Advanced Functional Programming
Uppsala University – Autumn 2012
Assignment 2

Anders Hassis Jonatan Jansson

January 18, 2013

1 Dictionaries

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

The *node*-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 `dicttree`, to the left.

1.1 create-dictionary

`create-dictionary` is defined as a constructor for our `dicttree` structure. If no argument is given, `string-compare` is 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 `lookup-aux` which finds a value in a node and its subnodes, given a key. `lookup-aux` returns a list of two values, the found value and *t*, or *nil* and *nil* if the key was not found. The reason for the second value in the list is to make it possible to store *nil* in the dictionary. Without the extra flag it would be impossible to know if *nil* means not found, or that the stored value was actually *nil*.

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 together with the `print-keys`-function that just prints the current key.

1.7 samekeys

Since our `fold-aux` makes sure the tree is always sorted the same way. `samekeys` compares the keys `dict1` and `dict2` from the `keys`-function and checks if they are the same using the built-in `equal`-function. This is however not an optimal solution and could be improved.

2 Macros

2.1 with-keys

`with-keys` is implemented as a recursive macro applying a function to the key value-pair of the root-node of the dictionary and then performs a recursive call on each child node. By doing this the function will be applied to all key value-pairs in the dictionary.

2.2 match-pattern

`match-pattern` chooses an expression to evaluate depending on a given expression and patterns to match the expression. The macro is implemented as three macros and one function. Each of these will be described briefly.

The `match-pattern`-macro is the main macro, evaluating the given expression and starts the recursion on the list of patterns.

The `pattern-match`-macro is made to step through each pattern, one at a time. If a pattern matches the expression the expression belonging to that pattern is evaluated. If it does not match a recursive call is made on the rest of the patterns. If no pattern matches `nil` is returned.

The `ismatching`-function takes an evaluated expression and a pattern and tries to match them. If there is a match `t` is returned, otherwise `nil`. The match is done by always taking the first element in the pattern. If it is an atom it can be mapped to the first element of the expression and a recursive call is made to check the rest of the pattern. If the first element is a list two recursive calls will be made, one to make sure that the list matches the first element of the expression, and one to match the rest of the pattern. The recursion only returns `t` if the lists ends at the same time, i.e. they match all the way. If the list would not match somewhere the recursion is stopped and `nil` is returned.

The `match`-macro works by the same principle as the `ismatching`-function but instead of returning `t` or `nil` it expands to a nested `let`-expression with all the assignments that have to be done in the matching. The recursion ends when there are no more assignments to do and the body-expression is inserted. Note that there is no case for patterns that doesn't match. The reason for this is that this macro is only supposed to be used on a pattern that is confirmed to match by the `ismatching`-function.

The result of using the `match-pattern`-macro will be code that evaluates the main expression and then tries to match each pattern until a match is found. When a match is found code to do all assignments in the matching is added using nested `let`-expressions and the innermost expression will be the body corresponding to the chosen match-case. The code will evaluate by doing all assignments and finally evaluating the body.