Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring 2020-2021

Programming Exam 4

Due date: 1 May 2021, Saturday, 23:59

1 Problem Definition

In this final programming examination for Haskell, you will once again be dealing with two parts. In the first part, you will be performing *safe* conversion between data types. And in the second part, you will be dealing with a dictionary-style structure implemented as a special tree. They're both based around a little inverse phone book implementation. Let's have some fun!

1.1 General Specifications

- The signatures of the functions, their explanations and specifications are given in the following section. Read them carefully.
- Make sure that your implementations comply with the function signatures.
- You may define any number of helper function(s) as you need.
- You can (and sometimes even should) use previous functions inside your new function definitions. *Some* exercises are designed to build up!
- You are not allowed to import any *extra* modules for this exam. `Data.Maybe` is imported for your convenience although it's not necessary. Please stick to the `Prelude` for the rest!
- Observe that the exam is out of 120 points. Each function has 4 extra points included!

1.2 Quick VPL Tips (In case you've forgotten them!)

- Evaluation is fast. If evaluation seems to hang for more than a few seconds, your code is entering an infinite loop or has an abnormal algorithmic complexity. Or you've lost your connection, which is *much* less likely!
- Although the run console does not support keyboard shortcuts, it should still be possible to copy and paste using the right-click menu (Tested on latest versions of Firefox and Chrome).
- Get familiar with the environment. Press the plus shapes button on the top-left corner to see all the options. You can download/upload files, change your font and theme, switch to fullscreen etc. Useful stuff!

2 Part I - Digit Conversion

To start off with, we will have a phone book consisting of phone numbers as keys and contact names as values. Being careful programmers, we want our phone book to contain only digits as keys and not arbitrary strings. So, to make our program safer on the type level, we define a custom digit data type as a lightweight wrapper around `Char` for use in our phone book [1]:

```
newtype Digit = Digit Char deriving (Show, Eq, Ord)
```

Observe that we make our data type convertible to a string, comparable and orderable automatically through the use of `deriving`. Also, we define an *alias* for lists of `Digit`s:

```
type PhoneNumber = [Digit]
```

Our goal in this part is performing *safe* conversions to our new data type. We will be using Haskell's standard `Maybe` type for this.

2.1 toDigit (14 points)

Step one: convert a single character to a `Digit` safely (so, `Maybe Digit`)! If the given character is a digit (characters between `'0'` and `'9'`), it should be converted to a properly wrapped `Digit`. If the given character is not a digit, the function should evaluate to `Nothing`.

Here's the signature and some example runs:

```
toDigit :: Char -> Maybe Digit
```

```
*PE4> toDigit '3'
Just (Digit '3')
*PE4> toDigit '9'
Just (Digit '9')
*PE4> toDigit 'a'
Nothing
*PE4> toDigit '?'
Nothing
*PE4> toDigit '\n'
Nothing
*PE4> toDigit '0'
Just (Digit '0')
*PE4> toDigit '+'
Nothing
```

2.2 toDigits (29 points)

And now it's time for the obvious extension: Converting `String`s to a list of `Digit`s (also named `PhoneNumber`) safely. The rules are simple:

- An empty string is invalid and should evaluate to `Nothing`.

[1] In real life, you should go for `data Digit = Zero | One | ... | Nine` which would be actually safe! The current type can be misused. We didn't go for it in this exam to keep things simpler.

- If **any** of the characters in the given string is not a digit, the string is invalid and once again the function evaluates to `Nothing`.
- Essentially, you should return a list of digits **only** for non-empty strings that contain only digits.

Here's the signature and some example runs:

```
toDigits :: String -> Maybe PhoneNumber
```

```
*PE4> toDigits ""
Nothing
*PE4> toDigits "1"
Just [Digit '1']
*PE4> toDigits "101"
Just [Digit '1',Digit '0',Digit '1']
*PE4> toDigits "5553332211"
Just [Digit '5',Digit '5',Digit '5',Digit '3',Digit '3',Digit '3',Digit '2',Digit
'2',Digit '1',Digit '1']
*PE4> toDigits "abcdef"
Nothing
*PE4> toDigits "ceng242"
Nothing
*PE4> toDigits "555x333"
Nothing
*PE4> toDigits "?27"
Nothing
```

3 Part II - Querying a Phonebook

Congratulations on safely converting the digits![2] Now it's time to go ahead with our little phone book implementation which will be based on the following abstract dictionary tree type:

```
data DictTree k v = Node [(k, DictTree k v)] | Leaf v deriving Show
```

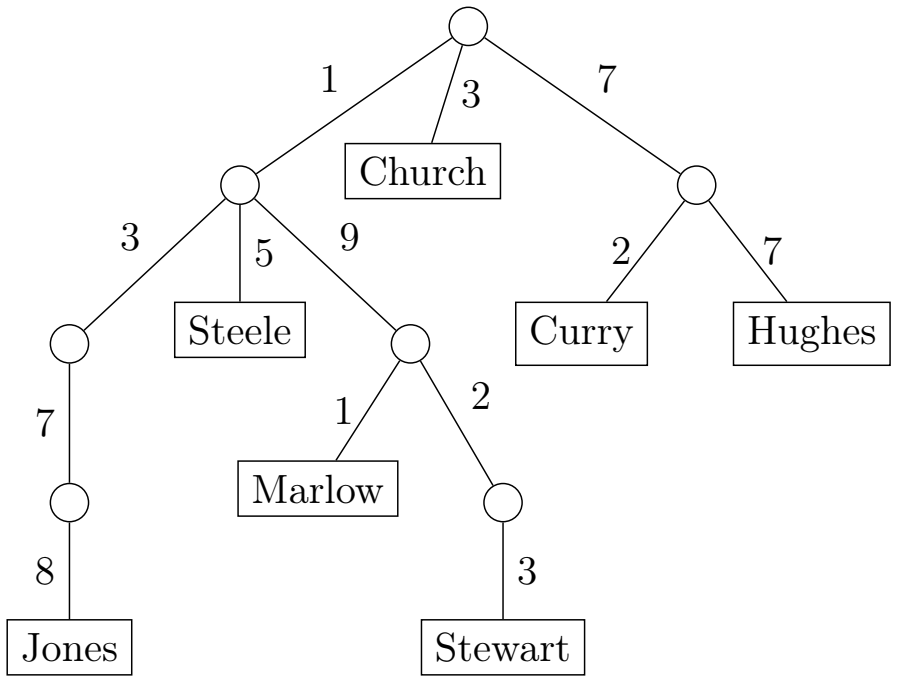
The tree is structured in a way that every internal node contains a bunch of *choices* matching initial parts of the key, with each choice leading to a different subtree. The leaves at the tips contain the actual values in the dictionary. This allows for some fast querying! Also, the parts of the key are kept in sorted order.

An example will make this clearer. First, we define an alias for a concrete type of `DigitTree` which will be the type of our phone book:

```
type DigitTree = DictTree Digit String
```

The parts of our key are `Digit`s and the values are `String`s representing the names in our phone book. Here's an example phone book containing some very short phone numbers associated with a list of particular names, in no particular order. There's also an illustration of the corresponding `DigitTree`.

[2] Unless you're reading ahead... If so, please go back and convert the digits first before I get angry!




```
Phone Book:
-------------
1378: Jones
15: Steele
191: Marlow
1923: Stewart
3: Church
72: Curry
77: Hughes
```

Of course this is all meaningless without the code! So here's the code corresponding to the tree representing the phone book:

```haskell
exampleTree :: DigitTree
exampleTree = Node [
    (Digit '1', Node [
        (Digit '3', Node [
            (Digit '7', Node [
                (Digit '8', Leaf "Jones")])]),
        (Digit '5', Leaf "Steele"),
        (Digit '9', Node [
            (Digit '1', Leaf "Marlow"),
            (Digit '2', Node [
                (Digit '3', Leaf "Stewart")])])]),
    (Digit '3', Leaf "Church"),
    (Digit '7', Node [
        (Digit '2', Leaf "Curry"),
        (Digit '7', Leaf "Hughes")])]
```

This tree will be referred to as `exampleTree` in the below examples and is also present in your VPL template.

To summarize, each digit is part of our key (the whole phone number) and the path of digits going to a leaf corresponds to the phone number of a person (or entity). Your work is going to be defining a few functions for querying phone books.

Before moving on, here are a few final specific details for the careful:

- The digits in the internal node list will always be in sorted order.

- There is no empty tree and the input will never contain any internal nodes with an empty list. Every internal node's list will have at least one pair.
- Similarly, there will be no tree composed of a single leaf. That would be a value with an empty key which does not make much sense!
- The tree cannot contain values at internal nodes. So, our example tree would not be able to have a phone number like `137: Sussman`, since 137 corresponds to an internal node.

3.1 numContacts - 14 pts

Count the number of contacts in the given phone book. That's it. Remember that the input tree will always be valid as per the final definitions given above. Here's the signature and some example runs (do not forget our `exampleTree`!):

```
numContacts :: DigitTree -> Int
```

```
*PE4> numContacts (Node [(Digit '2', Leaf "The One and Only")])
1
*PE4> numContacts $ Node [(Digit '3', Leaf "Mr. Tee"), (Digit '4', Leaf "Ms. Lef")]
2
*PE4> numContacts $ Node [(Digit '1',Node [(Digit '3',Leaf "Luke Unluke")]),(Digit
'4',Node [(Digit '2',Leaf "Kozmos Fehmi"),(Digit '5',Leaf "Tim Time")])]
3
*PE4> numContacts exampleTree
7
*PE4> numContacts $ Node [(Digit '0',Node [(Digit '7',Leaf "Tynetta")]),(Digit '1',
Leaf "Jonathan"),(Digit '2',Node [(Digit '0',Node [(Digit '3',Leaf "Diedre")]),(Dig
it '2',Node [(Digit '5',Leaf "Vesta")])]),(Digit '5',Node [(Digit '1',Leaf "Ilka")]
),(Digit '6',Node [(Digit '0',Node [(Digit '9',Leaf "Farah")]),(Digit '5',Leaf "Cle
mentine")]),(Digit '7',Node [(Digit '7',Leaf "Kenyatta")]),(Digit '8',Node [(Digit
'3',Leaf "Athena")]),(Digit '9',Node [(Digit '7',Node [(Digit '8',Leaf "Lyndsie")])
])]
10
```

3.2 getContacts - 34 pts

Convert the given phone book to a list of pairs, containing all the phone numbers in the phone book along with their associated contact names.

The order of the contacts is important: you should follow the list of each internal node in the given order from start to finish. This means that if you draw the tree, the names of the contacts in the list will correspond to the left-to-right order of the leaves. Remember that the digits in the leaves will always be in sorted order.

As always, the signature and some example runs:

```
getContacts :: DigitTree -> [(PhoneNumber, String)]
```

```
*PE4> getContacts (Node [(Digit '0', Leaf "I, Me & Myself Co.")])
[([Digit '0'],"I, Me & Myself Co.")]
*PE4> getContacts $ Node [(Digit '1', Leaf "Reception"), (Digit '4', Leaf "Emergenc
y"), (Digit '5', Leaf "Bar")]
[([Digit '1'],"Reception"),([Digit '4'],"Emergency"),([Digit '5'],"Bar")]
*PE4> getContacts $ Node [(Digit '3', Node [(Digit '2', Leaf "T"), (Digit '7', Node
 [(Digit '7', Leaf "Z")])])]
[([Digit '3',Digit '2'],"T"),([Digit '3',Digit '7',Digit '7'],"Z")]
*PE4> getContacts exampleTree
[([Digit '1',Digit '3',Digit '7',Digit '8'],"Jones"),([Digit '1',Digit '5'],"Steele
"),([Digit '1',Digit '9',Digit '1'],"Marlow"),([Digit '1',Digit '9',Digit '2',Digit
 '3'],"Stewart"),([Digit '3'],"Church"),([Digit '7',Digit '2'],"Curry"),([Digit '7'
,Digit '7'],"Hughes")]
*PE4> getContacts $ Node [(Digit '1',Node [(Digit '1',Node [(Digit '7',Node [(Digit
 '8',Leaf "Sassy")])])]),(Digit '2',Node [(Digit '6',Leaf "French Fry"),(Digit '8',
Node [(Digit '3',Node [(Digit '1',Leaf "Junior")])])]),(Digit '3',Node [(Digit '9',
Leaf "Frogger")]),(Digit '4',Node [(Digit '0',Leaf "Thor")]),(Digit '6',Node [(Digi
t '4',Node [(Digit '8',Node [(Digit '5',Leaf "Sunshine")])]),(Digit '5',Leaf "Doll"
),(Digit '8',Node [(Digit '5',Leaf "Tomcat")])]),(Digit '7',Node [(Digit '3',Node [
(Digit '3',Leaf "Pecan")]),(Digit '7',Node [(Digit '7',Node [(Digit '4',Leaf "Lulu"
)])])]),(Digit '8',Node [(Digit '6',Node [(Digit '2',Node [(Digit '5',Leaf "Tank")]
)])]),(Digit '9',Leaf "Bud")]
[([Digit '1',Digit '1',Digit '7',Digit '8'],"Sassy"),([Digit '2',Digit '6'],"French
 Fry"),([Digit '2',Digit '8',Digit '3',Digit '1'],"Junior"),([Digit '3',Digit '9'],
"Frogger"),([Digit '4',Digit '0'],"Thor"),([Digit '6',Digit '4',Digit '8',Digit '5'
],"Sunshine"),([Digit '6',Digit '5'],"Doll"),([Digit '6',Digit '8',Digit '5'],"Tomc
at"),([Digit '7',Digit '3',Digit '3'],"Pecan"),([Digit '7',Digit '7',Digit '7',Digi
t '4'],"Lulu"),([Digit '8',Digit '6',Digit '2',Digit '5'],"Tank"),([Digit '9'],"Bud
")]
```

3.3 autocomplete - 29 pts

This time, the goal is to autocomplete a given input with a list of possible suffixes and contact names (a bit like a search bar).

For example, if our phone book contains `212: Bill`, `222: Sarah`, `213: Tony`, then an autocompletion for `"21"` should result in `2: Bill`, `3: Tony`. Of course, an empty string should not be autocompleted (like an empty search bar!).

Once again, the signature and some example runs follow. All the examples are given on the `exampleTree` for extra clarity:

```
autocomplete :: String -> DigitTree -> [(PhoneNumber, String)]
```

```
*PE4> autocomplete "" exampleTree
[]
*PE4> autocomplete "asdfg" exampleTree
[]
*PE4> autocomplete "19" exampleTree
[([Digit '1'],"Marlow"),([Digit '2',Digit '3'],"Stewart")]
*PE4> autocomplete "77" exampleTree
[([],"Hughes")]
*PE4> autocomplete "7" exampleTree
[([Digit '2'],"Curry"),([Digit '7'],"Hughes")]
*PE4> autocomplete "13789" exampleTree
[]
*PE4> autocomplete "1378" exampleTree
[([],"Jones")]
*PE4> autocomplete "137" exampleTree
[([Digit '8'],"Jones")]
*PE4> autocomplete "13" exampleTree
[([Digit '7',Digit '8'],"Jones")]
*PE4> autocomplete "1" exampleTree
[([Digit '3',Digit '7',Digit '8'],"Jones"),([Digit '5'],"Steele"),([Digit '9',Digi
t '1'],"Marlow"),([Digit '9',Digit '2',Digit '3'],"Stewart")]
```

4 Regulations

1. **Implementation and Submission:** The template file named "PE4.hs" is available in the Virtual Programming Lab (VPL) activity called "PE4" on `OdtuClass`. At this point, you have two options:
   - You can download the template file, complete the implementation and test it with the given sample I/O on your local machine. Then submit the same file through this activity.
   - You can directly use the editor of VPL environment by using the auto-evaluation feature of this activity interactively. Saving the code is equivalent to submit a file.

   The second one is recommended. However, if you're more comfortable with working on your local machine, feel free to do it. Just make sure that your implementation can be compiled and tested in the VPL environment after you submit it.

   There is no limitation on online trials or submitted files through OdtuClass. The last one you submitted will be graded.

2. **Cheating: We have zero tolerance policy for cheating**. People involved in cheating (any kind of code sharing and codes taken from internet included) will be punished according to the university regulations.

3. **Evaluation:** Your program will be evaluated automatically using "black-box" technique so make sure to obey the specifications. No erroneous input will be used. Therefore, you don't have to consider the invalid expressions.

Important Note: The given sample I/O's are only to ease your debugging process and NOT official. Furthermore, it is not guaranteed that they cover all the cases of required functions. As a programmer, it is your responsibility to consider such extreme cases for the functions. Your implementations will be evaluated by the official testcases to determine your ***actual*** grade after the deadline.