CSE 331: Software Design & Implementation

Homework 4 (due Wednesday, October 25th 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 "HW4 Written" assignment. The following completed files should be directly submitted for the coding portion to the "HW4 Coding" assignment:

```
latin_ops.ts             list_ops.ts             ui.tsx
latin_ops_test.ts        list_ops_test.ts        ui_test.tsx
```

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

Reminder: Mutation is *not* allowed. Only write straight-line code, conditionals, and recursion. This applies to *all* code written for this assignment, including tests.

Before we can get to the problems, we need the following definitions.

Lists

In the previous assignment, we used lists containing several different types of data. Each of those was defined as a separate type. As a result, we had to define basic list functions like concat separately for each one of them, even though they are really all the same function.

In this assignment, we will instead define a generic list type

$$\textbf{type } \mathsf{List}_A \ := \ \mathsf{nil} \ \mid \ \mathsf{cons}(\mathsf{hd} : A,\ \mathsf{tl} : \mathsf{List}_A)$$

which can be used to store any type of data by substituting that data type for "A".

This list type still supports the standard functions we saw in lecture such as concat and rev:

$$\begin{aligned} \textbf{func } &\mathsf{concat}(\mathsf{nil}, R) &&:= R && \text{for any } R : \mathsf{List}_A \\ &\mathsf{concat}(\mathsf{cons}(a, L), R) &&:= \mathsf{cons}(a, \mathsf{concat}(L, R)) && \text{for any } a : A \text{ and } L, R : \mathsf{List}_A \end{aligned}$$

$$\begin{aligned} \textbf{func } &\mathsf{rev}(\mathsf{nil}) &&:= \mathsf{nil} \\ &\mathsf{rev}(\mathsf{cons}(a, L)) &&:= \mathsf{concat}(\mathsf{rev}(L), \mathsf{cons}(a, \mathsf{nil})) && \text{for any } a : A \text{ and } L : \mathsf{List}_A \end{aligned}$$

In the starter code for this assignment, the type `List`, the constant `nil`, and the functions `cons`, `concat`, `rev` are all defined in `list.ts`.

Character Lists

In the code, we will work exclusively with lists of character codes, which are integers. The following two functions, provided in `char_list.ts`, convert between strings and lists of character codes:

```
const explode = (s: string): List<number> => {..};
const compact = (L: List<number>): string => {..};
```

Like an array, a string is a compact and efficient representation of a sequence of characters, but as a result, correctness is more difficult with strings than lists. We will use these functions to let us work with lists instead.

The integers in the list version of the string are the Unicode character codes of the corresponding characters in the string. We can retrieve the character code from a string in JavaScript with the method `charCodeAt`, and we can convert a character code back to a string using `String.fromCharCode`. For example:

```
"a".charCodeAt(0);         // returns 97
String.fromCharCode(97);  // returns "a"
```

(Recall that JavaScript does not have a separate character type. Characters are just strings of length 1.)

1. Apple Cipher (10 points)

The following parts consist of a mix of written and coding work: part (a) is written and should be submitted with "HW4 Written", while parts (b,c) are coding and should be submitted with "HW4 Coding".

For the coding parts, you should start by checking out the starter code using the command

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

Then, install the modules using `npm install --no-audit`. You can run the tests using `npm run test`.

Our goal in this problem is to write a function cipher-encode that takes a list of characters as an argument and "returns a list of the same length but with each character replaced by the 'next' Latin character" and a function cipher-decode that takes a list of characters and "returns a list of the same length but with each character replaced by the 'previous' Latin character".

The 'next' and 'previous' characters are not the characters that come before or after a particular character in the standard alphabet. Instead they are defined by a scrambled arrangement of the characters created for this cipher. For example, character "a" has the 'next' character "e" and the 'previous' character "y", and "cse" fully encoded would be "kzi".

(a) Above, we were given an English definition of the problem, so our first step is to formalize it.

Write a formal definition of each function using recursion. Assume that we already have mathematical functions "nc" and "pc" that take a character code as input and return the next and previous Latin character's code, respectively ('next' and 'previous' defined by the cipher as mentioned above).

(b) Translate your definitions into TypeScript functions with the following declarations in `latin_ops.ts`:

```
export const cipher_encode = (L: List<number>): List<number> => {..};
export const cipher_decode = (L: List<number>): List<number> => {..};
```

The two mathematical functions nc and pc are already included in `latin_ops.ts`, where they are called `next_latin_char` and `prev_latin_char`, respectively. If you look through the implementations of these functions, you'll see the cipher's 'next' and 'previous' characters for each in the alphabet.

(c) Write test cases for the two new functions in the file `latin_ops_test.ts`. Follow the rules taught in lecture for choosing appropriate test cases.

Feel free to use `explode` (defined in `char_lists.ts`) and `compact` convert strings into lists and vice versa (in your tests only).

Include a brief comment above each test case justifying why you chose it.

Confirm that all your tests pass by running `npm run test`.

2. Whoopsie-Crazy (10 points)

The following parts consist of a mix of written and coding work: part (a) is written and should be submitted with "HW4 Written", while parts (b,c) are coding and should be submitted with "HW4 Coding".

Our goal in this problem is to write two functions. The first function, crazy-caps-encode, takes a list of characters as an argument and "returns a list of the same length but with *every other character*, starting with the first, made upper case". For example, the list corresponding to the string "crazy" would become the list corresponding to the string "CrAzY".

The second function, crazy-caps-decode, takes a list of characters as an argument and "returns a list of the same length but with *every other character*, starting with the first, made lower case". This undoes the effect of crazy-caps-encode.

Note that crazy-caps-encode is only defined on strings of lower case letters and crazy-caps-decode is only defined on strings in the form of the expected output of crazy-caps-encode (assuming it was given the correct lower case string input). This means that you do not have to even consider "bad" input cases when writing these functions, just assume the input is correct.

(a) Above, we were given an English definition of the problem, so our first step is to formalize it.

Write a formal definition of each function using recursion. In order to do so, you may need to define more than one function for each of these. Assume that we already have a mathematical function "uc" that returns the upper case version of a given character and a function "lc" that returns the lower case version of the given character.

(b) Translate your definitions into TypeScript functions with the following declarations in `latin_ops.ts`:

```
export const crazy_caps_encode = (L: List<number>): List<number> => {..};
export const crazy_caps_decode = (L: List<number>): List<number> => {..};
```

To get the character code for the capitalized version of a given character code, convert it from an integer to a string using `String.fromCharCode`, call the `toUpperCase` method, and then convert that string back to a number using the `charCodeAt` method. For lower case, use the same process but with `toLowerCase` instead. `toUpperCase` and `toLowerCase` are the Typescript versions of the "uc" and "lc" functions we used in our math definition.

(c) Write test cases for `crazy_caps_encode` and `crazy_caps_decode` in the file `latin_ops_test.ts`. Follow the rules taught in lecture for choosing appropriate test cases. (Be sure to include enough!) Note that you should only be defining test cases based on the expected behavior of the functions. You do not need to be writing tests with input that is not defined on these functions (see the above note).

Feel free to use `explode` and `compact` convert strings into lists and vice versa (in your tests only).

Include a brief comment above each test case justifying why you chose it.

Confirm that all your tests pass by running `npm run test`.

3. Have Your Take and Eat It Too (15 points)

The following parts consist entirely of written work. It should be submitted with "HW4 Written".

Below, we use the following function that returns *every other* element from a list, including the first:

$$\begin{aligned} \textbf{func } \text{take}(\text{nil}) &:= \text{nil} \\ \text{take}(\text{cons}(a, L)) &:= \text{cons}(a, \text{skip}(L)) \quad \text{for any } a : \mathbb{Z} \text{ and } L : \text{List} \end{aligned}$$

That function uses the following helper function, which does the same thing but starts by taking the second element in the list rather than the first one.

$$\begin{aligned} \textbf{func } \text{skip}(\text{nil}) &:= \text{nil} \\ \text{skip}(\text{cons}(a, L)) &:= \text{take}(L) \quad \text{for any } a : \mathbb{Z} \text{ and } L : \text{List} \end{aligned}$$

For example, with these definitions, we have $\text{take}(\text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil})))) = \text{cons}(1, \text{cons}(3, \text{nil}))$ and also $\text{skip}(\text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil})))) = \text{cons}(2, \text{nil})$.

Next, we define the following function that returns a list with an extra copy of every element, producing a list of twice the original length:

$$\begin{aligned} \textbf{func } \text{echo}(\text{nil}) &:= \text{nil} \\ \text{echo}(\text{cons}(a, L)) &:= \text{cons}(a, \text{cons}(a, \text{echo}(L))) \quad \text{for any } a : \mathbb{Z} \text{ and } L : \text{List} \end{aligned}$$

For example, we have $\text{echo}(\text{cons}(1, \text{cons}(2, \text{nil}))) = \text{cons}(1, \text{cons}(1, \text{cons}(2, \text{cons}(2, \text{nil}))))$.

(a) Prove, by structural induction, that we have $\text{take}(\text{echo}(S)) = S$ for any list S.

(b) You see the following snippet in some TypeScript code:

```
// Return take([1, 2] + echo(L)), where + is concat
return cons(1, L);  // much faster!
```

The comment tells us what it should return, but the code is not straight from the definition (level 1), so we will need to use reasoning to check that it is correct.

Show that this code is correct by proving that

$$\text{take}(\text{cons}(1, \text{cons}(2, \text{echo}(L)))) = \text{cons}(1, L)$$

for any list L. This should be a direct proof. Feel free to cite part (a).

4. Five, Fix, Seven, Eight (10 points)

The following parts consist of a mix of written and coding work: part (a) is written and should be submitted with "HW4 Written", while parts (b,c) are coding and should be submitted with "HW4 Coding".

Our goal in this problem is to write two functions. The first function, prefix(n, L), takes a natural number n and a list L of length at least n and "returns a list containing *just* the first n elements of L". The second function suffix(n, L), takes the same arguments and "returns a list containing everything *after* the first n elements of L". For example, prefix$(2, [1, 2, 3, 4, 5])$ would return $[1, 2]$ while suffix$(2, [1, 2, 3, 4, 5])$ would return $[3, 4, 5]$.

(a) Above, we were given an English definition of the problem, so our first step is to formalize it.

Write a formal definition of each function using recursion. Your functions must define an output for every list L and natural number n; however, the description above only specifies the output when the length of L is at least n. For list containing fewer than n elements, you can return the value "undefined".

Remember that the natural numbers are an inductively defined type: every natural number is either 0 (the base case) or $n + 1$ for some natural number n (the recursive case). You should take advantage of this when writing your functions by pattern matching.

(b) Translate your definitions into TypeScript functions with the following declarations in `list_ops.ts`:

```
export const prefix = <A,>(n: number, L: List<A>): List<A> => {..};
export const suffix = <A,>(n: number, L: List<A>): List<A> => {..};
```

Note that the return value is `List<A>` not "`List<A> | undefined`". If the list contains fewer than n elements, have your function throw an `Error` rather than returning undefined (as in the math definition).

(c) Write test cases for the two new functions in the file `list_ops_test.ts`. Follow the rules taught in lecture for choosing appropriate test cases.

Again, feel free to use `explode` and `compact` convert strings into lists and vice versa in your tests.

Include a brief comment above each test case justifying why you chose it.

Confirm that all your tests pass by running `npm run test`.

5. Living in the Last (25 points)

The following parts consist entirely of written work. It should be submitted with "HW4 Written".

Below, we use the following function that returns the last element from a list (or undefined if it has none):

$$\begin{aligned}
&\textbf{func}\ \text{last}(\text{nil}) && := \text{undefined} \\
&\phantom{\textbf{func}}\ \text{last}(\text{cons}(a, \text{nil})) && := a && \text{for any } a : \mathbb{Z} \text{ and } L : \text{List} \\
&\phantom{\textbf{func}}\ \text{last}(\text{cons}(a, \text{cons}(b, L))) && := \text{last}(\text{cons}(b, L)) && \text{for any } a, b : \mathbb{Z} \text{ and } L : \text{List}
\end{aligned}$$

If the list is empty, then there is no last element, so this returns undefined. If the list has only one element, a, then that is the last element, so a is returned. Otherwise, the list has at least two elements left, which means that the first element in the list is not the last, so we recurse on the rest.

(a) Prove, for any $b : \mathbb{Z}$ and L : List, that concat$(L, \text{cons}(b, \text{nil})) \neq$ nil.

Do this by cases on the shape of L. That is, every list is either nil or cons(a, R) for some $a : \mathbb{Z}$ and R : List. If we can prove the claim for lists of each of those shapes, then it holds for all lists.

Note that these cases are not only exhaustive but also *exclusive*. A list like cons(a, R) is not nil!

(b) Prove that, for any $b : \mathbb{Z}$ and any S : List, we have

$$\text{last}(\text{concat}(S, \text{cons}(b, \text{nil}))) = b$$

Do this by structural induction on S. Feel free to cite part (a) in your proof.

(c) You see the following snippet in some TypeScript code:

```
// Return last(rev(L))
return L.hd;  // since we know L != nil
```

The comment tells us what it should return, but the code is not straight from the definition, so we will need to use reasoning to check that it is correct.

First, note that, since we know $L \neq$ nil, we must have $L =$ cons(a, R) for some $a : \mathbb{Z}$ and R : List (because those two cases are exclusive), and in that case, `L.hd` is a.

Thus, we can show that this code is correct by proving that

$$\text{last}(\text{rev}(\text{cons}(a, R))) = a$$

for any $a : \mathbb{Z}$ and R : List. Do this with a direct proof. Feel free to cite part (b).

6. Vowel Movement (10 points)

The following parts consist of a mix of written and coding work: parts (a,c) are written and should be submitted with “HW4 Written”, while parts (b,d) are coding, to be submitted with “HW4 Coding”.

Our goal in this problem is to write two functions, pig-latin-encode and pig-latin-decode, each of which takes a list of characters as input and returns a new list of characters, with the former function translating a word into Pig Latin and the second function doing the inverse.

These function descriptions can look very complicated at first glance, so it is **highly recommended** that you read carefully, try to understand every single case independently, and try examples before writing your formal definition.

(a) In more detail, pig-latin-encode should transform lists of character codes as follows:

- If the list does not start with $n \geq 0$ consonants followed by a vowel, then return the input unchanged.
- If the list begins with $n = 0$ consonants followed by a vowel, then return the list representing the original string followed by “way”, e.g., “astray” becomes “astrayway”.
- If the list begins with $n \geq 1$ consonants followed by a vowel, then return a list with the suffix starting at that vowel followed by those consonants followed by “ay”, e.g., “call” becomes “allcay” and “stray” becomes “aystray”.
- As an exception to the previous case, if the last of the $n \geq 1$ consonants is a “q”, the first vowel is a “u”, and the “u” is followed by another vowel, then move the “u” with the consonants as well, e.g., “quay” becomes “ayquay”, not “uayqay”, and “rquit”becomes “itrquay”.

Write a formal definition of this function. Our mathematical definitions of functions do not include an “if” expression, but remember that you can include side conditions to further describe patterns. If any side condition is too long to write on one line, feel free to replace it with a short-hand label (e.g., “A”) and then define the full expression after the rest of the function.

Assume that there is already a mathematical function “cc” that returns the number of consonants at the beginning of the list before the first vowel and returns -1 if there are no vowels. Feel free to use “explode” to translate string constants like “way” into lists as well as the functions discussed previously like concat, prefix, suffix, last, and first (which you can assume exists and behaves similarly to last except instead getting the first value in a list). Do not use explode or compact on non-constant values. Feel free to assume there is a function “char” that takes a single character as a string and returns its character code. (We could define that as char(s) := first(explode(s)).)

(b) Translate your definitions into a TypeScript function with the following declaration in `latin_ops.ts`:

```
export const pig_latin_encode = (L: List<number>): List<number> => {..};
```

A function very similar to the mathematical function “cc” is provided in the code as

```
export const count_consonants = (L: List<number>): number|undefined => {..};
```

This is the same as cc except that it returns undefined when cc would return -1.

There are tests provided for this function in `latin_ops_test.ts`. Un-comment these and confirm that they all pass by running `npm run test`.

(c) The second function, pig-latin-decode, should transform lists of character codes as follows:

- If the list does not end with “ay”, then return it as is. (It doesn’t look like Pig Latin.)

- If there are no consonants just before “ay” or there is no vowel just before those consonants, then return it as is. (It doesn’t look like Pig Latin.)
- If the character just before “ay” is a “w” and the character before that is a vowel, then return everything before “way”, e.g., “away” becomes “a”.
- If the characters just before “ay” are "qu", then return “qu” followed by everything before “quay”, e.g., “ayquay” becomes "quay". If there are additional consonants before the “qu”, then those constants move to the front also, e.g., “aysquay” becomes “squay”, and “ailquay” becomes “lquai”.
- If there are $n \geq 1$ consonants before “ay” and the two prior cases don’t apply, then return those consonants followed by the characters before the consonants, e.g., “aystray” becomes “stray”, and “ateplay” becomes “plate”.

Write a formal definition of this function. (Do not stress out about the mathematical notation. Notation is just communication. The important point is to figure this out on paper, away from a keyboard.)

Hint: It may be easier to describe the conditions for cases with the list reversed!

(d) Translate your definitions into a TypeScript function with the following declaration in `latin_ops.ts`:

```
export const pig_latin_decode = (L: List<number>): List<number> => {..};
```

There are tests provided for this function in `latin_ops_test.ts`. Un-comment these and confirm that they all pass by running `npm run test`.

7. Cat's Out of the Tag (20 points)

The following parts consist entirely of coding work.

In this problem, we will create a simple UI that allows users to apply the cipher, crazy caps, and pig-latin encoding and decoding functions you created in the previous problems.

(a) Fill in the implementation of the function `ShowResult` in `ui.tsx`.

This function is passed a record containing a word; a name of an algorithm to apply, which is one of "cipher", "crazy-caps", or "pig-latin"; and an operation, which is one of "encode" or "decode". If the operation is "encode", you should call in the indicated `*_encode` function that you wrote earlier, and if it is "decode", you should call the indicated `*_decode` function.

For this problem, you should use `explode` and `compact` to translate strings to lists and vice versa.

Once you have the resulting string, return it inside of HTML "code" tag wrapped in a "p" (paragraph). The former will display the string in a fixed-width font inside the paragraph.

Start the application by running `npm run start`. You can then try out the application in a browser by opening `http://localhost:8080?word=`W`&algo=`A`&op=`O, where W is any word, A is one of the algorithm choices listed above, and O is one of the operation choices listed above.

(b) Write tests for `ShowResult` in `ui_test.tsx`.

Include a brief comment above each test case justifying why you chose it.

Confirm that all your tests pass by running `npm run test`.

(c) Fill in the implementation of the function `ShowForm` in `ui.tsx`.

There's a good amount of starter code to help you get started, but you should expand on it based on our expectations described below. (See the Mozilla Developer Network pages for more information on the described HTML tags if needed, and see the forms we used in prior HWs for examples).

The function should return an HTML "form" tag. Remember, when the "Submit" button is clicked, the page is redirected to the URL described by the `action` attribute (`"/"` is a stand in for `"http://localhost:8080/"`) with any parameters included in the form through "input" tags appended to the end. The `name` attribute of "input" tags determines the name of query parameters, and the `value` attribute determines the value of that parameter.

Your UI should allow the user to choose which algorithm to use with a drop-down, with their choice being placed in the "algo" query parameter of the URL.

It should also allow the user to specify whether to encode or decode using radio buttons, with their choice being placed in the "op" query parameter of the URL.

You are free to spend some time adjusting the styling to look the way you want. However, you will not be graded on how it looks as long as it is clear how to use it. One thing that can help with clarity are "label"s for the "input"s of the form, for example:

```
<label htmlFor="word">Word:</label>
```

(d) This function has only a single configuration, so we can test it manually:

- Open the browser to `http://localhost:8080` and make sure that you see the form.
- Fill in the form fields, press Submit, and make sure you see the result (using your UI from part (a)).
- Change each of the fields, press Submit, and make sure you see the change reflected in the result.

Congratulations! You have a functioning app.

8. Extra Credit: Roll the Twice (0 points)

Prove that take(twice-evens(S)) = twice(take(S)) holds for any list S, where "twice" and "twice-evens" are the functions we defined in the problems "Sugar and Spice and Everything Twice" and "Miami Twice", respectively, in quiz section.

Your proof should be by structural induction, but note that you cannot prove this statement directly. You need to prove a stronger claim to get this to work. (Hint: Try proving the statement as is and see where you get into trouble. You should then see what other facts you would need to prove to get this to work.)