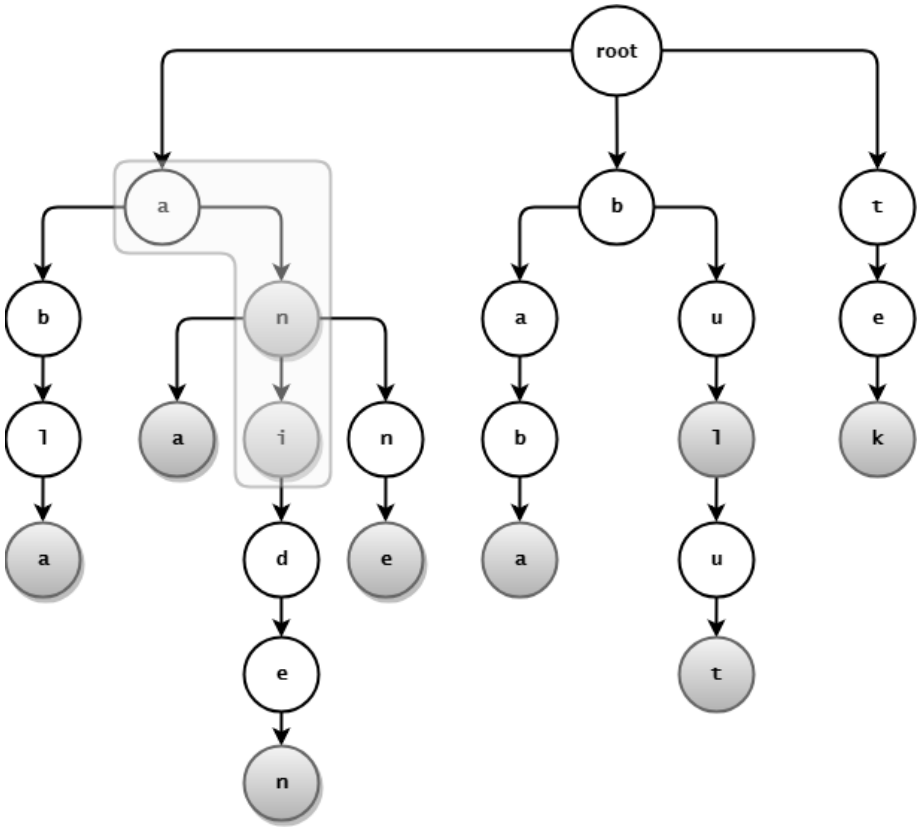
BLG468E – Functional Programming

Term Project

In this assignment, you are going to implement a trie data structure a.k.a. prefix tree which is a custom tree data structure.

Implementation Details

Trie is a data structure optimized for fast searching and prefix checking of strings. For a word with L letters, You can look up a word in $O(L)$ time in a trie. The nodes of a trie are associated with letters and a boolean flag which indicates the end of a word. Check out an illustration of a trie which contains following words: *abla, an, ana, ani (highlighted), aniden, anne, baba, bul, bulut, tek.*



A Trie Tree

As can be seen, some nodes are shadowed in the illustration to indicate the end of a word (which means the end flag is set).

In this assignment, you will use Data.Map module extensively (a.k.a. Dictionary or Hash Table in other languages) from the Haskell library. If you are not familiar with this module, you can look at the *Tips & Trick* section for tutorials.

The definition of trie is the following:

```
data Trie = Trie {end :: Bool, children :: Map Char Trie}
```

Using this data type, you are asked to implement the following functions:

- **`empty:`** Creates a new empty trie.
- **`insert:`** Takes a word (string) and a trie. Then, the given word inserted into the given trie and the resulting trie is returned. While you are adding words, sometimes you need to create a new node, you can use the `empty` function for this purpose.
- **`insertList:`** Takes a list of words as a parameter and returns a trie that contains the given words. We are expecting you to use the `fold` function (and the other functions you implemented) to implement this function.
- **`search:`** Takes a word and a trie. It returns `True` if the given word is in the trie, `False` otherwise.
- **`getWords:`** Takes a trie and returns all the words present in the trie. This function is a bit hard to implement but my suggestion would be to write a function that traverses the trie (while keeping track of where it came from) and also keeps an accumulator which found words are inserted into. You may want to use the `toList` function from the `Data.Map` module in the implementation.
- **`prefix:`** Takes a string (prefix) and a trie. It returns a list of strings that start with the given prefix. If there is no word starting with that prefix, it should return `Nothing`. You may want to use the `getWords` function to simplify the implementation of this function.

You can find the signatures of these functions and more in the appendix. Along with these functions, you are going to need `IO` functions to complete the assignment. A user interface as below is required:

```
a) Add Word
s) Search Word
f) Find words with prefix
p) Print all words
e) Exit
Enter the action:
a
Enter word/prefix:
kalem
New word is added!
```

Sample UI

You can find more pictures of the UI in the appendix. Also, a sample Calico test file will be provided. With that, you can check whether your program is compliant with the requested UI. Nevertheless, you are free to handle `IO` operations as you like but I have some suggestions:

1. Define an `Action` data type.
2. Define a `convertAction` function. (What should be the type of the function?)
3. Define a `getInput` function. You can use this function for getting an action and a word/prefix from the user.
4. Define a `doAction` function. After getting the action and the word/prefix from the user, this function can be used for performing the action. Note that you should treat the `Add` action differently since it changes the trie.

You may or may not follow these suggestions but don't forget that writing code in Haskell gets easier if you divide your program into meaningful functional parts.

Your program should take a file path as a command line argument. The file will contain the words that you are going to insert into the trie. You can find a sample in the appendix.

Name your module *trie.hs* and make sure that the following command creates an executable:

```
ghc trie.hs -o trie
```

Also make sure that your program can be executed as:

```
./trie words.txt
```

Tips & Tricks

- Some great tutorials for Data.Map: haskell-containers and learnyouhaskell.
- To handle file I/O operations, you need to use the System.IO module. You can find great examples in this link.
- You may want to use unlines & lines to manipulate strings.
- To get command line arguments, you need the System.Environment module.
- You may want to use `fromMaybe` and `maybe` functions from `Data.Maybe` module.
- Try to use *currying, higher order functions* and `($)` & `(.)` operators whenever possible.

Restrictions

- You are not allowed to use any construct that is not covered in the lecture as of May 5th.
- From the `Data.Map` module, you are only allowed to use these functions: `empty, insert, lookup, (!?), toList.`
- You can send me an e-mail (colakoglut@itu.edu.tr) if you have any questions about restrictions.

Appendix

Pictures

```
mostwanted hw $ ./trie words.txt
a) Add Word
s) Search Word
f) Find words with prefix
p) Print all words
e) Exit
Enter the action:
p
List of words in dictionary:
an
abla
ani
ana
aniden
anne
baba
bul
bulut
tek
```

```
a) Add Word
s) Search Word
f) Find words with prefix
p) Print all words
e) Exit
Enter the action:
s
Enter word/prefix:
silgi
NOT exist!
```

```
a) Add Word
s) Search Word
f) Find words with prefix
p) Print all words
e) Exit
Enter the action:
s
Enter word/prefix:
bulut
Exists in dictionary!
```

```
a) Add Word
s) Search Word
f) Find words with prefix
p) Print all words
e) Exit
Enter the action:
f
Enter word/prefix:
an
Found words:
ani
ana
aniden
anne
```

```
a) Add Word
s) Search Word
f) Find words with prefix
p) Print all words
e) Exit
Enter the action:
f
Enter word/prefix:
sa
No words found with that prefix!
```

UI Pictures (Sorry about the quality)

Code Skeleton (trie.hs)

```haskell
import qualified Data.Map -- as M (can be shortened)
import Data.Maybe
import System.Environment
import System.IO
import Prelude hiding (Word)

data Trie = Trie {end :: Bool, children :: Map Char Trie}
type Word = String

empty :: Trie
empty = undefined

insert :: Word -> Trie -> Trie
insert = undefined

insertList :: [Word] -> Trie
insertList = undefined

search :: Word -> Trie -> Bool
search = undefined

getWords :: Trie -> [Word]
getWords = undefined

prefix :: Word -> Trie -> Maybe [Word]
prefix = undefined
```

Sample Dictionary (words.txt)

```
abla
an
ana
ani
aniden
anne
baba
bul
bulut
tek
```

Calico Test File

(Will be announced later)