CSE 331: Software Design & Implementation

Homework Cipher (due Wednesday, July 17th at 11:00 PM)

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

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

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`, and `rev` are all defined in `list.ts`.

Character Lists

In this assignment we will create encoding and decoding functions to disguise strings. Correctness is more difficult with strings than lists, so we will **represent strings as Unicode character codes:** `List<number>`. (Character codes are technically integers, but the TypeScript `String` functions represent them with type `number`, so we will too!)

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. 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 => {..};
```

We will use these functions to help us take string inputs and convert to our desired form, and vice versa.

1. Apple Cipher (12 points)

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

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

```
git clone https://gitlab.cs.washington.edu/cse331-24su/cse331-24su-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 "i" and the 'previous' character "o". "cse" fully encoded would be "zky" and "zky" full decoded would be "cse".

(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 ("straight from the spec") 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.

It's easier to think in terms of strings than character codes, so feel free to use `explode` (defined in `char_lists.ts`) and `compact` to convert strings into lists and vice versa for your inputs/outputs (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 (13 points)

The following parts consist of a mix of written and coding work: part (a) is written and should be submitted with "HW Cipher Written", while parts (b,c) are coding and should be submitted with "HW Cipher 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 three characters*, starting with the third, made upper case". For example, the list corresponding to the string "craziness" would become the list corresponding to the string "crAziNesS".

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 three character*, starting with the third, made lower case". This undoes the effect of crazy-caps-encode.

Note: 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. 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 ("straight from the spec") 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> => {..};
```

The two mathematical functions uc and lc are already included in `latin_ops.ts`, where they are called `to_upper_case` and `to_lower_case` respectively.

(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. (Be sure to include enough!) Note that you should only be defining test cases based on the expected behavior of these functions. You do not need to be writing tests for input that is not defined on these functions (see the above note). Hint: refer to the testing that was done for `twice_evens` in Section Quilt!

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`.

3. The Horn on the Bus Goes Keep, Keep, Keep (15 points)

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

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

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

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.

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

For example, with these definitions, we have $\mathsf{keep}(\mathsf{cons}(1, \mathsf{cons}(2, \mathsf{cons}(3, \mathsf{nil})))) = \mathsf{cons}(1, \mathsf{cons}(3, \mathsf{nil}))$ and also $\mathsf{drop}(\mathsf{cons}(1, \mathsf{cons}(2, \mathsf{cons}(3, \mathsf{nil})))) = \mathsf{cons}(2, \mathsf{nil})$.

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:

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

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

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

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

```
// Return keep([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 spec (level 1), so we will need to use reasoning to check that it is correct.

Show that this code is correct by proving that

$$\mathsf{keep}(\mathsf{cons}(1, \mathsf{cons}(2, \mathsf{echo}(L)))) = \mathsf{cons}(1, L)$$

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

4. Five, Fix, Seven, Eight (13 points)

The following parts consist of a mix of written and coding work: part (a) is written and should be submitted with "HW Cipher Written", while parts (b,c) are coding and should be submitted with "HW Cipher 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 ("straight from the spec") into TypeScript functions with the following declarations in `list_ops.ts`:

```
export const prefix = <A,>(n: bigint, L: List<A>): List<A> => {..};
export const suffix = <A,>(n: bigint, 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 "HW Cipher Written".

Below, we use the following function that returns the last element from a list:

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

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, which means that the first element in the list is not the last, so we recurse on the rest of the list.

(a) Prove, for any $b : \mathbb{Z}$ and L : List, that concat$(L, \mathsf{cons}(b, \mathsf{nil})) \neq$ 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 *exclusive*. A list of the shape cons(a, R) is not nil!

(b) Prove, by structural induction, that we have last(concat$(S, \mathsf{cons}(b, \mathsf{nil}))) = b$ for any $b : \mathbb{Z}$ and any S : List. 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 spec (level 1), 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 $L = \mathsf{cons}(a, R)$ for some $a : \mathbb{Z}$ and R : List because those two cases are exclusive. Therefore, `L.hd` is a.

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

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

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

6. Vowel Movement (12 points)

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

Our goal in this problem is to write a function, worm-latin-encode, which takes a list of characters as input and returns a new list of characters, translating the given word into Worm Latin, our version of Pig Latin. The inverse of worm-latin-encode, worm-latin-decode, has been provided for you.

Disclaimer: unlike the other ciphers, a string that is encoded to Worm Latin cannot be perfectly decoded back to its starting input. This is expected, so do not be concerned if you find examples that don't translate back and forth in this way.

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

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

- If the list represents the string "bird", return the empty list. Guess the early bird got the worm!
- If the list does not contain a vowel (i.e. all consonants or no letters), return the input unchanged.
- If the list begins with $n = 0$ consonants followed by a vowel (i.e. begins with a vowel), return "w", followed by the beginning vowel, followed by "orm", followed by the suffix of the original string containing all but the beginning vowel, e.g., "elf" becomes "weormlf" and "autumn" becomes "waormutumn".
- If the list begins with $n \geq 1$ consonants followed by a vowel (i.e. begins with at least 1 consonant, then a vowel), return a list with the suffix of the original list starting at that vowel, followed by the prefix of the list containing only the initial consonants, followed by "orm", e.g., "thing" becomes "ingthorm", "stray" becomes "aystrorm", and "kevin" becomes "evinkorm".

Write a formal definition for this function. Unlike prior problems, it is not possible to write this math definition with pattern matching, instead **you must use side conditions** to describe qualities of inputs. 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. (This problem is likely a good example of how complicated side conditions can get. There is no alternative here, but if there was a cleaner pattern matching approach, it would be much more readable.)

Assume that there is already a mathematical function "cc" that takes a list of character codes and returns the number of consonants at the beginning of the list before the first vowel, or -1 if there are no vowels, e.g. "Katherine" returns 1, "cd" returns -1, and "Adam" returns 0.

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)), using the first function defined in lecture.)

Feel free to use "explode" and "compact" to translate string into lists and vice versa (though do not use these functions on non-constant values), as well as the other functions discussed previously like concat, prefix, suffix, first, and last (note that you should not use `hd` and `tl` in math as they only exist in code).

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

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

"cc" is translated into TypeScript and provided for you to use in `latin_ops.ts` as:

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

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

Note that these are only a subset of the tests that we need to write to be confident our code is correct according to our heuristics. Like HW Quilt, there will be some hidden staff tests for this function that will run on your code after the deadline. So, though we will not require you to write additional tests (any tests you add will not be graded), it's a good idea to take some extra time to be sure your function is correct.

(c) **Extra Credit**: The implementation of the inverse of `worm_latin_encode`, `worm_latin_decode`, has been provided for you in `latin_ops.ts`. worm-latin-decode, should transform lists of character codes as follows:

- If the list is empty, return the list representation of "bird".
- If the list begins with "w", followed immediately by a vowel and then "orm", return the vowel immediately after "w" followed by everything after "orm", e.g. "woormrm" becomes "orm".
- If the list ends in "orm" with at least one consonant immediately before, and the list starts with a vowel, return those consonants followed by the characters before the consonants. e.g., "ameshorm" becomes "shame", and "entorm" becomes "nte".
- Otherwise, the initial character is not "w" followed immediately by a vowel and then "orm" and the list does not start with a vowel and have consonants just before the ending "orm", return it as is. (It doesn't look like Worm Latin.) e.g. "ateorm" and "storm".

Write a formal definition for this function. Math formalization tips and hints:

- For conciseness, feel free to define variables that represent repeated expressions in your conditions or outputs. It may be useful to use labels for certain conditions (e.g. "A is '$n > 0$'") and combine labels with and, not, or (e.g. "A and not B").
- Make sure that your conditions are **exclusive and exhaustive**. This means you should be able to reorder the cases and all inputs will still fall in the proper case which may require including the opposite of a condition for a certain case to exclude it from another.
- **You should describe parts of the conditions for cases with the list reversed!** This is useful because "cc" counts consonants at the *beginning* of a list. (Note that, for any list L, it is true that rev(rev(L)) $= L$, we can prove this with structural induction, but we won't make you!)

7. Cat's Out of the Tag (10 points)

The following parts consist entirely of coding work. It should be submitted with "HW Cipher Code".

In this problem, we will create a simple UI that allows users to apply the cipher, crazy caps, and worm-latin encoding and decoding functions that 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 "worm-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.

We have provided a few tests for `ShowResult` in `ui_test.tsx` but not enough according to our heuristics. Though they will not be graded, we encourage you to write a few more or thoroughly manually test this functionality through the browser.

(b) 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 linked documentation and the forms from prior homeworks for examples as needed).

The function returns an HTML "form" tag. When the "Submit" button is clicked, the page is redirected to the URL described by the `action` attribute (`"/"` can be substituted 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.

It should be clear how to use the app, so we recommend you include "label"s for the "input"s of the form. For example:

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

(c) `ShowForm` 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 (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.