CSE 331: Software Design & Implementation

Homework Chatbot (due Wednesday, July 31st at 11:00 PM)

The problems in this assignment contain a mix of written and coding parts. When you have completed them, submit your solutions in Gradescope. The written problems should be clearly labeled and submitted to the "HW Chatbot Written", and the following completed files should be directly submitted for the "HW Chatbot Coding" assignment:

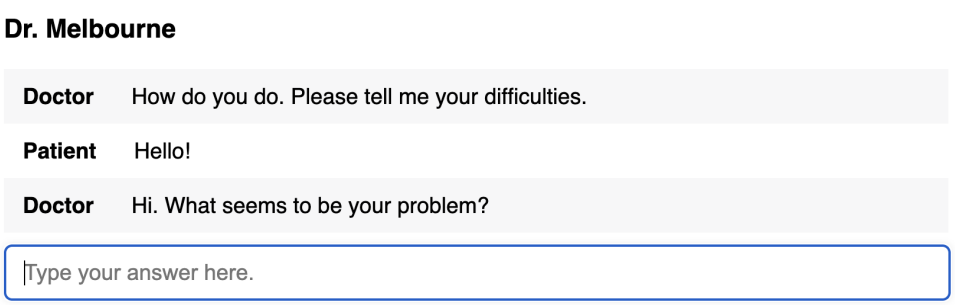
index.ts	routes.ts	chatbot.ts
words.ts	routes_test.ts	

Start by checking out the starter code using the command:

```
git clone https://gitlab.cs.washington.edu/cse331-24su/cse331-24su-materials/hw-chatbot.git
```

Then, install the modules using `npm install`. (The `--no-audit` is not necessary with server code.)

For this assignment, you will be implementing and reasoning about some helper functions related to manipulating and parsing strings. These will be used for a chatbot app where you can converse with "Dr. Melbourne".



You're only required to make changes in the above files, but the rest of the application is pretty interesting, so feel free to look around the files to understand how it is working!

Note that throughout this assignment there are integer values represented with `number` instead of `bigint`. The reason for this is that we are using existing Array and String functions which expect inputs to be of type `number`. You don't need to be concerned about this for any of the code you write, but if you do see a `number` representing a value that must be an integer, this is why!

In Homework Weave, we saw that sets are easily represented using standard inductive data types, such as lists containing no duplicates. The same is also true of maps. A map stores and supports operations on key-value pairs, and can be easily represented as a list of key-value pairs. For example, the list

$$\mathsf{cons}((\text{“a”}, \text{“apple”}), \mathsf{cons}((\text{“o”}, \text{“orange”}), \mathsf{cons}((\text{“p”}, \text{“pear”}), \mathsf{nil})))$$

represents a mapping that takes the key “a” to the value “apple”, the key “o” to the value “orange”, and the key “p” to the value “pear”.

In functional programming, lists of key-value pairs are usually referred to as “**association lists**”, and they are every bit as common as `Map`s in Java. When writing mathematical specifications, it is normal to work with immutable data types, so the most sensible way for us to define maps mathematically is as association lists.

Since association lists are just lists of pairs, we can build them using “cons”. However, in order to get information from them, it will be useful to define a couple of helper functions.

The function contains-key checks if there is some value associated with the key. It is defined as follows:

func	$\mathsf{contains\text{-}key}(x, \mathsf{nil})$	$:=$	false		for any $x : \mathbb{S}^*$
	$\mathsf{contains\text{-}key}(x, \mathsf{cons}((y, v), L))$	$:=$	true	if $x = y$	for any $x, y, v : \mathbb{S}^*$ and L : List
	$\mathsf{contains\text{-}key}(x, \mathsf{cons}((y, v), L))$	$:=$	$\mathsf{contains\text{-}key}(x, L)$	if $x \neq y$	for any $x, y, v : \mathbb{S}^*$ and L : List

The function get-value returns the value associated with the given key. It is defined, when there is some value associated with the key, as follows:

func	$\mathsf{get\text{-}value}(x, \mathsf{nil})$	$:=$	undefined		for any $x : \mathbb{S}^*$
	$\mathsf{get\text{-}value}(x, \mathsf{cons}((y, v), L))$	$:=$	v	if $x = y$	for any $x, y, v : \mathbb{S}^*$ and L : List
	$\mathsf{get\text{-}value}(x, \mathsf{cons}((y, v), L))$	$:=$	$\mathsf{get\text{-}value}(x, L)$	if $x \neq y$	for any $x, y, v : \mathbb{S}^*$ and L : List

In `assoc.ts`, we have have provided an `AssocList` type and “straight from the spec” implementations of the above helper functions. The `AssocList` has `string` keys and uses a generic parameter “`V`” for its values.

1. Loop Du Jour (20 points)

The following parts consist entirely of coding work. They should be submitted with "HW Chatbot Coding".

In this problem, you will implement a function that substitutes some words for others in an array of words (strings). We can formally define the substitution operation as "substitute(A, M)", where A is an array of words, M is an association list with pairs of key words and (replacement) value words, and substitute is defined as follows:

$$
\begin{array}{lll}
\textbf{func}\ \text{substitute}([], M) & := & [] \\
\text{substitute}(A + [w], M) & := & \text{substitute}(A, M) + [\text{get-value}(w, M)] \quad \text{if contains-key}(w, M) \\
\text{substitute}(A + [w], M) & := & \text{substitute}(A, M) + [w] \quad \text{if !contains-key}(w, M)
\end{array}
$$

(a) Write a JSDoc specification for the function `substitute` in `words.ts`.

A function declaration is already provided. It looks like this:

```
export const substitute = (words: string[], reps: AssocList<string>): void => {...}
```

Your specification should explain that `substitute` will **mutate** the array "`words`" so that its value, after the call completes, is substitute(words, reps), where substitute is the above mathematical definition.

(b) Implement `substitute` in `words.ts` using a loop.

The local variable "`j`" should be used as an index into the array "`words`". The invariant should be:

$$\text{words} = \text{substitute}(\text{words}_0[0\ ..\ j-1], \text{reps}) + \text{words}_0[j\ ..\ n-1]$$

where $n = $ words.length. **Your code must be correct with this invariant**, which claims that we have already substituted in the first j elements of words, but not the $n - j$ remaining elements.

There is one example test given for `substitute` in `words_test.ts`. Notice how the test creates an array, calls `substitute` on the array (which mutates it), and then compares the result to the expected value. It is a good idea to write additional tests to be confident that your code is correct or aid in debugging. However, you will not turn in these tests, so we will not be grading them.

2. Loops, I Did It Again (20 points)

The following parts consist entirely of written work. They should be submitted with "HW Chatbot Written".

In the previous problem, you wrote a function that replaced some words in an array with other words. That version was able to perform the changes in-place because each word was replaced by another word. In this problem and the next, we will prove the correctness of code that replaces individual words with *one or more words*. This version cannot be easily done in-place, so it will return a new array with the result.

The code is split into two parts. The first part performs the replacements, producing an array of arrays; we'll prove the correctness of this in this problem. The second step is concatenating all of the arrays into a single array. (Note: We won't require you to complete the proof of correctness for the second step, but you can in the last problem for Extra Credit!)

The first part of the function calculates "replace(A, M)", where A is an array of words, M is an association list of pairs of words and arrays of words, and replace is defined as follows:

$$
\begin{array}{lll}
\textbf{func}\ \text{replace}([], M) & := & [] \\
\quad \text{replace}(A + [w], M) & := & \text{replace}(A, M) + [\text{get-value}(w, M)] \quad \text{if contains-key}(w, M) \\
\quad \text{replace}(A + [w], M) & := & \text{replace}(A, M) + [[w]] \quad \text{if !contains-key}(w, M)
\end{array}
$$

The only difference between this definition and that of "substitute" is the last line, where we replace "$[w]$" with "$[[w]]$". Here, we are building an array of arrays. A word w that is a key in M is replaced by get-value(w, M), which will be an array of words, and a word w that is not in a key in M is replaced by the 1-element array $[w]$.

Now, consider the following code, which claims to calculate replace(A, M) in the array R:

```
let R: string[][] = [];
let i: number = 0;
{{ P1 : _____ }}
{{ Inv: R = replace(A[0 .. i − 1], M) }}
while (i !== A.length) {
   if (contains_key(A[i], M)) {
      const val = get_value(A[i], M);
     {{ _____ }}
      R.push(val);
     {{ P2 : _____ }}
   } else {
     {{ _____ }}
      R.push([A[i]]);
     {{ P3 : _____ }}
   }
  {{ _____ or _____ }}
  {{ Q : _____ }}
   i = i + 1;
  {{ _____ }}
}
{{ P4 : _____ }}
{{ Post : R = replace(A, M) }}
```

(a) Fill in the assertions for the code above.

You should use **forward** reasoning to fill out P_1 before the loop, P_4 after the loop, and the assertions within and immediately after the conditional. You should use **backward** reasoning to fill in the last assertion within the loop body and Q. (If you choose to reason through the code differently, you may need to place your assertions differently than outlined above. Additionally, the proofs required for part (b) will be different. This is *not* recommended.)

Like usual, you **should not use subscripts** for this problem. Instead, write all assertions in terms of the current values of variables.

Ensure that you write your assertions with mathematical definitions and notations rather than code notation. For example, instead of referring to A.length, we'll denote it as 'n'. If you want to repeat the exact loop invariant or one of the $P_{\#}$ assertions in other assertions, feel free to just write 'Inv' or $P_{\#}$. However, if any part of the assertion changes, you should rewrite it.

Note that `.push()` is appending onto an array. If we append a value z to array Z, that is equivalent to concatenating it (i.e. $Z + [z]$). In this code, we use the TS functions implemented from the definitions on the first page. You can use the math definitions of these functions in your assertions and proofs as if the definitions apply for values *of any type* in (key, value) pairs.

(b) Prove that P_1 implies Inv, P_4 implies Post, and that the assertion just below the conditional implies Q (by cases).

Performing the loop from this problem followed by the one in the last problem calculates concat(replace(L, M)), which is the one-to-many word replacement operation. It may be useful to glance at the loop in the last problem to be convinced that this makes sense!

3. At My Splits End (20 points)

The following problem consists entirely of coding work. It should be submitted with "HW Chatbot Coding".

In this problem, you will implement a function that splits a string into its words, with each punctuation character as its own word and all white space removed entirely. Some examples:

- the string "if this, then that" becomes the array of strings ["if", "this", ",", "then", "that"]
- the string "well, well, well..." becomes the array of strings ["well", ",", "well", ",", "well", ".", ".", "."]
- the string "hello goodbye " becomes the array of strings ["hello", "goodbye"]

From the English description, we ordinarily write a formalized math definition, but the recursive definition in this case would be pretty confusing. Instead, take the following declarative description of what it means for an array of words A to be a correct splitting of a string s. These are the conditions we want:

- If you concatenate the words of A, you get back the original string s, but with the white space excluded. (Equivalently: "join(A) = del-spaces(s)", where join is a TypeScript function that creates a string by concatenating together every value in the array separated by a space and del-spaces removes all spaces from a string as described in section.)
- Words that were separated by spaces in the original string and punctuation should be recognized as different words in A. This means that for $s =$ "if this, then that", "ifthis" is not a valid element in A, instead A needs to contain "if" and "this" separately.

To formalize splitting into words and removing spaces, we'll mark the locations where the string should be split. In our first example, the string "if this, then that" has the splits "|if| |this|,| |then| |that|", where each "|" indicates a split. Note that individual punctuation characters and spaces are their own piece and all *adjacent* letters are the same piece.

In the code, the split locations will be stored in an array `splits`, with each element in the array being an index of the character immediately following a split. It will always be the case that the first index is 0 and the last is the length of the string. Some examples:

- for the string "if this, then that", the `splits` array would store $[0, 2, 3, 7, 8, 9, 13, 14, 18]$
- for the string "well, well, well...", the `splits` array would store $[0, 4, 5, 6, 10, 11, 12, 16, 17, 18, 19]$
- for the string "hello goodbye ", the `splits` array would store $[0, 5, 6, 7, 8, 9, 16, 17, 18, 19, 20]$ (white space is 4 spaces in length)

If i and j are two adjacent elements in the splits array, then $s[i \,..\, j-1]$ is one of the pieces. The adjacent elements can create a piece that is a word, a space, or a punctuation character. For example, for "|if| |this|,| |then| |that|":

- $s[0 \,..\, 2-1] = s[0 \,..\, 1] =$ "if" (a word)
- $s[9 \,..\, 13-1] = s[9 \,..\, 12] =$ "then" (a word)
- $s[13 \,..\, 14-1] = s[13 \,..\, 13] =$ " " (white space between words)

Our implementation of `splitWords` will consist of two loops. The first loop will find the locations for all the splits. It will ensure that spaces and punctuation are split from the characters around them, while adjacent

letters are not split. The second loop, which we have provided, will produce the array of words by retrieving the substrings of the original string indicated by splits except for the spaces, which will be skipped.

Implement the first loop of `splitWords` in `words.ts`. The invariant is already provided. It references a variable "`j`", which keeps track of how much of the input string "`str`" we have processed so far. The invariant has three conditions:

1. "$0 = \text{splits}[0] < \text{splits}[1] < \cdots < \text{splits}[n-1] = j$, where $n = \text{splits.length}$."
   This says that the first split is just before the first character, the last split is just after the last character we've processed so far, and each split is after the previous one (meaning each split contains *at least* one character).

2. "For each $i = 0$ .. $n-1$, if $\text{splits}[i] + 1 < \text{splits}[i+1]$, then $\text{str}[\text{splits}[i]$ .. $\text{splits}[i+1] - 1]$ is all letters."
   This says that spaces and punctuation must be their own piece, pieces with more than one character must contain only letters.

3. "For each $i = 1$ .. $n-2$, $\text{splits}[i]$ is not between two letters."
   This says that adjacent letters in the original string should not be split, they should remain in the same piece in finalized splits. (Hint: "finalized" excludes the last split, which may be "in-progress" during the loop iterations).

You must implement the loop so that this invariant is preserved each time through the loop body. To make this easier, we have provided a function `CheckInv1` that should be called before the loop begins and at the end of the loop body so that it executes prior to each loop iteration. It will double check that these conditions are satisfied and throw an `Error` if not.

Here are some hints that can help with this problem:

- If you need more examples than those included in the spec, check out the provided test cases for some other examples.
- It's easiest if your loop places *at most* one split on each iteration. Consider how you can initialize your variables so this can be a consistent pattern.
- In your loop, you need to consider whether the character you're looking at is either a letter, space, or punctuation symbol. For each option, think about if you need to create a new split *or* extend the piece you're currently forming in the last split. For the latter case, it may be useful to update the value of the last split as you extend the piece.

After you've completed the first loop, take a look at the second loop that was provided for you and make sure you understand what it's doing. This loop maintains the given invariant which guarantees that `words` contains all of the characters from str that are not spaces up to the current value of the counter i, and that `words` doesn't contain any spaces. Just like in the first loop, the second loop uses a function, `CheckInv2`, to make sure the invariant is maintained.

Verify that the provided tests for `splitWords` now pass by running `npm run test`.

4. 10-Minute Power Map (20 points)

The following parts consists entirely of coding work. They should be submitted with "HW Chatbot Coding".

To finish our application, we will need additional operations that allow us to *mutate* a map, so our original `AssocList` is not sufficient. We will take advantage of TypeScript's built-in Map data structure, which provides the additional functionalities to set a (key, value) pair in a map and clear its contents (which we will only use in testing).

Prior to completing the remainder of this homework, you will need to read some documentation to learn more about the available Map methods. Specifically, you should read the documentation for the methods that allow you to make the following operations:

- Create an empty map.
- Check if there is some value associated with a given key in the map.
- Get the value associated with a given key, if such a pair exists in the map.
- Set a value for a given key in the map, replacing the current value if a pair with the given key already exists.
- Clear all pairs from the map. (This will only be used for testing.)

(a) We use a global variable, `LastUsed`, to map words inputted by users to the last response used by the chatbot for that word to avoid reusing the same responses. Currently, this is implemented with our `AssocList`, however, `LastUsed` must be mutated which makes it better suited to be implemented with a mutable map.

Update `LastUsed` to be a `const` variable with the type map and initialize it to an empty map. Then, update all usages of `LastUsed` in `chatbot.ts` to use the methods of the TS map. Each change that needs to be made is marked with a "TODO" comment, you do not need to update this file elsewhere, nor update other parts of the starter code for this step.

(b) Confirm that all given tests, besides those from `routes_test.ts` now pass by running `npm run test`.

(c) Start the server by running `npm run start`. You may need to comment out the parameters of the `load` function in `routes.ts` temporarily so this will run.

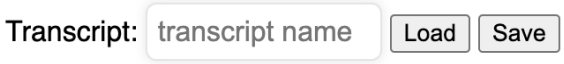
Verify that the app now works by opening up `http://localhost:8080` in your browser. That page should allow you to chat with Dr. Melbourne.

If you open up the "Network" tab in the Chrome developer tools, you should be able to see each request sent from the browser to the server and the server's response.

5. Here Is My Handle, Here Is My Route (20 points)

The following parts consist entirely of coding work. They should be submitted with "HW Chatbot Coding".

The application is running, but it is not yet fully functional. At the bottom of the page, there are buttons that aim to allow users to save and later re-load the chat transcript on the server. In this problem, we will make those buttons work by implementing operations to `save` and `load` transcripts.



We will store the transcripts in the server using a map, which you read about in the last problem. At the top of `routes.ts`, declare a `const` variable to hold the transcripts and initialize it to an empty map.

(a) Fill in the missing part of the `save` function in `routes.ts` so it stores the value sent by the client under the given name. The function should respond to the client with a record containing a field called "replaced" indicating with a boolean whether the value save replaced a previously saved transcript.

With those changes made, the provided tests for the "`save`" function should now pass.

(b) Implement the body of the `load` function to find the transcript with a given name. This will be a GET request, like the `chat` function, so it will receive arguments as **query** parameters. (You will want to use the provided "`first`" function to convert the value of query parameter into either a string or undefined. See the `chat` function for an example of how it is used.)

Your function should return a 400 status if the user did not provide a "name" query parameter or a 404 status if there was no transcript previously saved with that name. In both cases, also send back a string error message.

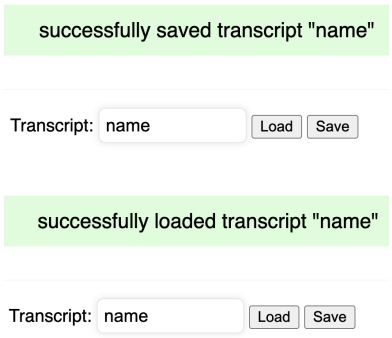
If there is a previously saved transcript with that name, send it back to the client inside of the "`value`" field of a record. We always send back records to the client. Here, the record we send will have a single field called "`value`" whose value is the transcript they requested.

(c) Write tests for your `load` function in `routes_test.ts`. There is an example test provided and some tips to get you started. Make sure that you implement `resetTranscriptsForTesting` and use it in your tests! When you are done, confirm that all your tests pass by running `npm run test`.

(d) Add routes for `/save` and `/load` in `index.ts`. While the load function expects GET requests, the save function expects POST requests.

(Re-)start the server. Open a browser window and confirm that you can save the transcript after chatting for a bit. Then, open a new window and confirm that you can load the saved transcript.

Note that if you stop and restart the server, it is expected that saved history will be lost.



Congratulations on competing another application!

6. Extra Credit: Jumping Through Loops (0 points)

The following parts consist entirely of written work. They should be submitted with "HW Chatbot Written".

In this problem, we continue the work started in problem 2 of checking the correctness of a function that replaces individual words in an array with *one or more words*. The first step was performing the word replacement but transformed the initial array A into an array of arrays R, which was completed in problem 2. The second step, is concatenate those arrays together into a final array.

The second part of the function calculates "concat(R)", where concat is defined as follows:

$$\begin{array}{lll} \textbf{func } \mathsf{concat}([]) & := & [] \\ \quad \mathsf{concat}(R \mathbin{+\!\!+} [[]]) & := & \mathsf{concat}(R) \\ \quad \mathsf{concat}(R \mathbin{+\!\!+} [A \mathbin{+\!\!+} [w]]) & := & \mathsf{concat}(R \mathbin{+\!\!+} [A]) \mathbin{+\!\!+} [w] \end{array}$$

To understand this definition a little better, let's start by proving the following:

(a) Let R be any array of arrays. Prove, by structural induction, that $\mathsf{concat}(R \mathbin{+\!\!+} [A]) = \mathsf{concat}(R) \mathbin{+\!\!+} A$ for any array A.

From the facts that $\mathsf{concat}([]) = []$ and $\mathsf{concat}(R \mathbin{+\!\!+} [A]) = \mathsf{concat}(R) \mathbin{+\!\!+} A$, we can (intuitively) see that

$$\begin{aligned} \mathsf{concat}([[A_1],\ \ldots\ [A_{n-1}],\ [A_n]]) &= \mathsf{concat}([[A_1],\ \ldots\ [A_{n-1}]]) \mathbin{+\!\!+} A_n \\ &= \ldots \\ &= \mathsf{concat}([]) \mathbin{+\!\!+} A_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} A_{n-1} \mathbin{+\!\!+} A_n \\ &= A_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} A_{n-1} \mathbin{+\!\!+} A_n \end{aligned}$$

where 'n' is the length of R. **In other words, concat(R) is just the concatenation of all the arrays in R into a single array.**

Now, consider the following code, which claims to calculate concat(R) in the array S:

```
let S: string[] = [];
let j: number = 0;
{{ P1 : _____ }}
{{ Inv1: S = concat(R[0 .. j − 1]) }}
while (j !== R.length) {
   const A: string[] = R[j];
   let k:  number = 0;
  {{ P2 : _____ }}
  {{ Inv2: S = concat(R[0 .. j − 1]) ++ A[0 .. k − 1] }}
   while (k !== A.length) {
     {{ _____ }}
      S.push(A[k]);
     {{ _____ }}
      k = k + 1;
     {{ P3 : _____ }}
   }
  {{ P4 : _____ }}
  {{ Q : _____ }}
   j = j + 1;
}
{{ P5 : _____ }}
{{ Post: S = concat(R) }}
```

The invariant of the outer loop says that S stores the result of applying concat to the first j elements of R. The invariant of the inner loop says that S stores that followed by the first k elements of A (which is $R[j]$). When we exit the inner loop, S will have all of $R[j]$ added to its end.

(b) Use **backward** reasoning to fill in Q from what we know just before exiting of the outer loop. Use **forward** reasoning to fill in all other assertions.

Write your assertions with mathematical definitions and notations rather than code notation. For example, instead of referring to R.length, we'll denote it as 'n', and instead of referring to A.length, we'll denote it as 'm'. If you want to repeat the exact loop invariant in other assertions, feel free to just write 'Inv.' However, if any part of the invariant changes you should rewrite it.

(c) Prove that P_1 implies Inv$_1$, P_2 implies Inv$_2$, P_3 implies Inv$_2$, P_4 implies Q, and P_5 implies Post.

Performing the loop from problem 2 followed by this one thus calculates concat(replace(L, M)), which is the one-to-many word replacement operation.