

CSE 331: Software Design & Implementation

Homework 6 (due Wednesday, May 10th 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 “HW6 Written” assignment. The following completed files should be directly submitted for the coding portion to the “HW6 Coding” assignment: `words.ts` `words_test.ts`

For complete instructions for how to submit assignments in this course see the [Homework Turn-in Guide](#).

1. Loop Du Jour (20 points)

The following parts consist entirely of coding work. They should be submitted with “HW6 Coding”.

Start by checking out the starter code using the command

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

Then, install the modules using `npm install --no-audit`.

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 words to (replacement) words, and `substitute` is defined as follows:

```
func substitute([], M)      := []
substitute(A ++ [w], M)    := substitute(A, M) ++ [M[w]]   if w is in M
substitute(A ++ [w], M)    := substitute(A, M) ++ [w]      if w is not in M
```

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 function 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

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

where $n = \text{words.length}$. Your code must be correct with this invariant.

- (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 take more lines of code to set up than previous tests.

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

The following parts consist entirely of written work. They should be submitted with “HW6 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 “ $\text{replace}(A, M)$ ”, where A is an array of words and M is a map from words to arrays of words and replace is defined as follows:

```
func replace([], M)      := []
  replace(A ++ [w], M) := replace(A, M) ++ [M[w]]  if w is in M
  replace(A ++ [w], M) := replace(A, M) ++ [[w]]   if w is not in M
```

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 $\text{replace}(A, M)$ in the array R :

```
let R: string[] [] = [];
let i: number = 0;
{{ P1 : _____ }}
{{ Inv: R = replace(A[0 .. i - 1], M) }}
while (i !== A.length) {
  const val = M.get(A[i]);
  if (val !== undefined) {
    {{ P2 : _____ }}
    {{ Q2 : _____ }}
    R.push(val);
  } else {
    {{ P3 : _____ }}
    {{ Q3 : _____ }}
    R.push([A[i]]);
  }
  i = i + 1;
}
{{ P4 : _____ }}
{{ Q4 : R = replace(A, M) }}
```

- (a) Use reasoning to fill in all blank assertions above. The ‘ P_i ’s should be filled in with forward reasoning and the ‘ Q_i ’s should be filled in with backward reasoning (where ‘ Q_1 ’ is ‘Inv’).

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_i implies Q_i for $i = 1, 2, 3, 4$ (with Q_1 being "Inv").

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 "HW6 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:

```

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 `concat(R ++ [A]) = concat(R) ++ A` for any array A .

From the facts that `concat([]) = []` and `concat(R ++ [A]) = 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, `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) {
    {{ P3: _____ }}
    {{ Q3: _____ }}
    S.push(A[k]);
    k = k + 1;
  }
  {{ P4: _____ }}
  {{ Q4: _____ }}
  j = j + 1;
}
{{ P5: _____ }}
{{ Q5: S = concat(R) }}

```

- (b) Use reasoning to fill in all blank assertions above. The ' P_i 's should be filled in with forward reasoning and the ' Q_i 's should be filled in with backward reasoning.

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 reason through the inner loop (P_3, Q_3, P_4) by using Inv_2 as the starting fact for forwards and backwards reasoning around the loop. After those assertions are filled out, you can reason through the outer loop, filling in the surrounding assertions that are still blank.

- (c) Prove that P_i implies Q_i for $i = 1, 2, 3, 4$ (with Q_1 being "Inv").

Performing the loop from the previous problem followed by this one thus calculates $\text{concat}(\text{replace}(L, M))$, 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 “HW6 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:

```
func join-words([])      := ""
   join-words([w])      := w
   join-words(A ++ [w]) := join-words(A) + w      if w is punctuation
   join-words(A ++ [w]) := join-words(A) + " " + w if w not is punctuation
```

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 it would be `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. Words of a Feather Flock Together (20 points)

The following parts consist entirely of written work. They should be submitted with “HW6 Written”.

In this problem, we will check the correctness of a function that looks for a phrase (a sequence of words) within a sentence (a larger sequence of words). This is the main tool that the program will use to generate responses. It has a large number of patterns, each consisting of words to look for and formulas for how to generate a new sentence from the parts of an input that contains those words. For example, one pattern says that an input of the form $X + \text{[“was”, “I”]} + Y$ can generate the response $\text{[“Were”, “you”]} + Y + \text{[“?”]}$.

Consider the following code, which claims to look for the array of words “sub” appearing as a sub-array with the larger the array of words “all”. Let us denote the length of sub by “ m ” and the length of all by “ n ”. We can safely assume that $m \leq n$.

```
let k: number = -1;
{{ P1 : _____ }}
{{ Inv1: there is no index  $0 \leq j \leq k$  such that  $\text{sub}[i] = \text{all}[j + i]$  for all  $i = 0 .. m - 1$  }}
while (k + sub.length !== all.length) {
  k = k + 1;
  let l: number = 0;
  {{ P2 : _____ }}
  {{ Inv2: there is no index  $0 \leq j \leq k - 1$  such that  $\text{sub}[i] = \text{all}[j + i]$  for all  $i = 0 .. m - 1$  and
     $\text{sub}[i] = \text{all}[k + i]$  for all  $i = 0 .. l - 1$  }}
  while (l !== sub.length && sub[l] === all[k+l]) {
    {{ P3 : _____ }}
    {{ Q3 : _____ }}
    l = l + 1;
  }
  {{ P4 : _____ }}
  if (l === sub.length) {
    {{ P5 : _____ }}
    {{ Q5 :  $\text{sub}[i] = \text{all}[k + i]$  for all  $i = 0 .. m - 1$  }}
    return true;
  }
  {{ P6 : _____ }}
  {{ Q6 : _____ }}
}
{{ P7 : _____ }}
{{ Q7 : there is no index  $0 \leq j \leq n - m$  such that  $\text{sub}[i] = \text{all}[j + i]$  for all  $i = 0 .. m - 1$  }}
return false;
```

While the precise invariants are spelled out above, an intuitive explanation is the outer loop goes through all the indices where the sub-string we’re looking for could begin (tracking progress with k) and stops when the sub-string could not fit in the remaining string.

The inner loop begins at the index the outer loop is on and checks if sub can be found starting at that index in all. This means checking that the ℓ th index of all beyond k corresponds to the ℓ th index of sub. i is used to represent an arbitrary index of sub.

If the sub string was found in all, we return true. If not, we mark that starting index off and continue looking. If we get to the end of all without finding the sub string, the invariants assure us that it does not appear anywhere in the array. Next, we will check those claims formally.

- (a) Use reasoning to fill in all blank assertions above. The ' P_i 's should be filled in with forward reasoning and the ' Q_i 's should be filled in with backward reasoning.

Write your assertions with mathematical definitions and notations rather than code notation. For example, instead of referring to `all.length`, we'll denote it as ' n ', and instead of `sub.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:

- Note the advice from problem 3 for how to reason through nested loops and apply it here also.
 - Because there are multiple returns in this function, there are multiple post conditions: Q_5 and Q_7 . There is no need to rewrite either of these assertions, take them as given as usual.
 - When looking at P_3 and Q_3 , note that the first part of `Inv2` ("there is no index ... such that ... for all $i = 0 .. m - 1$ ") does not involve ℓ , which is the only variable changed in the loop, so that part of the invariant will remain true throughout the inner loop – meaning that fact can be left out for those assertions for clarity.
- (b) Prove that P_i implies Q_i for $i = 1, 2, 3$ and $5, 6, 7$ (with Q_1 being "`Inv1`" and Q_2 being "`Inv2`"). (There is no Q_4 . P_4 is just included to help you figure out P_5 and P_6)

6. At My Splits End (20 points)

The following parts consist entirely of coding work. They should be submitted with "HW6 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, $s[9 .. 13 - 1] = s[9 .. 12] = \text{"then"}$. "then" is a word, but other adjacent elements may instead represent splits around a space. For example, $s[13 .. 14 - 1] = s[13 .. 13] = \text{" "}$.

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] < ... < splits[n - 1] = j, where n = splits.length."
This says that the first split marks the first character, the last split marks 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 $i = 0 .. j - 1$, 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 $i = 1 .. j - 1$, 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) Verify that the app now works by running `npm run start` and 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.

Congratulations on completing 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 “HW6 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 .