e. the building function will be higher-order.

At this point you are in a position to answer the questions in Part III.

5 What to do

There are four parts to this exercise and all but three of the marks are assigned to the first three parts. You should only attempt Part IV when you have completed Parts I–III.

The fishing data set (`fishingData`) is included in the template along with the representations of the trees in Figures 1 and 2 (`fig1` and `fig2` respectively) for testing purposes.

A function `lookUp :: (Eq a, Show a, Show b) => a -> [(a, b)] -> b` is defined in the template that will look up the value of an item of type `a` in a list of `(a, b)` pairs. If there is no binding for the item in the list the function displays a useful error message that may help you if you need to debug your code.

Universal preconditions

There are two universal preconditions in what follows:

- There is a header (`Header`) associated with every data row (`Row`), and thus every table (`[Row]`), and the attribute values in each row are consistent with the corresponding header.
- All data records are unique, so it is not possible for the same record to have two conflicting classifications.

5.1 Part I: Utilities

1. Define a function `allSame :: Eq a => [a] -> Bool` that will return `True` if every item in a given list is the same; `False` otherwise. For example,

   ```
   *Main> allSame []
   True
   *Main> allSame [9,9,9]
   True
   *Main> allSame "abc"
   False
   ```

   [2 Marks]

2. Define a function `remove :: Eq a => a -> [(a, b)] -> [(a, b)]` that will remove an item of type `a` from a table of `(a, b)` pairs. If the item isn't found the table should be returned unmodified. For example,

   ```
   *Main> remove 1 [(3,'a'),(1,'b'),(7,'a')]
   [(3,'a'),(7,'a')]
   *Main> remove 6 []
   []
   ```

   [2 Marks]

3. Using the predefined `lookUp` function, define a function `lookUpAtt :: AttName -> Header -> Row -> AttValue` that will look up the value of a given attribute in a given data row. Note that in order to do this you need the header information which is also a parameter of the function. A precondition is that the attribute name is present in the given header. For example,

   ```
   *Main> table !! 0
   ["sunny","hot","high","calm","bad"]
   *Main> lookUpAtt "temp" header (table !! 0)
   "hot"
   ```

   Note that `header` and `table` are the header and rows of the fishing data set (`fishingData`) and are defined in the template.
   [2 Marks]

4. Define a function `removeAtt :: AttName -> Header -> Row -> Row` that will remove the value of a named attribute from a given data row. The order of the elements in the result must be the same as in the original row. Again, you need the corresponding header to do this. For example,

   ```
   *Main> table !! 0
   ["sunny","hot","high","calm","bad"]
   *Main> removeAtt "temp" header (table !! 0)
   ["sunny","high","calm","bad"]
   ```

   [2 Marks]

5. A *mapping* (sometimes just called a *map*, but not to be confused with Haskell's `map` function) is a table that associates an item with a *list* of values. As an example, each `Attribute` in this exercise is a specific type of mapping from `AttNames` to lists of `AttValues`. Define a polymorphic function `addToMapping :: Eq a => (a, b) -> [(a, [b])] -> [(a, [b])]` that will add a new `(x, v)` pair to a given mapping. If there is already a binding for `x` then `v` should be added to the *front* of the existing list of values associated with `x`. A precondition is that there will be at most one binding for `x` in the table. If there is no existing binding for `x` then then the new binding `(x, [v])` should be added to the mapping. For example,

   ```
   *Main> addToMapping (1,'a') [(2,"b")]
   [(2,"b"),(1,"a")]
   *Main> addToMapping (5,'a') [(2,"b"),(5,"bcd")]
   [(2,"b"),(5,"abcd")]
   ```

   The order of the elements in the mapping returned is unimportant.

   [3 Marks]

6. Define a function `buildFrequencyTable :: Attribute -> DataSet -> [(AttValue, Int)]` that will build a frequency table that counts the number of occurrences of each value of a given attribute in a given data set. For example,

   ```
   *Main> buildFrequencyTable result fishingData
   [("good",9),("bad",5)]
   *Main> buildFrequencyTable outlook fishingData
   [("sunny",5),("overcast",4),("rainy",5)]
   *Main> buildFrequencyTable outlook ([],[])
   [("sunny",0),("overcast",0),("rainy",0)]
   ```

   The order of the elements in the resulting table is unimportant.

   [3 Marks]

5.2 Part II: Functions on trees

1. Define a function `nodes :: DecisionTree -> Int` that will count the total number of nodes and leaves in a given decision tree. A `Null` tree is defined to have a count of zero. For example,

   ```
   *Main> nodes fig1
   18
   *Main> nodes fig2
   8
   ```

   [2 Marks]

2. Using the `lookUp` and `lookUpAtt` functions, define a function `evalTree :: DecisionTree -> Header -> Row -> AttValue` that will evaluate a given tree using the attribute values in a given data row. Again, in order to do this you need the header corresponding to the given data, as described in Section 3. If the tree is `Null` the function should return `""`. For example,

   ```
   *Main> evalTree fig1 header (table !! 5)
   "bad"
   *Main> evalTree fig2 header (table !! 4)
   "good"
   ```

   [3 Marks]

5.3 Part III: Building a tree

This part requires you to write two functions that, used together, will build a decision tree from a given data set. In this part of the problem we'll assume that a tree is always built by selecting the *first* attribute in the data set's header, unless this happens to be the classifier attribute in which case we pick the next one along. This was how the tree of Figure 1 was built. A function for selecting attributes in this way is predefined in the template thus:

```
type AttSelector = DataSet -> Attribute -> Attribute

nextAtt :: AttSelector
nextAtt (header, table) (classifierName, _)
  = head (filter ((/= classifierName) . fst) header)
```

The idea is for the tree building function to take an attribute selector function (`AttSelector`) as a *parameter*. We can then change the behaviour of the building function simply by passing in different selector functions. Note that the type synonym representing the attribute selection function appears at the top of the template along with the other type synonyms.

1. Using the utility functions in questions 2–5 of Part I, or otherwise, define a function `partitionData :: DataSet -> Attribute -> Partition` that will partition a data set using the specified attribute, as described in Section 4. A constant `outlookPartition :: Partition` representing the result of partitioning the fishing data set using the `outlook` attribute is defined in the template for testing purposes. For example,

   ```
   *Main> partitionData fishingData outlook == outlookPartition
   True
   *Main> let ps = partitionData fishingData outlook
   *Main> let (val, (header', table')) = ps !! 1
   *Main> header'
   [("temp",["hot","mild","cool"]),("humidity",["high","normal"]),
   ("wind",["windy","calm"]),("result",["good","bad"])]
   *Main> table'
   [["hot","high","calm","good"],["cool","normal","windy","good"],
   ["mild","high","windy","good"],["hot","normal","calm","good"]]
   ```

[5 Marks]

2. Define a tree-building function `buildTree :: DataSet -> Attribute -> AttSelector -> DecisionTree` that will use the given attribute selector function to build a decision tree from a given data set, as described in Section 4. The `Attribute` parameter denotes the classification attribute; in the fishing data set this happens to be the last 'column', but in general it doesn't have to be. The `AttSelector` is the *function* that selects the next attribute, given a `DataSet` and classifier `Attribute`, as described above. As a example, given the fishing data set and the simple `nextAtt` selector function defined above, `buildTree` should generate the tree shown in Figure 1:

```
*Main> buildTree fishingData result nextAtt == fig1
True
```

[4 Marks]

5.4 Part IV: Partitioning using information gain

The `nextAtt` function above always chooses the first attribute in the current header as the basis of the partition. However, this typically leads to trees that are far from optimal. A smarter way is to choose the attribute that maximises the *information gain*; this was how the tree in Figure 2 was built.

The idea is to construct a probability distribution for the classification attribute, which defines how each of its values is apportioned in the classification 'column' of a given data set. For example, in the fishing data set the classification attribute `result` has just two possible values: `good` and `bad`. There are 9 occurrences of `good` and 5 of `bad` out of a total of 14 rows in the original data set, F say, so the probability distribution is defined by the two probabilities $Prob(F, \texttt{result}, \texttt{good}) = \frac{9}{14}$ and $Prob(F, \texttt{result}, \texttt{bad}) = \frac{5}{14}$. It is easy to see how this generalises to more than two values.

Given an arbitrary data set D, the *entropy* of a probability distribution for an attribute A that has n_A associated values, $V_{A,i}, 1 \leq i \leq n_A$, is defined to be:

$$E(D, A) = \sum_{i=1}^{n_A} -p_i \times \log_2 p_i$$

where $p_i = Prob(D, A, V_{A,i})$ and

$$Prob(D, A, x) = \frac{\text{no. of occurrences of value } x \text{ for attribute } A \text{ in } D}{\text{no. of rows in } D}$$

For example, for the above fishing data (F, say), we get:

$$\begin{aligned} E(F, \texttt{result}) &= -Prob(F, \texttt{result}, \texttt{good}) \times \log_2(Prob(F, \texttt{result}, \texttt{good})) \\ &\quad -Prob(F, \texttt{result}, \texttt{bad}) \times \log_2(Prob(F, \texttt{result}, \texttt{bad})) \\ &= -\frac{9}{14} \times \log_2\left(\frac{9}{14}\right) - \frac{5}{14} \times \log_2\left(\frac{5}{14}\right) \\ &= 0.940 \end{aligned}$$

to three decimal places (d.p.)

The *gain* for a particular attribute is a measure of the reduction in entropy that would arise by picking that attribute for partitioning. Given a data set D, classification attribute C and partitioning attribute P, the gain is:

$$G(D, P, C) = E(D, C) - \sum_{i=1}^{n_P} Prob(D, P, i) \times E(D[P, V_{P,i}], C)$$

where $D[P, x]$ is the partition of D associated with value x of attribute P.

As an example, the three sub-tables in Table 2 correspond to $F[\texttt{outlook}, \texttt{sunny}]$, $F[\texttt{outlook}, \texttt{overcast}]$ and $F[\texttt{outlook},\texttt{rainy}]$ respectively. The associated entropies for these using `result` as the classification attribute (i.e. $C = \texttt{result}$ above) are respectively:

$$\begin{aligned}
E(F[\texttt{outlook,sunny}], \texttt{result}) &= -\frac{2}{5} \times \log_2\left(\frac{2}{5}\right) - \frac{3}{5} \times \log_2\left(\frac{3}{5}\right) = 0.971 \\
E(F[\texttt{outlook,overcast}], \texttt{result}) &= -\frac{4}{4} \times \log_2\left(\frac{4}{4}\right) - \frac{0}{4} \times \log_2\left(\frac{0}{4}\right) = 0.0 \\
E(F[\texttt{outlook,rainy}], \texttt{result}) &= -\frac{3}{5} \times \log_2\left(\frac{3}{5}\right) - \frac{2}{5} \times \log_2\left(\frac{2}{5}\right) = 0.971
\end{aligned}$$

where $0 \times \log_2(0)$ is taken to be 0. The other probabilities are given by:

$$\begin{aligned}
Prob(F, \texttt{outlook,sunny}) &= \frac{5}{14} \\
Prob(F, \texttt{outlook,overcast}) &= \frac{4}{14} \\
Prob(F, \texttt{outlook,rainy}) &= \frac{5}{14}
\end{aligned}$$

Thus, the gain for `outlook` is:

$$\begin{aligned}
G(F, \texttt{outlook}, \texttt{result}) &= 0.940 - \left(\frac{5}{14} \times 0.971 + \frac{4}{14} \times 0.0 + \frac{5}{14} \times 0.971\right) \\
&= 0.246
\end{aligned}$$

to three d.p. (this actually comes out to 0.247 to three d.p. if you don't round the intermediate calculations).

In order to use this to build an optimised decision tree we simply pick the attribute that has the largest information gain.

1. A function `xlogx` has been defined in the template to compute $x \times \log_2 x$ for a given x. Using this, define a function `entropy :: DataSet -> Attribute -> Double` that computes the entropy of a given data set with respect to a given attribute, as described above. The `buildFrequencyTable` function from Part I should help you here. The entropy of an empty data set is defined to be 0.0. For example,

```
*Main> entropy fishingData result
0.9402859586706309
*Main> entropy fishingData temp
1.5566567074628228
*Main> entropy (header, []) result
0.0
```

2. Define a function `gain :: DataSet -> Attribute -> Attribute -> Double` that computes the information gain of a given data set with respect to a given partitioning attribute (first `Attribute` argument) and classification attribute (second `Attribute` argument), as described above. For example,

   ```
   *Main> gain fishingData outlook result
   0.2467498197744391
   *Main> gain fishingData temp result
   2.9222565658954647e-2
   *Main> gain fishingData humidity result
   0.15183550136234136
   *Main> gain fishingData wind result
   4.812703040826927e-2
   ```

   Note that the maximum gain is for `outlook`, so you would pick this one next to partition on.

3. Define an attribute selection function `bestGainAtt :: AttSelector` that selects the attribute that has the largest information gain, as described above.

   Important: when deciding which attribute to select using gain you must first remove the classification attribute (`result` in the fishing example) from the header. This is because there is a chance that this attribute could yield the maximum gain and we don't want to partition the data based on it. Its job is solely to classify.

   Thus, for example:

   ```
   *Main> bestGainAtt fishingData result
   ("outlook",["sunny","overcast","rainy"])
   *Main> buildTree fishingData result bestGainAtt
   Node "outlook" [("sunny",Node "humidity" [("high",Leaf "bad"),
   ("normal",Leaf "good")]),("overcast",Leaf "good"),
   ("rainy",Node "wind" [("windy",Leaf "bad"),
   ("calm",Leaf "good")])]
   *Main> buildTree fishingData result bestGainAtt == fig2
   True
   ```

2 Marks total for Part IV