Advanced Functional Programming
Uppsala University – Autumn 2012
Assignment 3

Anders Hassis Jonatan Jansson

December 20, 2012

1 Dictionaries

Our solution revolves around two data structures `Tree` and `Dictionary`. `Tree` contains *key, value, left, right* and `Dictionary` contains *root, compare* where compare is a function that will defines how comparing should be handled.

The *Tree*-structure is created as a binary tree to contain all key value-pairs and is sorted with the lowest value, according to the *compare*-function in `Dictionary`, to the left.

1.1 create-dictionary

`create-dictionary` is defined as a constructor for our `Dictionary` structure. If no argument is given, the built-in function `compare` will be used as default.

1.2 lookup

`lookup` finds the value in a dictionary corresponding to a given key. This is done using a auxilary function `lookupaux` which finds a value in a node and its subnodes, given a key. We use pattern matching to match results based on the compare function.

1.3 update

`update` creates a new dictionary with the same content as a given dictionary, but with a new key value-pair added. If the value exists in the dictionary it is updated with the new value.

1.4 fold

`fold` traverses through the dictionary in-order and applies a given function to each key value-pair. It should be noted that due to the order of the recursive calls, the fold will always be performed in-order from smallest to highest key.

1.5 rebalance

`rebalance` makes a list of all elements in our tree, chooses a pivot element in the middle as root and splits the list in two halves. Each half becomes a subtree which in their are split in half to recursively create a binary tree. Since its always split in halves, all subtrees will be balanced.

1.6 keys

`keys` make use of our `fold`-function and accumulating all keys to a single list.

1.7 samekeys

Takes two dictionaries as argument, then uses generates list of keys from the `keys`-function and then compares these two by using `list_equals`-function element by element.

2 Cuttings

`papercuts` uses an auxilary function `papercutsaux` to step through the list

3 Lazy permutations

4 Lazy word generation