CSE 331: Software Design & Implementation

Homework 7 (due Wednesday, November 15th 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 as a pdf to the "HW7 Written" assignment. The following completed files should be directly submitted for the coding portion to the "HW7 Coding" assignment:

index.ts routes.ts routes_test.ts words.ts words_test.ts

For complete instructions for how to submit assignments in this course see the Homework Turn-in Guide.

Remember the rules for reasoning listed in the last assignment. **Unless explicitly allowed** in the problem statement, you **should not use subscripts** in assertions to refer to prior values (instead write the assertions in terms of current values of variables), and you **should not use backward reasoning**.

1. Loop Du Jour (20 points)

The following parts consist entirely of coding work. They should be submitted with "HW7 Coding".

Start by checking out the starter code using the command

```
git clone https://gitlab.cs.washington.edu/cse331-23au-materials/hw-chatbot.git
```

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

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 a map from key words to (replacement) value words, and substitute is defined as follows:

$$
\begin{array}{lll}
\textbf{func}\ \text{substitute}([], M) & := & [] \\
\quad \text{substitute}(A + [w], M) & := & \text{substitute}(A, M) + [M[w]] \quad \text{if } w \text{ is in } M \\
\quad \text{substitute}(A + [w], M) & := & \text{substitute}(A, M) + [w] \quad \text{if } w \text{ is not in } M
\end{array}
$$

The notation "$M[w]$" means to look up the value corresponding to the key w in the map M. In TypeScript code, this would be `M.get(w)`, which returns undefined when w is not a key in M.

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

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

```
export const substitute = (words: string[], reps: Map<string, 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), with the mathematical definition above.

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

Declare a local variable "`j`" that is an index into the array "`words`". Your invariant should be that

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

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

(c) Write tests for `substitute` in `words_test.ts`.

Follow the rules taught in lecture for choosing your test cases. You don't have to justify your choices in comments this time.

You may find tests for functions that mutate an input, rather than return a new value, may take a few more lines of code to set up than tests we've previously written.

2. Loops, I Did It Again (8 points)

The following parts consist entirely of written work. They should be submitted with "HW7 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, and the second part concatenates all the arrays into a single array. We will prove the correctness of the first part in this problem and the second part in the next problem.

The first part of the function calculates "replace(A, M)", where A is an array of words and M is a map from words to arrays of words and replace is defined as follows:

$$\begin{array}{lrll}
\textbf{func}\ \text{replace}([], M) & := & [] & \\
\text{replace}(A + [w], M) & := & \text{replace}(A, M) + [M[w]] & \text{if } w \text{ is in } M \\
\text{replace}(A + [w], M) & := & \text{replace}(A, M) + [[w]] & \text{if } w \text{ is not in } M
\end{array}$$

The only difference between this definition and that of "substitute" is on the last line, where we replace "$[w]$" with "$[[w]]$". In this case, we are building an array of arrays. A word w that is key in M is replaced by $M[w]$, 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) {
   const val = M.get(A[i]);
   if (val !== undefined) {
     {{ _____ }}
      R.push(val);
     {{ P2 : _____ }}
   } else {
                                          {{ _____ }}
      R.push([A[i]]);
                                          {{ P3 : _____ }}
   }
  {{ Q : _____ }}
   i = i + 1;
}
{{ P4 : _____ }}
{{ Post : R = replace(A, M) }}
```

(a) Use **forward** reasoning to fill in P_1, before the loop; P_2 and P_3 (the postconditions of the two branches) inside the loop; and P_4, after we exit the loop. Use **backward** reasoning up from just before exiting the loop to fill in Q.

Note that `.push()` is appending onto an array. If we append a value z to array Z, that is equivalent to concatenating as so: $Z + [z]$.

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 in other assertions, feel free to just write 'Inv.' However, if any part of the invariant changes you should rewrite it.

(b) Prove that P_1 implies Inv, P_2 implies Q, P_3 implies Q, and P_4 implies the postcondition.

Think carefully about what the lines of code tell you about the string you're looking at on each iteration and how it relates to the map M. As a hint, remember that `.get()` indicates that a key is not present in the map by returning `undefined`. With this information, what does entering into each branch tell us?

3. Jumping Through Loops (12 points)

The following parts consist entirely of written work. They should be submitted with "HW7 Written".

In this problem, we continue the work started in the previous problem on checking the correctness of a function that replaces individual words in an array with one or more words. The first part performed the word replacement but transformed the initial array A into an array of arrays R. In the second part, we will concatenate those arrays together into a second array.

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

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

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 $\text{concat}(R + [A]) = \text{concat}(R) + A$ for any array A.

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

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

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 **forward** reasoning to fill in P_1, before the outer loop; P_2, before the inner loop; P_3, at the end of the inner loop; P_4, after we exit the inner loop; and P_5, after we exit the outer loop. Use **backward** reasoning to fill in Q from what we know just before exiting of the outer loop.

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.

Advice: There are nested loops in this problem which is different than the examples we've seen before. We recommend that you try to reason through the loops independently. Ignore the outer loop and use Inv_2 as your base fact to reason through the inner loop (including P_4 immediately after it), then ignore the inner loop and use Inv_1 as your base fact to reason through the rest of the outer loop.

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

4. Two Sides of the Same Join (20 points)

The following parts consist entirely of coding work. They should be submitted with "HW7 Coding".

In this problem, you will implement a function that joins multiple words together into a single string. We can formally define this operation as "join-words(A)", where join-words is defined as follows:

$$
\begin{array}{lcll}
\textbf{func}\ \text{join-words}([]) & := & \text{“”} & \\
\quad \text{join-words}([w]) & := & w & \\
\quad \text{join-words}(A + [w]) & := & \text{join-words}(A) + w & \text{if } w \text{ is punctuation} \\
\quad \text{join-words}(A + [w]) & := & \text{join-words}(A) + \text{“ ”} + w & \text{if } w \text{ not is punctuation}
\end{array}
$$

In other words, this concatenates the words in the array but adds a space before each word to separate it from prior words and punctuation.

The `Array` class in JavaScript already includes a function called `join` that will turn an array of strings into a single string by concatenating them together (with a string of your choice in between adjacent elements). Unfortunately, that is not enough to implement the function above, which has different behavior based on whether the next element in the array is punctuation or not. Instead, we will need to implement this ourselves.

(a) Add code to the body of `joinWords` in `words.ts` to return the correct answer when "`words`" is empty.

That leaves us only with the case when `words` has at least one element.

(b) Implement the rest of `joinWords` with a loop.

Declare a local variable "`j`" that is an index into the array "`words`" and a variable "`parts`" that is an array of strings that will be concatenated at the end (with nothing extra in between those strings).

Your invariant should be that join(parts) = join-words(words$[0 \,..\, j-1]$), where "join" is the built-in function that simply concatenates the strings in "parts". (Note that, in Typescript, this operation would be performed by `parts.join("")`.)

`words.ts` contains a function `isPunct` that you should use to determine if a particular character is punctuation. To append to the end of an array, you can use `.push()`.

(c) Verify that the provided tests now pass by running `npm run test`.

5. At My Splits End (20 points)

The following parts consist entirely of coding work. They should be submitted with "HW7 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. For example, the string "if this, then that" would become the array of strings ["if", "this", ",", "then", "that"].

From that English description, we would ordinarily write a formalized math definition, but in this case a recursive definition would be pretty confusing, so we won't make you do that :). 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 s but with the white space excluded. This is equivalent to, "join(A) = del-spaces(s)", where join is the TypeScript function described in problem 4 and del-spaces is the function from quiz section that removes spaces from a string.
- Words, including punctuation, that were separated by spaces in the original string, 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 removing the spaces, we'll mark the locations where the string should be split. In our example, the splits would be at "|if| |this|,| |then| |that|", where each "|" indicates a split. Note that individual punctuation characters 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. For example, with the string above, the `splits` array would store $[0, 2, 3, 7, 8, 9, 13, 14, 18]$. It will always be the case that the first index is 0 and the last is the length of the string, so 0 and 18 in our example.

If i and j are two adjacent elements in the splits array, then $s[i \,..\, j-1]$ is one of the pieces. For example, for "|if| |this|,| |then| |that|", $s[9 \,..\, 13-1] = s[9 \,..\, 12] =$ "then". "then" is a word, but other adjacent elements may instead represent splits around a space. For example, $s[13 \,..\, 14-1] = s[13 \,..\, 13] =$ " ".

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.

(a) Implement the first loop of `splitWords`. 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. "$0 =$ splits$[0] <$ splits$[1] < \cdots <$ splits$[n-1] = j$, where $n =$ 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 splits$[i]+1 <$ splits$[i+1]$, then str[splits$[i]$ .. 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 $i = 1 \,..\, n-1$, 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.

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 it executes before each 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:

- Look at the provided test cases for some examples of cases you may want to consider. It is best to reason through these cases beforehand while writing the code rather than try to debug with the tests after.
- 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 the cases that the character you're looking at is 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. For the latter case, it may be useful to add a split to `splits` and update that value as you extend the piece.

After you've completed the first loop, take a look at the second loop of `splitWords` which has been provided 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 part (a), this loop uses a function, `CheckInv2`, to make sure the invariant is maintained.

(b) Verify that the provided tests now pass by running `npm run test`.

(c) Start the server by running `npm run start`. If that command fails due to "error TS5042" (as it seems to often on Windows), you can instead run `npm run build` followed by `npm run server`. The advantage of `npm run start` is that it will automatically restart the server if you change any of the TypeScript code. If you use the other commands, you will need to stop the server (with control-C) and re-run those commands to use the new code.

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.

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

The following parts consist entirely of coding work. They should be submitted with "HW7 Coding".

The application is now 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.

(a) We first need some place to store the transcripts in the server. We can take advantage of JavaScript's built-in `Map` type, which, like Java's `Map`, allows you to easily get and set values for a given key.

Add the following declaration (along with a descriptive comment) near the top of `routes.ts`, outside of any function. The variable "`transcripts`" will map string keys to JSON data. The latter will have the `unknown` type in TypeScript.

```
const transcripts: Map<string, unknown> = new Map<string, unknown>();
```

You can see this page for documentation on the `Map` class, including all of its available methods.

Now, you can fill in the missing part of the `save` function so that it stores the value sent by the client under the given name. The function should send back to the client a record with a field indicating whether the value save replaced a previously saved value.

You can also fill in the body of the "`resetTranscriptsForTesting`" function. This function, which is *only* used in the tests, should remove all the saved transcripts from the map. The `Map` class includes a method that will do this with a single call. (See the documentation.)

With those changes made, the 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 an error if the user did not provide a "name" query parameter or if there was no transcript previously saved with that name. Indicate the error with a 404 status and 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`.

When you are done, confirm that all tests pass by running `npm run test`.

(d) Add routes for `/load` and `/save` 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!

7. Extra Credit: She Wore a Raspberry Array (0 points)

The following parts consist entirely of written work. They should be submitted with "HW7 written".

In problem 1, we defined a simple function "substitute" that allowed us to replace words in an array with replacement words. In problems 2-3, we defined functions "replace" and "concat" whose composition has the effect of substitute but now with the ability to replace a word by multiple words. When a word is replaced by only a single word, however, this second version should do the same thing as the first. In this problem, we will prove that formally.

To state that claim formally, we first need some definitions. Let M_1 be any map of words to their replacements. Define M_2 to be the map from words to arrays of words containing the same keys as M_1 and where, if $v = M_1[w]$, then $[v] = M_2[w]$, i.e., the value associated with key w in M_2 is the 1-element array containing M_1's value for key w.

With those definitions, we want to prove that

$$\text{concat}(\text{replace}(A, M_2)) = \text{substitute}(A, M_1)$$

holds for any array A. Prove this by structural induction on A.