

# CSE 331: Software Design & Implementation

---

## Homework 7 (due Wednesday, May 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	map.ts	chatbot.ts
words.ts	routes_test.ts	map_test.ts	

Start by checking out the starter code using the command

```
git clone https://gitlab.cs.washington.edu/cse331-24sp/cse331-24sp-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".

### Dr. Melbourne

**Doctor** How do you do. Please tell me your difficulties.

**Patient** Hello!

**Doctor** Hi. What seems to be your problem?

|Type your answer here.

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 6, we saw that sets are easily represented using standard, inductive data types simply as lists containing no duplicates, the same is also true of maps. A map is an ADT that stores and supports operations on key-value pairs, and are easily represented as lists of key-value pairs. For example, the list

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

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 usually called "**association lists**", and they are every bit as common as Maps 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 (Note: this means the abstract state of maps, in this case, is a list of key-value pairs).

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.

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

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

Second, 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:

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

In `assoc.ts`, we have provided an `AssocList` type and "straight from the spec" TypeScript implementations of the above helper functions. The `AssocList` has `string` keys and uses a generic parameter "V" for its values. Later in the homework we'll implement a map ADT that utilizes this association list.

## 1. Loop Du Jour (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 substitutes some words for others in an array of words (strings). We can formally define the substitution operation as “ $\text{substitute}(A, M)$ ”, where  $A$  is an array of words,  $M$  is an association list with pairs of key words and (replacement) value words, and  $\text{substitute}$  is defined as follows:

```
func substitute([], M)      := []
substitute(A ++ [w], M)    := substitute(A, M) ++ [get-value(w, M)]  if contains-key(w, M)
substitute(A ++ [w], M)    := substitute(A, M) ++ [w]                if !contains-key(w, M)
```

(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  $\text{substitute}(\text{words}, \text{reps})$ , with the mathematical definition above.

(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  $n = \text{words.length}$ . **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.

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), then compares the result to the expected value. It is a good idea to write additional tests to be confident your code is correct or aid in debugging, but you will not turn these tests in 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 “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; we’ll prove the correctness of this in this problem. (The second step is concatenating all the arrays into a single array. We won’t require you to complete the proof of correctness for that step, but you can complete problem 6 for Extra Credit where we do this!)

The first part of the function calculates “ $\text{replace}(A, M)$ ”, where  $A$  is an array of words,  $M$  is an association list of pairs of words and arrays of words, and  $\text{replace}$  is defined as follows:

```

func replace([], M)      := []
      replace(A ++ [w], M) := replace(A, M) ++ [get-value(w, M)]  if contains-key(w, M)
      replace(A ++ [w], M) := replace(A, M) ++ [[w]]              if !contains-key(w, M)
  
```

The only difference between this definition and that of “ $\text{substitute}$ ” is on 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  $\text{get-value}(w, M)$ , 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  $\text{replace}(A, M)$  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 just after the conditional, and to use **backward** reasoning to fill in  $Q$  from what we know at the last assertion within the loop body. (If you choose to reason through the code differently, you may need to place your assertions differently than outlined above and the proofs required for part (b) will be different. This is not recommended.)

You **should not use subscripts** for this problem, instead write all assertions in terms of the current values of variables.

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 as so:  $Z \# [z]$ .

(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 problem 6 calculates  $\text{concat}(\text{replace}(L, M))$ , which is the one-to-many word replacement operation. (It may be useful to glance at the loop in problem 6 to be convinced this makes sense!)

### 3. 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. 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, so 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. (Equivalently: “ $\text{join}(A) = \text{del-spaces}(s)$ ”, where  $\text{join}$  is a TypeScript function which creates a string concatenating every value in the array with spaces in between and  $\text{del-spaces}$  removes all spaces from a string as described in section).
- Words, including punctuation, that were separated by spaces in the original string, should be recognized as different words in  $A$ . This means that for  $s = \text{“if this, then that”}$ , “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|”:

- $s[0 .. 2 - 1] = s[0 .. 1] = \text{“if”}$  (a word)
- $s[9 .. 13 - 1] = s[9 .. 12] = \text{“then”}$  (a word)
- $s[13 .. 14 - 1] = s[13 .. 13] = \text{“ ”}$  (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.

- (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. “ $0 = \text{splits}[0] < \text{splits}[1] < \dots < \text{splits}[n - 1] = j$ , 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 `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  $i = 1 .. n-2$ , `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 (“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 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:

- If you need more examples than those above, 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 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.

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

## 4. 20-Minute Power Map (20 points)

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

To finish our application, we will need additional operations that allow us to *mutate* a map, so our original `AssocList` is not sufficient. To do this, we will create an ADT that provides the additional functionalities to set a (key, value) pair in a map and clear its contents (which we will use only in testing).

(a) Create a new **interface** in `map.ts` that includes operations to:

- Check if there is some value associated with a given key in the map and return a boolean.
- Get the value associated with a given key if such a pair exists in the map, otherwise appropriately handle keys that do not exist.
- Set a value for a given key in the map, replacing the current value if a pair with the given key already exists. Return a boolean indicating if a value was replaced.
- Clear all pairs from the map. (This will only be used for testing.)

You may name your interface and its methods as you choose. Include proper JSDoc specification for these.

Hints: note that the first two operations have similar behavior to the provided helper functions in `assoc.ts`, like in HW5, you may be able to borrow from the existing documentation.

(b) Now create a **class** that implements the interface you designed above. Remember to properly document the class and abstraction function.

The constructor should take no parameters and create an empty `AssocList` (with the generic parameter `unknown`) to store in a field. Your method implementations can use the existing helper functions in `assoc.ts` as they are useful.

Hint: the set operation will need to add a pair to the `AssocList` field and reassign the field so it contains the updated pair. Note that this will sufficiently implement the required behavior of replacing any existing pair with the given key, provided the get value operation always returns the value of the first pair from the front of the list.

(c) Create a factory function, which you can decide the name of, that has a return type of the map interface and returns an instance of the class you used to implement the interface. Write a JSDoc specification for this function.

(d) Write tests for each function of your ADT in `map_test.ts`.

You should create an instance of the class using your factory function, then call the functions of your ADT on that instance within your tests.

Follow our testing heuristics to design your tests and add comment justifications for the tests you choose. Remember that the helper functions `get_value` and `contains_key` have already been tested (see `assoc_test.ts`).

(e) 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 our new mutable map ADT.

Update `LastUsed` to be a `const` variable with the type of your mutable map interface and initialize it by calling the factory function you created in part (d) at the top of `chatbot.ts`.

Update all usages of `LastUsed` in `chatbot.ts` to use the methods of your mutable map ADT. Each change that needs to be made is marked with a “TODO” comment, you need not update this file elsewhere, nor update other parts of the starter code for this step.

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



(f) 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. Note to Windows users: if that command fails due to “error TS5042”, 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.

## 5. 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 by implementing operations to save and load transcripts.

Transcript:

We will store the transcripts in the server using the mutable map you created in the last part. At the top of `routes.ts`, declare a `const` variable to hold the transcripts and initialize by calling the factory function you created for your ADT.

- (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.)

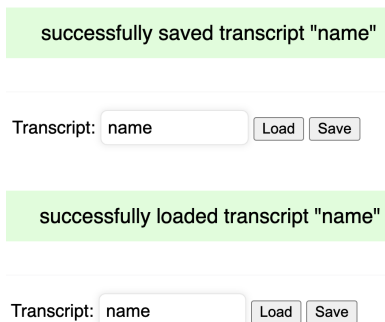
- (c) Write tests for your `load` function in `routes_test.ts`. There is an example test provided and some tips to get you started.

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 completing another application!

## 6. Extra Credit: Jumping Through Loops (0 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 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  $R$ . In the second part, we will concatenate those arrays together into a final array.

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

```
func concat([])           := []
concat( $R \# []$ )         := concat( $R$ )
concat( $R \# [A \# [w]]$ ) := concat( $R \# [A]$ ) \# [w]
```

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

$$\begin{aligned} & \text{concat}([A_1, \dots, A_{n-1}, A_n]) \\ &= \text{concat}([A_1, \dots, A_{n-1}] \# A_n) \\ &= \dots \\ &= \text{concat}([] \# A_1) \# \dots \# A_{n-1} \# A_n \\ &= A_1 \# \dots \# A_{n-1} \# A_n \end{aligned}$$

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

Now, consider the following code, which claims to calculate  $\text{concat}(R)$  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  $\text{concat}$  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.

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

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