HW 10 **Prof. Caldwell**
Due: 28 September 2010 **COSC 3015**

1 Modeling Finite Functions as Lists

Now that you know how to write recursive functions on lists, we will model finite functions as lists of pairs.

You can think of a list of pairs of type `[(a,b)]` as a finite function with domain `a` and codomain `b`. If the pair `(x,y)` is in the list (call it `f`) then `f` maps `x` to the value `y`.

We can defined the type of finite functions with a Haskell datatype as follows:

```
data FinFun a b = FF [(a,b)]
```

Applying the constructor `FF` to a list of pairs casts it as an element of the type `FinFun`.

Exercise 1.1. Based on the code provided on the HW web-page you must write the following functions.

```
update :: Eq a => (a,b) -> FinFun a b -> FinFun a b
functional :: Eq a => [(a,b)] -> Bool
domain :: (Eq a, Eq b) => FinFun a b -> [a]
range :: (Eq a, Eq b) => FinFun a b -> [b]
apply :: (Eq a, Eq b) => FinFun a b -> a -> Maybe b
```

1.1 `update :: Eq a => (a,b) -> FinFun a b -> FinFun a b`

This function should be called as `update(i,v) f` where `f` is a finite function with domain `a` and codomain `b`. If `i` is the first element of any pair in `f` return the finite function that is just like `f` except that it maps `i` to `v`. If `i` is not the first element of any pair in `f`, return the finite function the behaves just like `f` but also contains the pair `(i,v)`. You might find your `remove_all` function from the previous homework useful.

1.2 `functional :: Eq a => [(a,b)] -> Bool`

Recall the functionality property; we say the set of pairs f is functional when the following holds:

$$\forall i : a. \forall j, k : b. (f(i) = j) \wedge (f(i) = k) \Rightarrow j = k$$

Implement a predicate that takes a list of pairs and says whether or not it is functional. The expression `(functional m)`, where `m` is a list of pairs, returns `True` iff for every pair `(i,v)`$\in$ `m`, there is no pair `(j,v')` $\in$ `m` with `i=j` and `v<>v'`. You can be more strict and just return `True` when no two pairs in `m` have the same first element (even if they have equal second elements.) I found the function `unique` useful, but there are many ways to do it.

1.3 `domain :: (Eq a, Eq b) => FinFun a b -> [a]`

This function returns a list of values in the domain `a` that the function is actually defined for. Hint - recall that the function `map` applies a function to every element of a list and that `fst` and `snd` project the first and second elements from a pair.

1.4 `range :: (Eq a, Eq b) => FinFun a b -> [b]`

This function returns a list of values in the codomain `b` that is the range of the finite function. See hint above.

1.5 `apply :: (Eq a, Eq b) => FinFun a b -> a -> Maybe b`

`apply f x` models function application. Look up `x` in the finite function `f` and return `Nothing` if `x` $\notin$ `(domain f)` and return `Just y` when the pair `(x,y)` $\in$ `f`.