Assignment N1: Chatterbots

Jacek Malec & Noric Couderc

Assignment N1: Chatterbots

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 (i.e., it works);
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).

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.

Today there are plenty of chatterbots on the web, from enhancement to commercial sites to research projects, mostly based on Large Language Models

technology. (Choose EDAP20 to find out more:-)

Nowadays it is, of course, GPT-4-like technology that rules, but it is amazing how much one can achieve with so little technology as in the following piece of code.

Getting Started

This Haskell project uses a build system called Stack. Stack can build haskell programs, but also takes care of downloading dependencies, etc.

- To compile the program, use the command `stack build`, the first time you do it, it might take some time.
- To get access to ghci with loaded modules, use `stack ghci`
- To run the tests, use `stack test`
- To run the chatbot, use `stack run`

If you want to run the tests every time you update a file, you can type `stack test --file-watch`, it also works for `stack build`.

Provided material

A ZIP file with the following contents:

- `app`: The code for the main module
  - `Main.hs`: Source for the main program
- `src`: The code for the main library of the program
  - `Chatterbot.hs`: The file you have to modify.
  - `Utilities.hs`: A file with some utility functions
- `test`: The test source files.
  - `ChatterbotTest.hs`: The source of the tests
- `README.md`: The instructions for the assignment.
- `stack.yaml`: The file containing our project-level settings.
- `package.yaml`: The package configuration settings (dependencies, etc.)
- `Setup.hs`: A file generated by stack.
- `n1.cabal`: A file generated by stack.
- `ChangeLog.md`: A file generated by stack.

You have to change the file `src/Chatterbot.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. fileckage

What to submit:

Only the file `Chatterbot.hs` with your names in the first (comment) line of the file, completed with your solutions. To pass, your code must:

1. Work correctly (use `stack test` to verify this in advance, before submitting), you are allowed to add tests, but we will test your solution against

the original tests.
2. Be highly readable, written reasonably in point-free style (use e.g., HLint for that purpose).

How to submit:

Use course Canvas for that purpose.

Deadline:

- You should submit your file on the 17/4/2025 at the latest.

The Assignment

In this assignment you will implement a Chatterbot module which makes it possible to easily create a chatterbot like Eliza.

```
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:

```
USER: I need some sleep
ELIZA: Are you sure you need some sleep?
```

Here's how the chatbot works:

1. We read input from the user, and use a `prepare` function to put text in lowercase, and remove punctuation.
2. We match the text from the user with a pattern, so we will make a `match` function.

3. Then, we'll take the match, and put in in the template. We'll make a `substitute` function for that.
4. We will then format this output and print it to the user, that's the `present` function.

We will see that using just `match` and `substitute` is not enough, so we will make a `transformationApply` function, this function will do a little bit more work between matching and substitution.

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, but the file `src/Utilities.hs` also contains some functions you might find useful.

Do not despair if you don't get their purpose immediately. Feel free to return to this point when you either encounter them in the provided code, or when it feels that types seem to match some particular need you just have. E.g., the function pick will be needed for implementing `stateOfMind` later on; probably only then you will easily appreciate its behaviour.

```
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
```

The file `src/Utilities.hs` contains these functions, and more.

Pattern matching

A pattern is a list of things, some of which might be wildcards, to represent that, we'll make some types.

```
-- An element, may be a wildcard!
data PatternElem a = Wildcard | Item a
  deriving (Eq, Show)

-- A list of pattern elements makes a pattern
newtype Pattern a = Pattern [PatternElem a]
  deriving (Eq, Show)

-- Templates are the same as patterns
type Template a = Pattern a
```

As a warm-up, **write a function** `mkPattern` which takes a wildcard and a list, and makes a pattern.

```
mkPattern :: Eq a => a -> [a] -> Pattern a

-- For example, treating a '*' as a wildcard
ghci> mkPattern '*' "Hi *!"
Pattern [Item 'H', Item 'i', Item ' ', Wildcard, Item '!']
```

Next, let's look at the following two functions, which form the core of the chatterbot.

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

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

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

```
ghci> substitute (mkPattern 'x' "3*cos(x) + 4 - x") "5.37"
"3*cos(5.37) + 4 - 5.37"
```

Write the `substitute` function.

Match

The function `match p s` tries to match the pattern `p` with the list `s`. It returns a `Maybe [a]` and which means there are three possible cases:

1. The pattern and the list can't match, so the function returns `Nothing`, indicating the match failed.
2. The pattern and the list match, in which case there are two cases:
    1. The pattern didn't contain a wildcard, in which case we extract nothing, the function returns `Just []`.

2. The pattern and the list match, with wildcards, so we extracted something from the list, in which case the result with have the shape `Just [...]`.

In the case where the patterns contains *several* wildcards, the match will only contain the first.

```
match (mkPattern '*' "frodo") "gandalf" = Nothing
match (mkPattern 'x' "2*x+3+x") "2*7+3" = Nothing
match (mkPattern 'x' "abcd") "abcd" = Just []
match (mkPattern 2 [1,3..5]) [1,3..5] = Just []
match (mkPattern 'x' "2*x+3") "2*7+3" = Just "7"
match (mkPattern '*' "* and *") "you and me" = Just "you"
```

Write the `match` function

Hints:

- Start by looking at the cases where either the pattern is empty, or the list is empty.
- Next, look at the case of patterns which do not contain *any* wildcard.
- Next, for the case where the pattern starts with a wildcard, we use two helper functions: `singleWildcardMatch` and `longerWildcardMatch`:
  - `singleWildCardMatch` works like this: First, we match the rest of the list with the pattern, if that worked, we return the current element, otherwise, `Nothing`.
  - `longerWildCardMatch` works the same, but:
    - it doesn't *consume* the first wildcard of the pattern.
    - it extracts the match, and prepends the current element of the list to it.
- What makes this exercise tricky is how the functions call each other: `match` calls both `singleWildcardMatch` and `longerWildcardMatch`, and both functions *also* call `match`!
- Finally, both `*WildcardMatch` functions can be one-liners, if you use some of the utility functions given in `Utilities.hs`.

Below is a table listing what the result should be for different cases.

	"bdo"	"dobedo"	"bedobe"
`singleWildcardMatch (mkPattern '*' "*do")`	Just "b"	Nothing	Nothing
`longerWildcardMatch (mkPattern '*' "*do")`	Nothing	Just "dobe"	Nothing
`match (mkPattern '*' "*do")`	Just "b"	Just "dobe"	Nothing

Pattern Transformations

Now that we can match a string with a pattern and substitute, we'll write a function to insert the match into a template.

A pattern transformation is a pair of a pattern and a template, for example:

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

Given a transformation and a string, the chatterbot will:

1. Match the pattern with the string
2. Transform the match using a function (you will see later why this is useful)
3. Substitute the match into the template.

So, for example, we can extract the name from the string, and put it in the French version (`id` is a special function which doesn't do anything):

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

transformationApply id "My name is Zacharias"
    frenchPresentation = Just "Je m'appelle Zacharias"
```

Write a function:

```
transformationApply :: Eq a =>
    ([a] -> [a]) -> -- Function to apply to the extracted data
    [a] -> -- The string to match
    (Pattern a, Template a) -- The transformation
    -> Maybe [a] -- The result
```

Now, given a transformation, we're not sure we'll successfully match the string with the first pattern. So we'd like to try many transformations in a row.

Write a function which operates on a whole list of pattern transformations:

```
transformationsApply :: Eq a =>
    ([a] -> [a]) -> -- Function to apply to extracted match
    [(Pattern a, Template a)] -> -- List of transformations
    [a] -> -- String to match
    Maybe [a] -- Result
```

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.

Reflections

One of the problems when making Eliza, is that if the user uses pronouns, we can't put them in templates directly.

```
USER: I don't like my job
ELIZA: Why do you say you don't like my job?
```

In the above example, we need to replace `my job` by `your job`, we call that "reflecting". Reflecting a phrase means that you replace each occurrence of a first person word or expression with the second person equivalent.

Before we do that, we'd like to match with *words*, while so far, we've matched with *letters*. We will make a type `Phrase` which is a list of words.

```
type Phrase = [String]
```

And we can convert between `String`s and `Phrase`s with functions that we have in the Prelude.

```
words   :: String -> [String]
unwords :: [String] -> String
```

Now, **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.

Note: This function doesn't use patterns, but instead replaces one word, using the list of reflections, so there are some sentences it doesn't handle well, especially

when dealing with "you": `i love you` becomes `you love me`, but `you are my best friend` becomes `me am your best friend`. Because in one case, `you -> me` and in the other, it should be `you -> i`. However, it is used *after* we have matched, so such a simple function might be enough for our uses.

Applying Rules

Now, when reflections work, you can write an important function of the chatterbot: `rulesApply`.

```
rulesApply :: [(Pattern String, Template String)] -> Phrase -> Phrase
```

`rulesApply` transforms a phrase, like so:

1. Match the phrase with a pattern.
2. Reflect the match.
3. Substitute the match in the target pattern.

Use the function `transformationsApply` you implemented earlier to write this function.

Randomization

We could stop here, but Eliza would be very mechanical: If you typed the same sentence twice, she would give the same answer twice. That's because rules apply the same template every time.

Instead, we'll make a bot brain, which is a list of rules, where each rule is a pattern with several templates, and we'll choose a template each time Eliza needs to answer.

```
newtype Rule = Rule (Pattern String, [Template String])
  deriving (Eq, Show)

type BotBrain = [Rule]
```

The main module has a list of strings, though, we need a function to convert from a tuple of strings to a rule.

Write a function:

```
ruleCompile :: (String, [String]) -> Rule
```

You may use the functions `stringToPattern` and `starPattern`, but remember that we should make the text in the patterns lower case.

Next, we need a function which would match a phrase with a pattern, and select a template at random.

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

This function takes a `BotBrain` and returns a function that takes the input text from the user, and returns the answer from the bot, but wrapped in `IO`, why is that?

In the course, we explain that Haskell pushes the developer to write *pure* functions: Functions which return the same result when given the same arguments, and have no side-effects. However, a function which produces random numbers is pretty much the opposite of a pure function: The entire point is to return different results each time the function is called. In Haskell, we can have functions that return random numbers, but they have side-effects, and Haskell reflects that in the return type, which *must* be wrapped in `IO`.

Here is an example, where we use the module `System.Random` from the standard library:

```
import System.Random

rollDice :: IO ()
rollDice = do
   r <- randomIO :: IO Float
   putStrLn ("You rolled " ++ show (floor (6*r+1)))
```

Because the function `randomIO` has side-effects to compute its result, the function that calls it also must return a type wrapped in `IO`. This represents the idea that side-effects spread to the caller: If a function calls a function which has side-effects, it has side-effects too.

Write the function:

```
makePair :: Rule -> IO (Pattern String, Template String)
```

Which takes a rule and produces a tuple of a pattern with a template, and uses `randomIO` to pick a random template. The function `pick` from `Utilities.hs` might be useful.

Reductions

If the user isn't too careful with their words, the chatbot may sound very mechanical:

```
USER: I am very very tired
ELIZA: How long have you been very very tired?
```

A better response would be `How long have you been tired`?

To make the bot more believable, we will to reduce the *input* from the user, before matching.

```
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 *")
  ]
```

The function `prepare` reads the input from the user, and reduces it.

```
prepare = reduce . words . map toLower .
          filter (not . flip elem ".,:;*!#%&|")
```

`reduce` is defined as:

```
reduce :: Phrase -> Phrase
reduce = reductionsApply reductions
```

Now, **write the function:**

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

One key aspect of this function is that it will keep applying reductions until it can't remove anything from the string anymore. The function `fix` in `Utilities.hs` should be useful here.

You can try the new implementation with this question:

```
"can you please tell me what Haskell is"
```

Finishing up

Make sure the tests pass with `stack test`.

Once you have the tests working, you can use `stack run` to talk with Eliza!