Middle East Technical University Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring 2016-2017

Homework 2 - WordTree

Due date: 16 04 2017, Sunday, 23:55

1 Objective

This homework aims to familiarize you with more advanced functional programming concepts by implementing operations on a recursive data structure using Haskell.

Keywords: *functional programming, recursive data structures*

2 Problem Definition

In this homework, you are going to implement functions that operate on **WordTree**. WordTree is a n-ary tree structure where common prefixes of words are stored at the parents and remaining parts are stored at the leaf of the trees in a **sorted** manner. It is used for fast addition, deletion and search of the prefixes and words.

Data type of feature structure is defined as follows in Haskell.

```
data WordTree = Word String | Subword String [WordTree] | Root [WordTree]
```

Using this data structure words can be stored and accessed in an efficient manner, allow for fast searching for prefixes and words under it. It has three constructors:

- Root constructor marks the beginning of the WordTree and only contains list of its branches. All WordTree structures start with a Root constructor. Root constructor cannot be used on any lower branch. Root with an empty list represent an empty[1] WordTree. Example:

```
emptyTree = Root []
```

[1] `emptyTree` will be used throughout the homework text to represent an empty tree for various operations.

- Word constructor is used to represent two things. First one is complete words, and they are used if there are no other word in the WordTree with matching prefixes. Secondly, it is used for remaining part of a word. When used in this manner they contain the remaining part of the word where the prefixes stored at the parent. They are always at the leaves of the tree. Examples:

```
exampleTree1 = Root [Word "Hello", Word "World"]
exampleTree2 = Root [Subword "Hel" [Word "lo", Word "p"]]
```

  In exampleTree1, the words "Hello" and "World" are stored fully because their first letter is different, therefore have no matching letters starting from the beginning. In exampleTree2, only "lo" and "p" are stored with the Word constructor because they contain a common "Hel" prefix in the beginning and that is stored in their parent.

- Subword constructor is used for storing prefixes of words. It must have at least two branches of any kind. Subword with less than that can be represented with either with Subword or Word depending on whether they have a child or not. Example:

```
Root [Subword "Work" [Word "",Word "er"]]
```

3 Specifications

In this second homework, you are going to implement addition, deletion, word pathing, prefix search and display operations on `WordTree`.

3.1 The `WordTree` Module

You are going to implement `WordTree` functions in this module using the WordTree data structure. It is defined below:

```
module WordTree(WordTree(Word, Subword, Root), emptyTree, getAllPaths, addWords, ↩
    deleteWords, getWordsBeginWith) where

data WordTree = Word String | Subword String [WordTree] | Root [WordTree]
```

Please note that exporting constructors for data structures is not standard for Abstract Data Types, the reason it is exported here is for grading your homeworks.
Implementation details for the module are elaborated in the following subsections. Trees given in the arguments of functions are always sorted.

3.2 `WordTree` as an Instance of Show - 15 pts

`WordTree` data type should implement the functionality of the `Show` typeclass to display the data structure in a readable form.

```
instance Show WordTree where
    ...
```

Display of `WordTree` instances should obey the following specifications.

- Print Subword in the following format:

  `<indentation><prev_sub1><prev_sub2>..<prev_subN><subword>:<newline>`

- Print Word in the following format:

  `<indentation><prev_sub1><prev_sub2>..<prev_subN><word><newline>`

- Number of spaces used for indentation is calculated using the formula $indentation = (Depth - 1) \times 2$ where $Depth = 0$ at Root.

Examples:

```
*WordTree> emptyTree

*WordTree> show emptyTree
""
*WordTree> Root [Word "Dam",Word "Plane",Word "Ship"]
Dam
Plane
Ship

*WordTree> Root [Subword "Da" [Subword "m" [Word "",Subword "ag" [Word "e"↩
    ,Word "ing"]],Word "rk"]]
Da:
  Dam:
    Dam
    Damag:
      Damage
      Damaging
  Dark

*WordTree>
```

```
*WordTree> Root [Subword "Ca" [Subword "n" [Word "",Word "teen"],Word "ptain",Subword "r" [Word "",Subword "r" [Subword "ie" [Word "d",Word "s"],Word "y"]]],Subword "He" [Word "ck",Subword "l" [Word "lo",Subword "p" [Word "",Subword "e" [Word "d",Word "r"],Word "ing"]]]]
Ca:
  Can:
    Can
    Canteen
  Captain
  Car:
    Car
    Carr:
      Carrie:
        Carried
        Carries
      Carry
He:
  Heck
  Hel:
    Hello
    Help:
      Help
      Helpe:
        Helped
        Helper
      Helping

*WordTree>
```

3.3 Function getAllPaths - 10 pts

```
getAllPaths :: WordTree -> [[String]]
```

This function returns all the paths each word takes in the tree. Paths are represented with a list starting from the root. Each Subword in the path from root to leaf is a distinct element in the list. Last element of the list is the Word at the leaf node.
Examples:

```
*WordTree> getAllPaths emptyTree
[]
*WordTree> let exampleTree1 = Root [Word "Hello", Word "World"]
*WordTree> getAllPaths exampleTree1
[["Hello"],["World"]]
*WordTree> let exampleTree2 = Root [Subword "Hel" [Word "lo", Word "p"]]
*WordTree> getAllPaths exampleTree2
[["Hel","lo"],["Hel","p"]]
```

```
*WordTree> let test1 = Root [Subword "He" [Word "ck",Subword "l" [Word "lo↩
    ",Subword "p" [Word "",Subword "e" [Word "d",Word "r"],Word "ing"]]]]
*WordTree> test1
He:
  Heck
  Hel:
    Hello
    Help:
      Help
      Helpe:
        Helped
        Helper
      Helping

*WordTree> getAllPaths test1
[["He","ck"],["He","l","lo"],["He","l","p",""],["He","l","p","e","d"],["He↩
    ","l","p","e","r"],["He","l","p","ing"]]
*WordTree> let test2 = Root [Subword "Ca" [Subword "n" [Word "",Word "teen↩
    "],Word "ptain",Subword "r" [Word "",Subword "r" [Subword "ie" [Word "d↩
    ",Word "s"],Word "y"]]],Subword "He" [Word "ck",Subword "l" [Word "lo",↩
    Subword "p" [Word "",Subword "e" [Word "d",Word "r"],Word "ing"]]]]
*WordTree> test2
Ca:
  Can:
    Can
    Canteen
  Captain
  Car:
    Car
    Carr:
      Carrie:
        Carried
        Carries
      Carry
He:
  Heck
  Hel:
    Hello
    Help:
      Help
      Helpe:
        Helped
        Helper
      Helping

*WordTree>
```

```
*WordTree> getAllPaths test2
[["Ca","n",""],["Ca","n","teen"],["Ca","ptain"],["Ca","r",""],["Ca","r","r↩
   ","ie","d"],["Ca","r","r","ie","s"],["Ca","r","r","y"],["He","ck"],["He↩
   ","l","lo"],["He","l","p",""],["He","l","p","e","d"],["He","l","p","e",↩
   "r"],["He","l","p","ing"]]
*WordTree> let test3 = Root [Subword "F" [Word "alse",Subword "i" [Word "↩
   asco",Word "le"]],Subword "Re" [Word "aper",Subword "po" [Word "",Word ↩
   "sitory"]],Subword "T" [Subword "a" [Subword "il" [Word "",Word "or"],↩
   Word "p"],Word "esla"]]
*WordTree> test3
F:
  False
  Fi:
    Fiasco
    File
Re:
  Reaper
  Repo:
    Repo
    Repository
T:
  Ta:
    Tail:
      Tail
      Tailor
    Tap
  Tesla

*WordTree> getAllPaths test3
[["F","alse"],["F","i","asco"],["F","i","le"],["Re","aper"],["Re","po",""↩
   ],["Re","po","sitory"],["T","a","il",""],["T","a","il","or"],["T","a","↩
   p"],["T","esla"]]
```

3.4 Function addWords - 27.5 pts

```
addWords :: WordTree -> [String] -> WordTree
```

This function adds all the words in the second argument to the tree. When performing addition follow the specifications for Wordtree described in Problem Definition. It should also should obey the following specifications:

- When used with an empty list, return the original tree.
- List given in the second argument are not guaranteed to be sorted.
- At every level, words are stored in lexicographic order.
- WordTree should not contain any duplicates.
- Subword must be the the longest common prefix two words share.
- If the word is both a subword and word, It should have an empty string at its leaf.

Examples:

```
*WordTree> let tree1 = addWords emptyTree []
*WordTree> tree1

*WordTree> let tree2 = addWords emptyTree ["Hello"]
*WordTree> tree2
Hello

*WordTree> let tree3 = addWords tree2 ["World", "Help"]
*WordTree> tree3
Hel:
  Hello
  Help
World

*WordTree> let tree4 = addWords tree2 ["Hi"]
*WordTree> tree4
H:
  Hello
  Hi

*WordTree>
```

```
*WordTree> let tree5 = addWords emptyTree ["Helper", "Help", "Helped", "↩
    Helping", "Hello", "Heck"]
*WordTree> tree5
He:
  Heck
  Hel:
    Hello
    Help:
      Help
      Helpe:
        Helped
        Helper
      Helping

*WordTree> let tree6 = addWords emptyTree ["Helper", "Help", "Helped", "↩
    Helping", "Hello", "Heck", "Car", "Carry", "Can", "Carries", "Carried",↩
     "Captain", "Canteen"]
*WordTree> tree6
Ca:
  Can:
    Can
    Canteen
  Captain
  Car:
    Car
    Carr:
      Carrie:
        Carried
        Carries
      Carry
He:
  Heck
  Hel:
    Hello
    Help:
      Help
      Helpe:
        Helped
        Helper
      Helping

*WordTree>
```

```
*WordTree> let tree7 = addWords emptyTree ["File", "Repository", "Repo", "↩
    Reaper", "False", "Fiasco", "Tail", "Tailor", "Tap", "Tesla"]
*WordTree> tree7
F:
  False
  Fi:
    Fiasco
    File
Re:
  Reaper
  Repo:
    Repo
    Repository
T:
  Ta:
    Tail:
      Tail
      Tailor
    Tap
  Tesla

*WordTree> let tree8 = addWords tree2 ["Helper", "Help", "Helped", "↩
    Helping", "Hello", "Heck"]
*WordTree> tree8
He:
  Heck
  Hel:
    Hello
    Help:
      Help
      Helpe:
        Helped
        Helper
      Helping

*WordTree>
```

```
*WordTree> let tree9 = addWords tree4 ["Helper", "Help", "Helped", "↩
    Helping", "Hello", "Heck", "Car", "Carry", "Can", "Carries", "Carried",↩
    "Captain", "Canteen"]
*WordTree> tree9
Ca:
  Can:
    Can
    Canteen
  Captain
  Car:
    Car
    Carr:
      Carrie:
        Carried
        Carries
      Carry
H:
  He:
    Heck
    Hel:
      Hello
      Help:
        Help
        Helpe:
          Helped
          Helper
        Helping
  Hi

*WordTree>
```

```
*WordTree> let tree10 = addWords tree6 ["File", "Repository", "Repo", "↩
   Reaper", "False", "Fiasco", "Tail", "Tailor", "Tap", "Tesla"]
*WordTree> tree10
Ca:
  Can:
    Can
    Canteen
  Captain
  Car:
    Car
    Carr:
      Carrie:
        Carried
        Carries
      Carry
F:
  False
  Fi:
    Fiasco
    File
He:
  Heck
  Hel:
    Hello
    Help:
      Help
      Helpe:
        Helped
        Helper
      Helping
Re:
  Reaper
  Repo:
    Repo
    Repository
T:
  Ta:
    Tail:
      Tail
      Tailor
    Tap
  Tesla

*WordTree>
```

3.5 Function deleteWords - 27.5 pts

```
deleteWords :: WordTree -> [String] -> WordTree
```

This function deletes all the words in the second argument from the tree. When performing deletion follow the specifications for Wordtree described in Problem Definition. It should also should obey the following specifications:

- When used with an empty list, return the original tree.
- List given in the second argument are not guaranteed to be sorted.
- If any word inside the list is not in the WordTree, skip that word.
- After deletion if the Subword has only single child, there are two cases. If the child is a Word, you need to merge the strings and convert Subword into a Word. If the child is a Subword, you need to merge the words and the merged Subword should adopt the children of the child. Examples of this can be seen in tree11 and tree12.

Examples[2]:

```
*WordTree> deleteWords emptyTree []

*WordTree> deleteWords emptyTree ["Hello"]

*WordTree> deleteWords tree2 []
Hello

*WordTree> deleteWords tree2 ["Hello"]

*WordTree> deleteWords tree3 ["Help"]
Hello
World

*WordTree> deleteWords tree3 ["World"]
Hel:
  Hello
  Help

*WordTree> deleteWords tree3 ["World"]
Hel:
  Hello
  Help

*WordTree> deleteWords tree3 ["World", "Help"]
Hello

*WordTree> deleteWords tree3 ["World", "Help", "Hello"]

*WordTree>
```

[2]Trees given in the addwords section is used

```
*WordTree> deleteWords tree6 ["Car", "Carry", "Can", "Carries", "Carried",↩
    "Captain", "Canteen"]
He:
  Heck
  Hel:
    Hello
    Help:
      Help
      Helpe:
        Helped
        Helper
      Helping

*WordTree> deleteWords tree10 ["Reaper", "Repo", "Carries", "Can"]
Ca:
  Canteen
  Captain
  Car:
    Car
    Carr:
      Carried
      Carry
F:
  False
  Fi:
    Fiasco
    File
He:
  Heck
  Hel:
    Hello
    Help:
      Help
      Helpe:
        Helped
        Helper
      Helping
Repository
T:
  Ta:
    Tail:
      Tail
      Tailor
    Tap
  Tesla

*WordTree>
```

```
*WordTree> let tree11 = addWords emptyTree ["Car", "Carpenter"]
*WordTree> tree11
Car:
  Car
  Carpenter

*WordTree> deleteWords tree11 ["Car"]
Carpenter

*WordTree> let tree12 = addWords emptyTree ["Can", "Car", "Canopy"]
*WordTree> tree12
Ca:
  Can:
    Can
    Canopy
  Car

*WordTree> deleteWords tree12 ["Car"]
Can:
  Can
  Canopy

*WordTree>
```

3.6 Function getWordsBeginWith - 20 pts

```
getWordsBeginWith :: WordTree -> String -> [String]
```

This function retrieves all the words start with prefix given in the second argument.If given with an empty string, it should return all the words.
You are not allowed to implement getWordsBeginWith function by manipulating lists coming from the getAllPaths or any other similar function. Your function should get the results directly from the tree instead of eliminating them from a list of words. This defeats the whole purpose of WordTree. getWordsBeginWith function should do a fast prefix search using the WordTree structure. Otherwise you will not receive any points from this function.
Examples[3]:

```
*WordTree> getWordsBeginWith tree5 "H"
["Heck","Hello","Help","Helped","Helper","Helping"]
*WordTree> getWordsBeginWith tree5 "Helpe"
["Helped","Helper"]
*WordTree> getWordsBeginWith tree6 ""
["Can","Canteen","Captain","Car","Carried","Carries","Carry","Heck","Hello↩
    ","Help","Helped","Helper","Helping"]
*WordTree> getWordsBeginWith tree7 "F"
["False","Fiasco","File"]
*WordTree> getWordsBeginWith tree7 "Rea"
["Reaper"]
*WordTree> getWordsBeginWith tree7 "Ta"
["Tail","Tailor","Tap"]
*WordTree> getWordsBeginWith tree7 "T"
["Tail","Tailor","Tap","Tesla"]
*WordTree> getWordsBeginWith tree10 "Re"
["Reaper","Repo","Repository"]
*WordTree> getWordsBeginWith tree10 "Tail"
["Tail","Tailor"]
*WordTree> getWordsBeginWith tree10 "Help"
["Help","Helped","Helper","Helping"]
*WordTree> getWordsBeginWith tree10 "Carr"
["Carried","Carries","Carry"]
```

[3]Trees given in the addwords section is used

4 Regulations

- **Programming Language:** You must code your program in Haskell. Your submission will be compiled with `ghc` on Cengclass. You are expected make sure your code compiles successfully with `ghc` on CengClass.
- **Modules:** You are not allowed to import any modules. Otherwise you will receive 0 for this assignment.
- **Late Submission:** For programming assignments 10 late days can be distributed between all assignments. Each assignment cannot be submitted more than 3 days late.
- **Cheating:** Using code from any source other than your own is considered cheating. This includes but not limited to internet sources, previous homeworks, and friends. In case of cheating, the university regulations will be applied.
- **Newsgroup:** You must follow the newsgroup (news.ceng.metu.edu.tr) for discussions and possible updates on a daily basis.
- **Grading:** This homework will be graded out of 100.

5 Submission

Submission will be done via Cengclass. You are expected to either edit the file HW2.hs directly on Cengclass or upload it after working on your computer.