EDAN40: Chatterbots (2015 version)

Summary

This assignment is your first exercise in working with a Haskell program which is somewhat larger than just toy examples. You are given a skeleton solution to the task and you are expected to add various missing pieces to it.

The purpose of this assignment is threefold:

1. Teach you **to read** and understand some already existing Haskell code;
2. Teach you to write fully **functioning** code: you won't get any deeper comments about your solution regarding the programming style before it fullfills the functional requirements;
3. Teach you the preferred style of writing Haskell code (you may wish to look up the expressions "**point-free** style", or "point-free programming" or have a look at [this blog post](http://buffered.io/posts/point-free-style-what-is-it-good-for/)).

The assignment should be completed **in pairs**.

Introduction

One of the most famous programs in the history of artificial intelligence is Joseph Weizenbaum's program Eliza. Written in the early sixties it emulates, or parodies, a Rogerian psychoanalyst who tries to make the patient talk about his or her deepest feelings by turning everything said into a question. Eliza's fame is due to the remarkable observation that many people found her seem so friendly and understanding. In fact some said she was the first analyst they had met who really listened.

Over the years Eliza has had many successors and the phenomenon she represents is now known as a *chatterbot*. A possible definition is:

A *chatterbot* is a program that attempts to simulate typed conversation, with the aim of at least temporarily fooling a human into thinking they are talking to another person.

There are some hints that Eliza has been modified for use in other contexts than scientific investigation, see e.g. here, renamed to [Analiza](http://motherboard.vice.com/read/the-cia-used-artificial-intelligence-to-interrogate-its-own-agents-in-the-80s). It seems that the author of this post has never heard of Eliza, though.

Today there are plenty of chatterbots on the web, from enhancement to commercial sites to research projects to mockup celebrities like [John Lennon](http://triumphpc.com/johnlennon/). You may wish to check the [chatbots.org](http://www.chatbots.org/) site. Due to the resemblance to the Turing test, chatterbots usually perform well in the quite prestigous [Loebner prize competition](http://www.loebner.net/Prizef/loebner-prize.html).

Sometimes they even [talk to each other](http://www.youtube.com/watch?v=WnzlbyTZsQY). There are even [ideas of using them for state matters](http://www.slate.com/articles/technology/future_tense/2015/10/using_chatbots_to_distract_isis_recruiters_on_social_media.single.html).

The Assignment

In this assignment you will implement a Chatterbot module which makes it possible to easily create systems of this kind.

import Chatterbot

main = chatterbot "Eliza" eliza

eliza = [

("I need \*",

["Why do you need \* ?",

"Would it really help you to get \* ?",

"Are you sure you need \* ?"]),

("Why don't you \*",

["Do you really think I don't \* ?",

"Perhaps eventually I will \* .",

"Do you really want me to \* ?"]),

{- ... and so on ... -} ]

The chatterbot is defined by a list of pattern-based rules. Each rule consists of two parts, a pattern and a list of templates. Input is matched against the pattern and if it matches, one of the templates is given as a response. Words in the input which match the wildcard '\*' are bound to the corresponding wildcard in the template. For example:

*You:* I need help  
*Eliza:* Are you sure you need help ?

You can take a look at the full main program [here](http://fileadmin.cs.lth.se/cs/Education/EDAN40/assignment1/Eliza.hs.txt).

1. Some useful functions

To create a good solution to a problem it is crucial to have a good collection of basic building blocks. To find such primitives your first choice should always be the standard prelude. The next choice would be the other [standard libraries](http://www.haskell.org/ghc/docs/7.4.1/html/libraries/). In this assignment we will for example make heavy use of the datatype Maybe (found in [Data.Maybe](http://hackage.haskell.org/package/base-4.2.0.1/docs/Data-Maybe.html)) including the associated function maybe. Make sure you understand their purpose and what they do.

Below are some other general functions which you will find useful in this assignment. Your first task is to study them carefully and figure out what they do so that you can use them later on. You might for example write a proper documentation comment for each one of them.

map2 :: (a -> b, c -> d) -> (a, c) -> (b, d)

map2 (f1, f2) (x1, x2) = (f1 x1, f2 x2)

mmap :: (a -> b) -> Maybe a -> Maybe b

mmap f Nothing = Nothing

mmap f (Just x) = Just (f x)

orElse :: Maybe a -> Maybe a -> Maybe a

orElse Nothing x = x

orElse (Just a) \_ = Just a

try :: (a -> Maybe a) -> a -> a

try f x = maybe x id (f x)

fix :: Eq a => (a -> a) -> a -> a

fix f x

| f x == x = x

| otherwise = fix f (f x)

pick :: RealFrac r => r -> [a] -> a

pick u xs = xs !! (floor.(u\*).fromIntegral.length) xs

2. Pattern matching

The basic machinery of chatterbots consists of *pattern matching*. Your first task is to write the following two functions:

substitute :: Eq a => a -> [a] -> [a] -> [a]

match :: Eq a => a -> [a] -> [a] -> Maybe [a]

The function match wildcard p s tries to match the two lists p and s. The list p is considered a pattern which may contain elements equal to wildcard. The list s may not contain any wildcards. The pattern p is said to match the list s if every element in p matches corresponding elements in s. A non-wildcard matches a single element in s if they are equal and a wildcard matches an arbitrarily long (non-empty) sublist. If the two lists match, the function returns Just r where r is the sublist in swhich matches the first occurence of wildcard in p. If the lists don't match, the result is Nothing. Here are some examples:

match 'x' "2\*x+3" "2\*7+3" = Just "7"

match '\*' "frodo" "gandalf" = Nothing

match 2 [1,3..5] [1,3..5] = Just []

match '\*' "\* and \*" "you and me" = Just "you"

match 'x' "2\*x+3+x" "2\*7+3" = Nothing

The function substitute wildcard t s replaces each occurrence of the element wildcard in the list t with the list s. For example:

substitute 'x' "3\*cos(x) + 4 - x" "5.37" =

"3\*cos(5.37) + 4 - 5.37"

Hints:

* As with most list functions you have to define different equations for the cases of empty and non-empty lists.
* The function substitute is the easier one of the two so you might want to write it first.
* The function match is a greater challenge. In fact, it is the key to the whole chatterbot solution. In the pattern matching there are three base cases, each of which should be fairly straightforward:
  + Both lists are empty.
  + The pattern list is empty but the other list is not.
  + The pattern list is not empty but the other list is.

In the equation for the case of two non-empty lists you have to consider the two cases where the first element of the pattern is a wildcard and when it is not:

* + If it is not a wildcard the two lists match if the first elements are equal and the rests of the lists match.
  + When the first element is a wildcard there are two possible ways to reach a match. Your implementation of this case should be written as a combination of the results of two auxilliary functions, singleWildcardMatch andlongerWildcardMatch, each of which implements either of the ways. The function singleWildcardMatch defines the case when the rest of the list matches with the rest of the pattern, i.e. the front wildcard removed. And the functionlongerWildcardMatch defines the case when rest of the list matches with the pattern with the wildcard retained at the front. Note that both the auxilliary functions need to use match in their definitions.

The example in the table below illustrates the difference between the three cases that can arise when the first element is a wildcard:

|  |  |  |  |
| --- | --- | --- | --- |
|  | "bdo" | "dobedo" | "bedobe" |
| singleWildcardMatch "\*do" | Just "b" | Nothing | Nothing |
| longerWildcardMatch "\*do" | Nothing | Just "dobe" | Nothing |
| match '\*' "\*do" | Just "b" | Just "dobe" | Nothing |

Finally, to obtain good definitions of match and its auxilliary functions it is essential that you use some of the utility functions given in section 1 above.

3. Pattern transformations

A *pattern transformation* is a pair of patterns, for example:

frenchPresentation = ("My name is \*", "Je m'appelle \*")

a) Write a function

transformationApply :: Eq a => a -> ([a] -> [a]) ->

[a] -> ([a], [a]) -> Maybe [a]

where *applying* a transformation to a list means that you match the list against the first pattern and then, if the match succeeds, substitute the result into the second list.

transformationApply '\*' id "My name is Zacharias"

frenchPresentation = Just "Je m'appelle Zacharias"

The second parameter to this function is another function which is used to transform the result of the match before the substitution is made. You will later see how this is useful.

b) Write a function which operates on a whole list of pattern transformations

transformationsApply :: Eq a => a -> ([a] -> [a]) ->

[([a], [a])] -> [a] -> Maybe [a]

This function uses transformationApply on the patterns in the list until one succeeds. The result of that transformation is then returned. If all patterns fail, the function returns Nothing. Note that the two last parameters to this function are given in opposite order, relative to the previous one.

4. Chatterbot implementation

The main function in the Chatterbot module is shown below. It is essentially an interactive loop which reads a line of input and then responds by writing a line of output.

chatterbot :: String -> [(String, [String])] -> IO ()

chatterbot botName botRules = do

putStrLn ("\n\nHi! I am " ++ botName

++ ". How are you?")

botloop

where

brain = rulesCompile botRules

botloop = do

putStr "\n: "

question <- getLine

answer <- stateOfMind brain

putStrLn (botName ++ ": "

++ (present . answer . prepare) question)

if (not . endOfDialog) question

then botloop

else return ()

The chatterBot function takes as arguments the name of the bot and a list of rules. The rules are converted from strings to an internal format and bound to the local variable brain. In each turn of the loop the function stateOfMind takes the brain and generates a random response function answer. The actual response is created by wrapping answer with functions which do suitable formatting of input and output. By convention we will do all work on lower case characters.

You will now implement the functions used by the main program. While doing so you will use the functions on patterns defined above. A chatterbot works on strings, but you will not use the pattern functions on strings directly. Instead you will apply them on lists of words, or *phrases*.

type Phrase = [String]

type PhrasePair = (Phrase, Phrase)

type BotBrain = [(Phrase, [Phrase])]

The prelude has functions which make the conversion between strings and phrases easy:

words :: String -> [String]

unwords :: [String] -> String

One of the keys to the success of Eliza is a simple technique known as *reflection*. Reflecting a phrase means that you replace each occurrence of a first person word or expression with the corresponding second person word or expression and vice versa.

a) Write a function

reflect :: Phrase -> Phrase

which tries to replace each word in a phrase with its corresponding word in the map reflections below.

reflections =

[ ("am", "are"),

("was", "were"),

("i", "you"),

("i'm", "you are"),

("i'd", "you would"),

("i've", "you have"),

("i'll", "you will"),

("my", "your"),

("me", "you"),

("are", "am"),

("you're", "i am"),

("you've", "i have"),

("you'll", "i will"),

("your", "my"),

("yours", "mine"),

("you", "me")

]

For example:

reflect ["i", "will", "never", "see", "my",

"reflection", "in", "your", "eyes"] =

["you", "will", "never", "see", "your",

"reflection", "in", "my", "eyes"]

Hint: There is a prelude function which is particularly useful here.

b) Use the function transformationsApply you implemented earlier to write the central chatterbot function

rulesApply :: [PhrasePair] -> Phrase -> Phrase

which transforms a phrase according to a list of pattern transformations, and where the intermediate result of the match is reflected. Note that the suitable wildcard element is "\*" rather than '\*' since we are working with phrases and not strings.

c) The rules for a chatterbot specify a number of possible responses for each pattern transformation and the actual response is chosen randomly. Write a function

stateOfMind :: BotBrain -> IO (Phrase -> Phrase)

which is based on the function rulesApply and generates a random response function from a BotBrain argument. For this purpose you need the standard library module System.Random. The following example shows how it can be used.

import System.Random

rollDice :: IO ()

rollDice = do

r <- randomIO :: IO Float

putStrLn ("You rolled " ++ show (floor (6\*r+1)))

Note that randomIO has a monadic return type. Why?

Finally, you need a function which determines the end condition for a dialog and three functions which convert between strings and phrases:

endOfDialog :: String -> Bool

prepare :: String -> Phrase

present :: Phrase -> String

rulesCompile :: [(String, [String])] -> BotBrain

The end condition can be fairly simple:

endOfDialog = (=="quit") . map toLower

To present an answer not much needs to be done:

present = unwords

In preparing the input however it is useful to use the standard library Data.Char to do some extra formatting. Make sure you understand the following definition.

prepare = words . map toLower .

filter (not . flip elem ".,:;\*!#%&|")

d) Write the function rulesCompile.

Now you have a complete bot. It is time to test it. Have a nice chat with Eliza!

5. Improvement

A difficulty in designing chatterbots is to avoid making the mimicking too rigid. It can easily destroy the illusion of a listening person.

If, for example, you say I am very very tired you may get the response How long have you been very very tired?, but better response would have been How long have you been tired?. A general approach which adds a lot of flexibility is to *reduce* the input by removing some syntactic variants of questions which are essentially the same.

reductions :: [PhrasePair]

reductions = (map.map2) (words, words)

[ ( "please \*", "\*" ),

( "can you \*", "\*" ),

( "could you \*", "\*" ),

( "tell me if you are \*", "are you \*" ),

( "tell me who \* is", "who is \*" ),

( "tell me what \* is", "what is \*" ),

( "do you know who \* is", "who is \*" ),

( "do you know what \* is", "what is \*" ),

( "are you very \*", "are you \*" ),

( "i am very \*", "i am \*" ),

( "hi \*", "hello \*")

]

Start by modifying the definition of prepare.

prepare = reduce . words . map toLower .

filter (not . flip elem ".,:;\*!#%&|")

and define the function reduce as

reduce :: Phrase -> Phrase

reduce = reductionsApply reductions

Write the function

reductionsApply :: [PhrasePair] -> Phrase -> Phrase

You can try the new implementation with this question:

"can you please tell me what Haskell is"

Provided material

The partial solution described above can be downloaded [here](http://fileadmin.cs.lth.se/cs/Education/EDAN40/assignment1/assignment1.zip). The archive contains four files:

* Utilities.hs
* Eliza.hs
* Chatterbot.hs
* ChatterbotTest.hs

The functions you need to write are marked {- TO BE WRITTEN -}. Dummy solutions are included in order to make the program type check although it is not finished.

The ChatterbotTest.hs will let you check the functional correctness of your solution: instead of Eliza.hs load it into the interpreter and see the outcome.

Submission

* **What to submit:**  
  Only the file Chatterbot.hs completed with your solutions to tasks 2, 3a-b, 4a-d and 5. To pass, your code must (1) work correctly (use ChatterbotTest.hs to verify this in advance, before submitting), and (2) be highly readable, written in point-free style.
* **How to submit:**  
  E-mail to [fp@cs.lth.se](mailto:fp@cs.lth.se) with the subject line of "Assignment 1 by *username1* and *username2*", where *username1* and*username2* are your logins (e.g. ada13jma). Replies will be sent to the addresses you have specified in your StIL account, so please make sure that forwarding works correctly.  
  In the mail you should state how many hours per person you have spent on this assignment.
* **Deadline:**  
  November 30, 2015.