Unit 5 - Advanced (hierarchical) data structures

In this unit, we discuss the definition and management of operations around hierarchical data structures. In preivous courses, these data structures were discussed in their mutable version. In this course we explore their immutable implementation, in order to prevent side effects that could happen when updating the references or trying to use `null` values in the mutable implementation. We will start by explaining the implementation of generic trees, and then we will implement binary search trees. We conclude this section by showing how to implement decision trees, that are a popular data structure used in machine learning.

Trees

A tree can be recursively defined as either an empty tree, or a node containing data and a sequence of subtrees (children). This means that its type definition will be both polymorphic and recursive:

```
type Tree<'a> =
| Empty
| Node of 'a * List<Tree<'a>>
```

Like for lists, we can define `map` and `fold` higher-order functions, respectively to mutate the content of each node in a tree, and to accumulate the result of a tree operation into an accumulator.

The `map` function will traverse the tree in some order and apply a function to each node of the tree. We choose to first apply the function to the content of the current node, and then recursively traverse the list of sub-trees and apply `map` to them. The `map` applied to an empty tree of course results in an empty tree. In the case of a non-empty tree, we apply `f` to the current element, and then we call the `map` function **for lists** on **the list of subtrees**, by passing a function that calls the **tree map** on each element of the list. We generate a new node by taking the result of `f` applied to the current node and the result of mapping the list of trees with the tree map.

```
member this.Map (f : 'a -> 'b) =
  match this with
  | Empty -> Empty
  | Node(x,subtrees) ->
      Node(f x,subtrees |> List.map(fun t -> t.Map f))
```

The `fold` works similarly to its counterpart for lists. It takes as input a function that takes as input a state and an element of the tree, and updates the state. Moreover, we pass to `fold` also the initial value of the state. The function updates the state by calling `f` with the current state and

element, thus generating a new state that we call `state1`. It then call `fold` **for lists** passing a function that uses the accumulator and each tree. This function will use `fold` **for trees** to update the accumulator with each subtree. As initial value of the accumulator we pass `state1`, the newly generated state at the current level.

```
member this.Fold (f : 'state -> 'a -> 'state) (state : 'state) : 'state =
  match this with
  | Empty -> state
  | Node(x,subtrees) ->
      let state1 = f state x
      subtrees |>
      List.fold (fun s tree -> tree.Fold f s) state1
```

Binary Search Trees

A binary search tree is a data structure that can be used to implement a dictionary. This means that each node stores an element identified by a unique key and a value. Each node can also has two sub-trees as children. The keys stored in the root of the left sub-tree are all smaller than the key in the current node, and the keys in the right sub-tree are all greater. We start by defining a type to describe the key-value pair stored as element in the node:

```
type Entry<'k,'v> =
  {
    Key : 'k
    Value : 'v
  }
  with
    static member Create(key : 'k, value : 'v) =
      {
        Key = key
        Value = value
      }
```

We can now define the data structure to represent the node of a binary search tree: this data structure will store the actual data, which is an `Entry`, and its left and right sub-tree.

```
type BinaryNode<'k, 'v> when 'k : comparison =
  {
    Entry : Entry<'k, 'v>
    Left : BinarySearchTree<'k, 'v>
    Right : BinarySearchTree<'k, 'v>
  }
  with
    static member
```

```
    Create(
          entry : Entry<'k, 'v>,
          left : BinarySearchTree<'k, 'v>,
          right: BinarySearchTree<'k, 'v>) =
    {
      Entry = entry
      Left = left
      Right = right
    }
```

Note that we must ensure that the key can be used with a comparison operator to check the binary search property, thus we must enforce a type constraint on the generic type `'k`. Finally a binary search tree is polymorphic and defined as either an `Empty` tree or a `Node` containing a `BinaryNode`.

```
and BinarySearchTree<'k, 'v> when 'k : comparison =
| Empty
| Node of BinaryNode<'k, 'v>
```

We now proceed to implement the operations on the dictionary, i.e. `find`, `add`, and `remove`.

Element Lookup

The lookup in a binary search tree searches for a key in the tree and returns the corresponding value in the entry. It also might fail to retrieve the given key, so the return type is `Option<'v>`. The function is defined recursively: if the tree is empty then the lookup fails returning `None`. Otherwise we check if the key in the current node is the one we are looking for. If it is, then we return it encapsulated inside the case `Some` of `Option`. Otherwise if the key we are looking for is smaller then we recursively look in the left sub-tree, otherwise we look to the right. This recursive process will stop as soon as we either find the key we are looking for or we reach an empty sub-tree:

```
member this.TryFind(key : 'k) : Option<'v> =
  match this with
  | Empty -> None
  | Node node ->
    if node.Entry.Key = key then
      Some node.Entry.Value
    elif key < node.Entry.Key then
      node.Left.TryFind key
    else
      node.Right.TryFind key
```

Adding an Element

Adding an element requires finding the proper place to position the new node in the binary search tree in order not to break the binary search property. Note that, being the data structure immutable, `Add` will never modify the existing tree but rather return a new tree that contains also the new entry to be added. The procedure is again recursive: if we are in an empty tree then we just return a new node with an empty left and right sub-tree. If the entry that we want to add is already there (i.e. the key already exists), we replace it. Thus we return a new node with the new entry and having the right and left sub-tree of the old node. Otherwise if the key of the entry to be added is less than the key in the current node, then we recursively call `Add` on the left sub-tree and we return a new node with the same entity and right sub-tree as the current one, but having as left sub-tree the result of the recursive call to `Add`. Otherwise we do the opposite: we recursively call `Add` on the right sub-tree and we create a new node containing the entry and the left sub-tree of the current one, but as right sub-tree the result of `Add`. This recursive process ends as soon as `Add` will be called with an empty sub-tree.

```
match this with
  | Empty -> Node(BinaryNode.Create(entry, Empty, Empty))
  | Node node ->
    if node.Entry.Key = entry.Key then
      Node(BinaryNode.Create(entry, node.Left, node.Right))
    elif entry.Key < node.Entry.Key then
      Node (BinaryNode.Create(node.Entry,node.Left.Add(entry),node.Right))
    else
      Node (BinaryNode.Create(node.Entry,node.Left,node.Right.Add(entry)))
```

Deleting an Element

Deleting an element is more complex and requires to consider three different cases:

1. We delete from an empty tree. This is a base case that has no effect whatsoever. This case may happen when the tree is really empty or when the recursive delete fails to find the entry that we want to delete. In that case the delete has no effect.
2. We delete a node with two empty sub-trees as children.
3. We delete a node where only one of the two sub-trees is non-empty.
4. We delete a node where both sub-trees are non-empty.

Again, keep in mind that this is an immutable representation, so we never modify the current tree but we rather return a copy of it without the element that we are deleting. The function is recurisive and looks at the key stored in the current root. If the key is not matched we use the binary search property to recursively call `Remove` on one of the sub-trees. We than use the result of `Remove` to build a new binary tree that contains a modified version of one of the children without the element that we want to delete. If the key matches the one stored in the root the we proceed with the deletion.

Case 2 requires simply to return an empty tree, since the current level has no sub-trees. Case 3

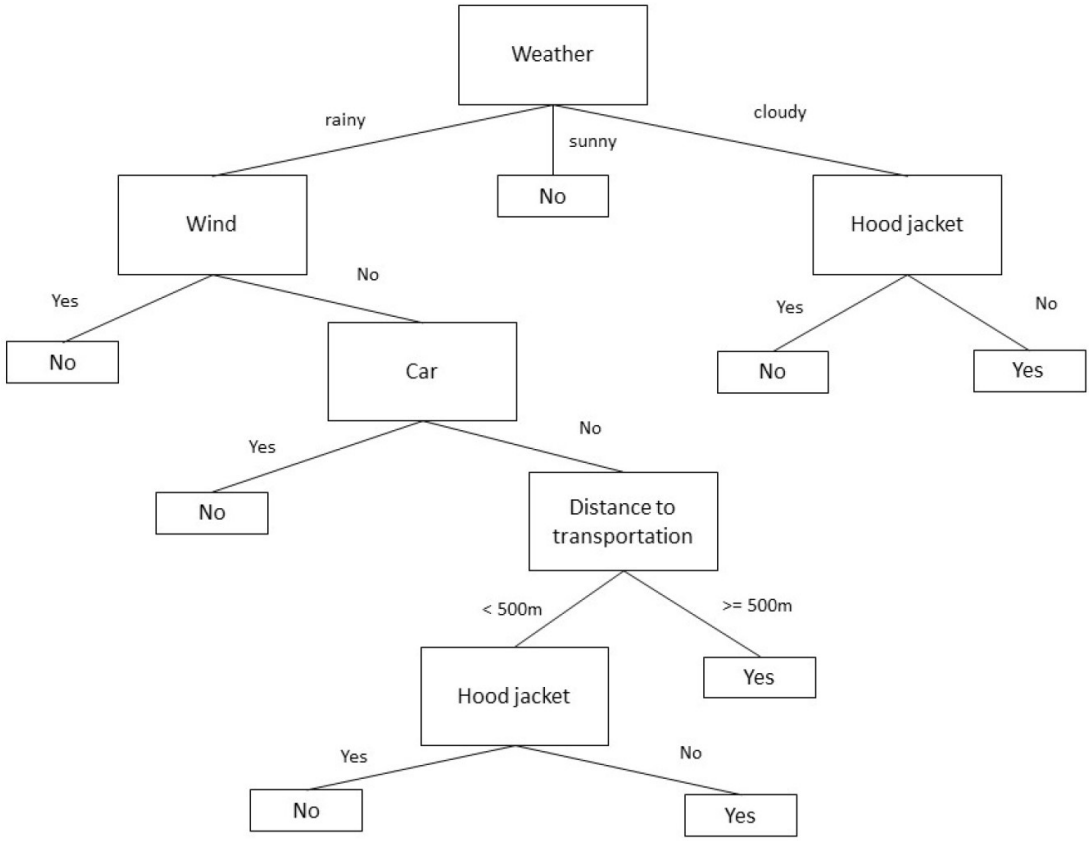
requires simply to return the non-empty child sub-tree. Case 4 requires first finds the leftmost node in the right sub-tree, which we call `rightmostNode`. Equivalently it would also be possible to give an implementation where we find the rightmost node in the left sub-tree. We then recursively call remove on this node, obtaining a new binary tree called `newLeft`. Note that, with such node, we surely fall in one of the previous two cases. We then create a new node at the current having as root `rightmostNode`, as left child `newLeft`, and as right child the same right sub-tree of the current level.

```
member this.Remove(key : 'k) =
  let rec getRightMostElement (tree : BinarySearchTree<'k, 'v>) =
    match tree with
    | Node ({ Entry = _; Left = _; Right = Empty } as node) ->
        node
    | Node ({ Entry = _; Left = _; Right = right }) ->
        getRightMostElement right

  match this with
  | Empty -> Empty
  | Node node ->
    if node.Entry.Key = key then
      match node.Left,node.Right with
      | Empty,Empty -> Empty
      | Node tree,Empty
      | Empty, Node tree -> Node tree
      | Node _, Node _ ->
          let rightMostNode = getRightMostElement node.Left
          let newLeft = node.Left.Remove(rightMostNode.Entry.Key)
          Node (BinaryNode.Create(rightMostNode.Entry, newLeft, node.Right))
    elif key < node.Entry.Key then
      Node (BinaryNode.Create(node.Entry,node.Left.Remove key,node.Right))
    else
      Node (BinaryNode.Create(node.Entry,node.Left,node.Right.Remove key))
```

Decision trees

A hierarchical structure often used in artificial intelligent is decision trees. Let us consider a set of data points made of different features. Each feature has a label (name) and a value that can be either discrete or continuous. Each data point can belong to a different class that describes it. A decision tree is a data structure that, given a data point as input, is able to classify it, i.e. to decide to what class it belongs. In this section we only learn, for simplicity, how to implement the data structure representing a decision tree and how to implement the classification function, but in artificial intelligence decision trees can be generated, so that their structure is not hard-coded but rather learnt from a training set of data points. In the picture below you find a deicision tree that is able to decide for you if you should bring your umbrella depending on the weather condition:



A decision tree containts two kinds of nodes: a node containing a decision to make, and a node containing an outcome, which basically decides to what class assigning a data point. Each decision node contains a series of predicates that are tested in order to decide what path we need to follow to reach a su-tree, that can be itself another decision or a simple outcome. Outcome nodes only specify a belonging class and do not have children.

Let us start by defining a data structure for a data point. This is simply a record containing its label and value:

```
type Feature<'a, 'label> =
  {
    Label : 'label
    Value : 'a
  }
  with
    static member Create(lable, value) =
      {
        Label = lable
        Value = value
      }
```

Note that we use a generic type `'label` for the label itself to ensure type safety when building a decision tree, so that it is not possible to provide an invalid label. A decision node has two

components: one is the lable of the feature that we are going to evaluate with that decision, and the other is a list of pairs, which we call paths, made of a predicate and a decision sub-tree.

```
type Decision<'a, 'label, '_class> when 'label : comparison =
  {
    Label : 'label
    Paths : List<('a -> bool) * DecisionTree<'a, 'label, '_class>>
  }
  with
    static member Create(
      label : 'label,
      paths : List<('a -> bool) * DecisionTree<'a, 'label, '_class>>
    ) =
      {
        Label = label
        Paths = paths
      }
```

Note that we use the type constraint `'label : comparison` because later we need to perform a comparison on the type `'label`. Every time we reach a decision node, we find the feature in the data point matching the label, and then we test its value against the predicate of each path. As soon as the predicate is satisfied, we recursively call the classification algorithm on the corresponding sub-tree. We can now define a node in the decision tree as a polymorphic type that is either an `Outcome` or a `Decision`:

```
and DecisionTree<'a, 'label, '_class> when 'label : comparison =
| Outcome of '_class
| Decision of Decision<'a, 'label, '_class>
```

We can now start implementing the classification method. This method takes as input a data point, which we can model as a `Map` where the key is the name of a feature and the value its corresponding value. The method checks the current root node and behaves according to the following cases:

1. If the node is an `Outcome`, then we simply return the class contained in it.
2. If the node is a `Decision`, then we try to find in the data point a feature with the same label of the decision. If this process fails, then the data point is malformed and we return `None`. If we succeed then we test the predicate contained in each path. If all predicates fail to match then it is not possible to classify the data point according to the decision rules. If one of the predicate evaluates to `true`, then we recursively call `Classify` on the corresponding sub-tree.

```
member this.Classify (features : Map<'label,'a>) =
  match this with
```

```
    | Outcome _class -> Some _class
    | Decision decision ->
        match
          features |>
          Map.tryFind(decision.Label) with
        | Some value ->
            match
              decision.Paths |>
              List.tryFind(fun (condition,_) -> condition value) with
            | Some(_,tree) -> tree.Classify features
            | None -> None
        | None -> None
```

Now it should appear clear why the type constraint on `'label` has been enforced: using the function `tryFind` for `Map` (not for `List`) requires that the generic type `'T` of the list is comparable, because `Map` is implemented as a search tree.

In order to test this, let us use the weather decision tree shown in the picture above. We need to define an extra auxiliary type to describe the possible values that a weather feature can have:

```
type WeatherFeature =
| Sunny
| Rainy
| Cloudy
| Bool of bool
| Float of float
```

and the possible labels:

```
type WeatherLabel =
| Weather
| Wind
| Hood
| Car
| Distance
```

The class can be simply represented by a `boolean` value, since it can only be `yes` or `no`. We can then hard-code the structure of the represented tree, which will have type `DecisionTree<WeatherFeature,WeatherLabel,bool>`:

```
let weatherTree =
  Decision(
    Decision.Create(
      Weather,
      [
```

```
    (fun v -> v = Rainy),Decision(
      Decision.Create(
        Wind,
        [
          (fun v -> v = (Bool true)),Outcome false
          (fun v -> v = (Bool false)),
            Decision(
              Decision.Create(
                Car,
                [
                  (fun v -> v = (Bool true)),Outcome false
                  (fun v -> v = (Bool false)),Decision(
                    Decision.Create(
                      Distance,
                      [
                        (fun v ->
                          match v with
                          | Float x when x >= 500.0 -> true
                          | _ -> false),Outcome true
                        (fun _ -> true),Decision(
                          Decision.Create(
                            Hood,
                            [
                              (fun v -> v = (Bool true)),Outcome false
                              (fun v -> v = (Bool false)),Outcome true
                            ]
                          )
                        )
                      ]
                    ))
                ]
              )
            )
        ]
      )
    )
    (fun v -> v = Sunny),Outcome false
    (fun v -> v = Cloudy),Decision(
      Decision.Create(
        Hood,
        [
          (fun v -> v = (Bool true)),Outcome false
          (fun v -> v = (Bool false)),Outcome true
        ]
      )
    )
  ])
)
```

and test it with the following data point:

```
let weatherData =
  [
    Weather,Rainy
    Wind,Bool false
    Car,Bool false
    Distance,Float 350.5
    Hood, Bool false
  ] |> Map.ofList
```

which will lead to yes as outcome.

Graphs

A Graph is a set of nodes and edges (oriented or not) that connect pairs of nodes. In previous courses you have seen that graphs can be represented as an adjacency matrix or adjacency list. Of course, also in functional programming it would be possible to represent a graph through an adjacency matrix with nested lits of `float` values. In this section we explore a way of representing graphs in functional programming similar to adjacency lists. We start by defining an edge as (notice that we consider an oriented graph):

```
type Edge<'a,'b> =
  {
    Origin        : 'a
    Destination   : 'a
    Weight        : Option<'b>
  }
  with
    static member Create(origin : 'a,destination : 'a,weight : Option<'b>) =
      {
        Origin = origin
        Destination = destination
        Weight = weight
      }
```

This gives a generic representation of an edge, where the nodes can have any time and so does the weight. It leaves also the choice of whether to have a weight for an edge at all or not. A graph can now be defined as a record containing simply a list of edges:

```
type Graph<'a,'b> when 'a : equality =
  {
    Edges     : List<Edge<'a,'b>>
  }
  with
```

```
    static member Create(edges : List<Edge<'a,'b>>) = { Edges = edges }
```

Notice that we are enforcing a type constraint `equality` on `'a` because we will need it for the methods that we will implement below. One of the first things that is missing from this implementation is the nodes stored in the graph. In order to determine the nodes from the edges, we can fold along the list of edges and accumulate the nodes of each edge in a list. Of course this could introduce duplicates, as two adjacent edges will always share a node, so that will be added twice. Before adding a node we thus check whether it is already in the accumulator or not.

```
member this.Nodes =
  this.Edges |>
  List.fold(
    fun nodes edge ->
      let nodes1 =
        if
          nodes |>
          List.contains edge.Origin
        then
          nodes
        else
          edge.Origin :: nodes
      let nodes1 =
        if
          nodes1 |> List.contains edge.Destination
        then
          nodes1
        else
          edge.Destination :: nodes1
      nodes1
  ) []
```

> Notice that `Nodes` is implemented as a .NET property (properties are the same as in C#). Properties in F# records can be defined as methods with the same syntax without parameters. This will define a read-only property.

> We could have used the F# `Set` data type which automatically handles duplicates for us. If you do so, further operations on `Nodes` should use the `Seq` module instead of `List`.

Another useful method, is finding the neighbours given a node. In order to do so, we must find all the edges in the graph that have the input node as origin. After this we can map the list to output only the destinations, which are the nodes adjacent to the input one.

```
member this.Neighbours(node : 'a) : List<'a> =
  this.Edges |>
  List.filter(fun e -> e.Origin = node) |>
```

```
    List.map(fun e -> e.Destination)
```

At this point we are ready to implement some more complex operations on graphs. We can define a `Map` on the graph, which applies a mapping function to each node in the graph and returns a graph with the result of the function application to each node. This simply requires to map each edge, applying the function to both `Origin` and `Destination`, and then re-creating a new edge with the result of the function application to both nodes.

```
  member this.Map (f : 'a -> 'a1) : Graph<'a1,'b>  =
    let mappedEdges =
      this.Edges |>
      List.map(fun e -> Edge.Create(f e.Origin,f e.Destination,e.Weight))
```

Finally we can define `fold` on a graph, which iterates through all the nodes and updates an accumulator after running a function that takes as input both a node and the current accumulator. This only requires to fold the list of nodes with the given function and accumulator. Conveniently, we have already implemented a property that extracts all the nodes in the graph, thus the implementation of `fold` is simply:

```
  member this.Fold (f : 'state -> 'a -> 'state) (init : 'state) =
    this.Nodes |> List.fold f init
```