Assignment 3: Storing Words in a Trie
CISC 260, Winter 2018
Topics: Haskell Algebraic Types

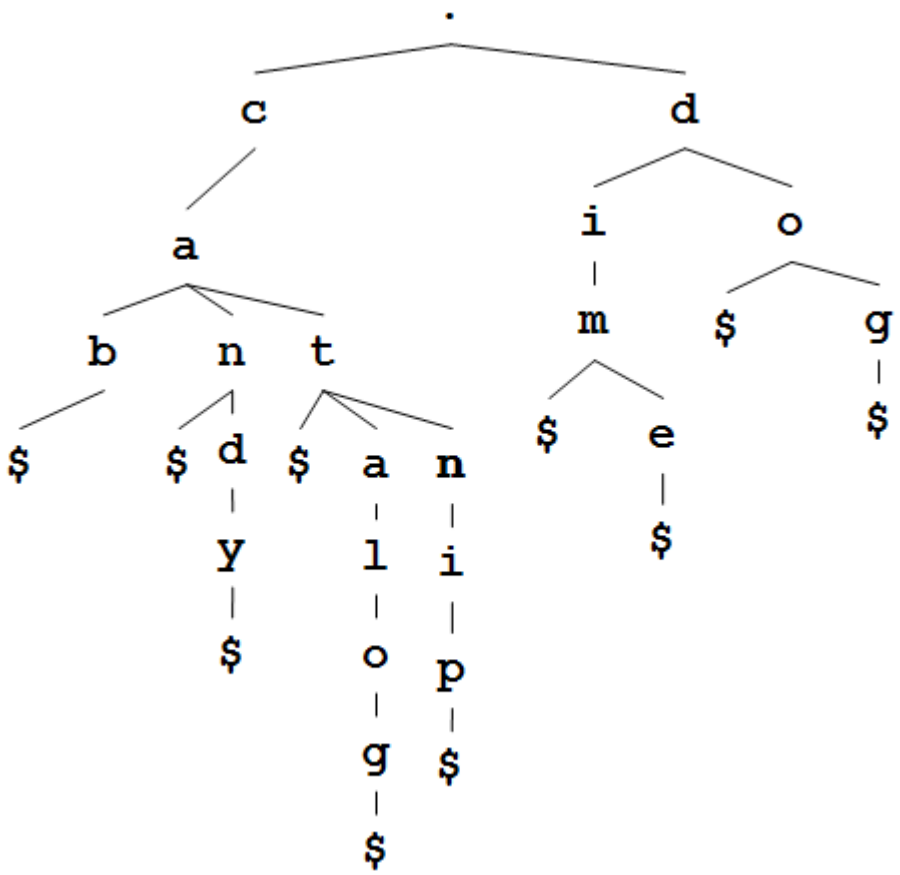
Due Date: 9 a.m. on Wednesday, March 7

Administrative Notes:
For this assignment and every other assignment in CISC 260, I will program a 24-hour "grace period" into OnQ. This means you can submit your assignment up to 24 hours late if you need to, but we will deduct 10% from your mark. After the grade period is over we will not accept any more submissions.

If there are special circumstances such as illness that prevent you from completing the assignment on time please submit the appropriate form to the Arts & Science portal https://www.queensu.ca/artsci/accommodations (or the equivalent form for your home faculty if you are registered in a faculty other than Arts & Science). The instructor will be notified when a student submits one of these forms; you do not have to contact her directly.

Summary:
A "trie" is a data structure designed to store sequences of numbers or characters for efficient retrieval in situations where some of the sequences may be prefixes of others. The name comes from the middle characters of the word "re**trie**val" and may be pronounced like "try" or like "tree". (I usually say "try" to avoid confusion with other kinds of trees such as binary search trees.)

In this assignment we'll use tries to store words. Here is a picture of small trie:



The tree pictured on the previous page contains the words "cab", "can", "candy", "cat", "catalog", "catnip", "dim", "dime", "do" and "dog". Each path from the root to a leaf of the trie represents one of those ten words, with the beginning of the word marked with the special character "." and the end marked with the special character "$". All other characters in the tries for this assignment will be lower-case letters.

A trie consists of a starting letter plus zero or more sub-tries. The sub-tries must be in alphabetical order by their starting characters. Since '$' comes before 'a' in the ASCII character code, if there's a sub-trie with '$' at the root it will always be the first sub-trie.

No two sub-tries of the same tree may have the same starting letter. For example, in the tree above the words "cat", "catalog" and "catnip" all share the same path through the letters 'c', 'a' and 't'.

There is a file called `Assignment3.hs` from OnQ with these instructions (under "Contents", then "Assignments". Download a copy as a starting point for your solution. This file contains a definition of an algebraic type for tries and two example tries that you may use for testing. I encourage you to make your own tries for testing and experiments as well. You may NOT change the definition of the algebraic type.

Terminology: I have invented a few terms to talk about different kinds of tries. A "complete trie" is a trie with a '.' character at the root, containing complete words starting with '.' and ending with '$'. A "sub-trie" is a trie with either a letter or '.' at the root, representing suffixes of complete words. (A sub-trie with '$' at the root would have no children and represent an empty list of strings.)

The trie in the picture on the previous page is a complete trie. A "legal trie" is a trie that obeys all the rules stated above -- all paths ending with '$', lower-case letters everywhere else (except for '$'s at the end of words and a '.' at the root if this is a complete trie), and the sub-tries of each trie in alphabetical order with no duplicates. All of your functions may assume that the tries passed as parameters are legal tries.

Requirements: Your job is to add four additional functions to this module. Please add them to your copy of Assignment3.hs and submit the modified file to OnQ.

1. `searchWord`: This function must take a string and a complete trie as parameters and return `True` if the string is a word in that tree and `False` otherwise. You may assume that the string consists only of lower-case letters; it should not contain "." or "$" or any other characters.

2. `wordsInTrie`: This function must take a complete trie as a parameter and return a list of all of the words in the tree, in alphabetical order. The strings in the list must not include the starting '.' or ending '$'.

3. `addWordToTrie`: This function must take a word and a trie as parameters and return a new trie which is the parameter trie with the word added to it. The word parameter will be a string of lower-case letters, without the starting '.' or ending '$'. If the word is already in the trie dont' add a duplicate.

4. `createTrie`: This function must take a list of words (without starting '.'s or ending '$'s) and return a trie containing all of these words. For full marks, this function must use your `addWordToTrie` function and a fold (one of `foldl`, `foldl1`, `foldr`, or `foldr1`, as we discussed in the topic about higher-order functions). A correct solution that does not make use of a fold will receive partial credit.

Feel free to create helper functions in addition to these required functions.

There is a transcript of a run of my solution on the following pages.

```
*Assn3Solution> webTrie
.
  c
    a
      b
        $
      n
        d
          y
            $
      t
        $
        a
          l
            o
              g
                $
        n
          i
            p
              $
  d
    i
      m
        $
        e
          $
    o
      $
      g
        $
```

```
*Assn3Solution> addWordToTrie "cart" webTrie (new characters highlighted in red)
.
  c
    a
      b
        $
      n
        d
          y
            $
      r
        t
          $
      t
        $
        a
          l
            o
              g
                $
        n
          i
            p
              $
  d
    i
      m
        $
        e
          $
    o
      $
      g
        $
```

```
*Assn3Solution> searchWord "catalog" webTrie
True
*Assn3Solution> searchWord "dim" webTrie
True
*Assn3Solution> searchWord "dime" webTrie
True
*Assn3Solution> searchWord "cand" webTrie
False
*Assn3Solution> searchWord "log" webTrie
False
*Assn3Solution> wordsInTrie webTrie
["cab","candy","cat","catalog","catnip","dim","dime","do","dog"]
*Assn3Solution> createTrie []
.
```

```
*Assn3Solution> createTrie
["queen","king","quite","kind","quantum","kill","quantity","kite","quality","keep"]
.
  k
    e
      e
        p
          $
    i
      l
        l
          $
      n
        d
          $
        g
          $
      t
        e
          $
  q
    u
      a
        l
          i
            t
              y
                $
        n
          t
            i
              t
                y
                  $
            u
              m
                $
      e
        e
          n
            $
      i
        t
          e
            $
```

VERY IMPORTANT RESTRICTIONS:

- You may not use your `wordsInTrie` function to help you implement any of the other required functions. For example, you may not implement `searchWord` by creating a list of all of the words in the trie and then searching that list. This would defeat the whole reason for using a trie data structure to save space and searching time.
- You may not change the supplied definition of the `Trie` type in any way. If you use a different way to represent a trie you will not receive any points for this assignment.

Helper Functions: You will need some helper functions to implement the required functions. The following is a list of some of the helper functions I used in my solution. You don't have to use the same helper functions, but you are welcome to use some or all of my ideas if you wish. I'm including this list as a hint about some strategies if you're having trouble breaking this assignment down into pieces.

- `searchString`: takes a string and a trie as parameters and searches for the string in the trie and returns `True` or `False`. This differs from `searchWord` because the string will end with '$' but might not start with '.' and the trie may be a complete trie or a sub-trie.
- `searchTrieList`: takes a string and a list of tries as parameters and searches for the strings in the list of tries and returns `True` or `False`.
- `addStringToTrie`: takes a string and a trie ands returns a new trie which is a copy of the second parameter with the string added to it. The string won't necessarily begin with '.' (because it might be only part of a word) but it will end with '$'.
- addStringToTrieList: takes a string and a list of tries as parameters and returns a modified list of tries with the string added to them. The list of tries will be a list of children of a larger trie, so none will have the same starting character. The result will be a copy of the list with one the string added to one of the tries OR the list with a new trie added to hold the string.
- `stringsInTrie`: takes a trie or sub-trie and returns an alphabetical list of all of the strings in the trie, including '.' and '$' characters

Marking Scheme:

- searchWord: 4
- wordsInTrie: 4
- addWordToTrie: 4
- createTrie: 4

total: 16

What to hand in (please read carefully) : Hand in a copy of `Assignment3.hs`, with the required functions added to it (as well as any helper functions you wish to include). You must not change the name or header line of this file; the name just still be Assignment3.hs and the header line it must still be

```
module Assignment3 where
```

If you don't use this name and this header line your file won't work with our testing code and we will deduct a 10% administrative penalty. (Fixing everyone's file names and header lines in a class this size adds up to a lot of unnecessary work!)

Type Declarations: Type declarations are not required for this assignment; no points will be given for including them and no points will be deducted for leaving them out. I do recommend using them because they can result in better error messages if you get a type wrong, but since you'll be starting this assignment before we've finished the "Types and Type Checking" topic there won't be points given for type declarations (or deducted for their absence).