CSE 331: Software Design & Implementation

Homework 3 (due Wednesday, April 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 as a pdf to the "HW3 Written" assignment. The following completed files should be directly submitted for the coding portion to the "HW3 Coding" assignment:

```
patterns.ts              quilt_ops.ts              index.tsx
patterns_test.ts         quilt_ops_test.ts
```

When the autograder runs for "HW3 Coding," it will run the tests you wrote and let you know if they pass/fail. However, this will not award any points, we will manually grade your tests for following the heuristics and justifications. If for some reason your tests aren't passing and you don't have time to fix it, please leave them as is, if you comment out any tests or code, we will not grade it.

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

Squares

In this assignment, we will write some programs that display quilt patterns. Each quilt is made up of squares. Mathematically, each square is a record of the following type:

$$\textbf{type}\ \text{Square} := \{\text{shape} : \text{Shape},\ \text{color} : \text{Color},\ \text{corner} : \text{Corner}\}$$

The properties of individual squares include shape and color, which are elements of the following types:

$$\textbf{type}\ \text{Shape} := \text{STRAIGHT} \mid \text{ROUND}$$

$$\textbf{type}\ \text{Color} := \text{GREEN} \mid \text{RED}$$

The last property, corner, describes the orientation of the square. Both square shapes have drawing primarily in one corner, so specifying which corner that is indicates the orientation, i.e., how the square is rotated.

$$\textbf{type}\ \text{Corner} := \text{NW} \mid \text{NE} \mid \text{SW} \mid \text{SE}$$

Quilts

A quilt is a 2D table of squares. We will represent each quilt as a list of lists of squares. We will call a list of squares a "row", and then a quilt is a list of rows. We define these two types inductively as follows:

$$\begin{aligned}\textbf{type}\ \text{Row} &:= \text{rnil} \mid \text{rcons}(\text{hd} : \text{Square},\ \text{tl} : \text{Row})\\ \textbf{type}\ \text{Quilt} &:= \text{qnil} \mid \text{qcons}(\text{hd} : \text{Row},\ \text{tl} : \text{Quilt})\end{aligned}$$

All rows in a quilt should have the same length. Mathematically, we can define a row's length recursively:

$$\begin{aligned}\textbf{func}\ \text{rlen}(\text{rnil}) &:= 0\\ \text{rlen}(\text{rcons}(a, L)) &:= 1 + \text{rlen}(L)\end{aligned}$$

Note, however, that our type definitions allow the quilt to contain rows of different lengths! It is an additional *invariant* of the Quilt type that all rows in each quilt must have the same length.

We can also define concatenation of rows as follows:

$$\begin{aligned}\textbf{func}\ \text{rconcat}(\text{rnil}, R) &:= R\\ \text{rconcat}(\text{rcons}(s, L), R) &:= \text{rcons}(s, \text{rconcat}(L, R))\end{aligned}$$

These two functions, whose names start with "r", are defined on lists of squares (rows). There are analogous definitions of functions, qlen and qconcat, whose names start with "q", that operate on lists of rows (quilts).

1. We Find the Defendant Quilty (20 points)

The following parts consist of a mix of written and coding work: parts (a,c,e,g) are coding and should be submitted with "HW3 Coding", while parts (b,d,f) are written and should be submitted with "HW3 Written".

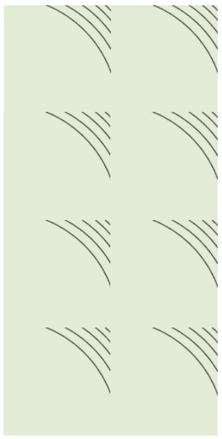
Start by checking out the starter code using the command

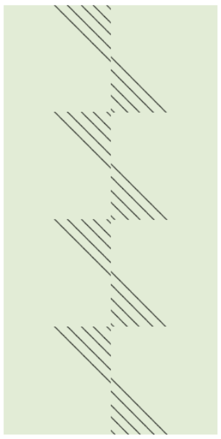
```
git clone https://gitlab.cs.washington.edu/cse331-24sp/cse331-24sp-materials/hw-quilt.git
```

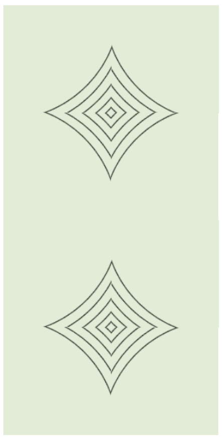
Then, install the modules using `npm install --no-audit`. You can run the tests using `npm run test`.

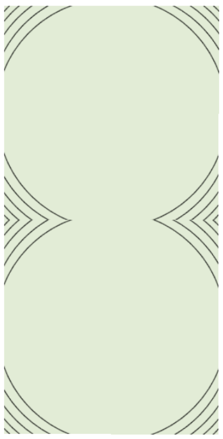
The provided TypeScript code includes the functions from the previous page in `quilt.ts`. It also includes a variety of constant variables to use. Have a look at those and feel free to `import` what you need for this part and the rest of the homework.

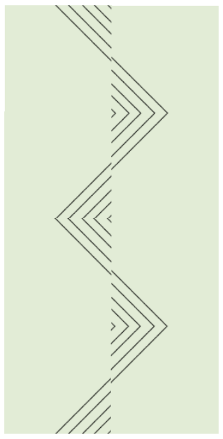
(a) Implement the functions `PatternA`, . . . , `PatternE` in `patterns.ts` to return quilts displaying each of the 2×4 pattern shown below. (Implement them in the most straightforward manner—no loops or recursion! Our staff solutions for the functions at this step each have 3-7 lines of code.)


A


B


C


D


E

You can see the pattern returned by your code by doing the following: start the application with `npm run start`, open a browser to `http://localhost:8080?pattern=A`, and change the `pattern=` query parameter in the URL to the pattern you want to see.

(b) What level of the correctness does the programming in part (a) fall in? What steps are required to ensure that that your answer is correct? Check that your code in part (a) is correct by doing those steps.

(c) Change each of the pattern functions to support an optional "color" argument of type `Color`. If the argument is not provided, default to green. Modify `index.tsx` to pass in the color argument. It is already parsed in the `GetColor` function which is called on line 74, but the result is not used in the following calls to the `Pattern` functions.

You can change the color to red by adding the `color=red` query parameter in the URL: `http://localhost:8080?pattern=A&color=red`

(d) What level of the correctness does the programming in part (c) fall in? What steps are required to ensure that that your answer is correct? Check that your code in part (c) is correct by doing those steps.

(e) Next, we will change each pattern function to be able to produce quilts with different numbers of rows. We'll do so by giving them a (required) "rows" argument of type `bigint` indicating the required number. First, however, we need to specify each pattern's behavior for different numbers of rows. This is done for you with mathematical definitions of patterns A–E, given on page 11.

Change the code for each pattern to match these definitions, matching the mathematical definition as closely as possible (for example, use recursion!). Our staff solutions for the functions at this step each have 10-20 lines of code.

The patterns are not defined for negative numbers of rows. In those cases, your code should throw an `Error`. (The syntax for this is the same as in Java: `throw new Error(".. explanation ..")`.) Furthermore, patterns C and D are only defined for even numbers of rows, so it should throw an `Error` for odd numbers as well. Note that unlike previous functions you've wrote in this course where these checks were just encouraged as defensive programming, it is not optional to include the errors for these, we are letting callers of the function rely on the error behavior.

In Typescript, required parameters must be listed before optional parameters, so you'll need to place the "rows" argument before the "color" argument.

Modify `index.tsx` to pass in the row query argument. It is already parsed in the `GetRows` function which is called on line 75, but the result is not used in the following calls to the `Pattern` functions. Then, you can change the number of rows to 8, for example, by adding a `rows=8` query parameter in the URL: `http://localhost:8080?pattern=A&color=red&rows=8`.

For now, confirm that each of your functions produces the same patterns as before when we set `rows=4` but that they return only the top half when we set `rows=2`. (We will do more testing below.)

(f) What level of the correctness does the programming in part (e) fall in? What steps are actually required to ensure that that your answer is correct? (Hint: the checks done in (e) are not enough.) Also, why is it important that we write the code to be as similar to the mathematical description as possible?

(g) Following the heuristics we've learned, write test cases for each of the pattern functions in the file `patterns_test.ts`. An example test case is already provided for PatternA, but you will need more cases to cover all the important subdomains.

Include a brief comment above each test case justifying why you chose it. See the "Writing Test Cases" section at the end of homework 2 for a reminder of what we're looking for.

Note that you can check that a function throws an exception using `assert.throws`. An example of how to do this is included in `patterns_test.ts` that you can uncomment. You should test the error cases you wrote for each function.

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

2. Twice to Meet You (18 points)

The following parts consist entirely of written work. It should be submitted with "HW3 Written".

Suppose that you see the following snippet of TypeScript code in some large TypeScript program. The code in the snippet uses `len`, sum, `twice_evens`, and twice_odds, all of which are TypeScript implementations of the mathematical functions of the same names defined in the section solutions.

Suppose that you see the following snippet of TypeScript code in some large TypeScript program. Assume the functions used in the code are TypeScript implementations of the mathematical functions that we've defined of the same names.

```
const x = sum(twice_evens(L));
const y = sum(twice_odds(L));

if (len(L) === 2) {
  return x + y;  // = 3 * sum(L)
}
```

The comment shows the definition of what should be returned, but the code is not a direct translation of that. Below, we will use reasoning to prove that the code is correct.

Note that, if $\mathsf{len}(L) = 2$, then $L = \mathsf{cons}(a, \mathsf{cons}(b, \mathsf{nil}))$ for some integers a and b, as we have previously proven in section. We will use that below.

(a) Using the fact that $L = \mathsf{cons}(a, \mathsf{cons}(b, \mathsf{nil}))$, prove by calculation that $\mathsf{sum}(L) = a + b$.

(b) Using the fact that $L = \mathsf{cons}(a, \mathsf{cons}(b, \mathsf{nil}))$, prove by calculation that $\mathsf{sum}(\text{twice-evens}(L)) = 2a + b$.

(c) Using the fact that $L = \mathsf{cons}(a, \mathsf{cons}(b, \mathsf{nil}))$, prove by calculation that $\mathsf{sum}(\text{twice-odds}(L)) = a + 2b$.

(d) Prove that the code is correct by showing that $x + y = 3\,\mathsf{sum}(L)$, i.e., that

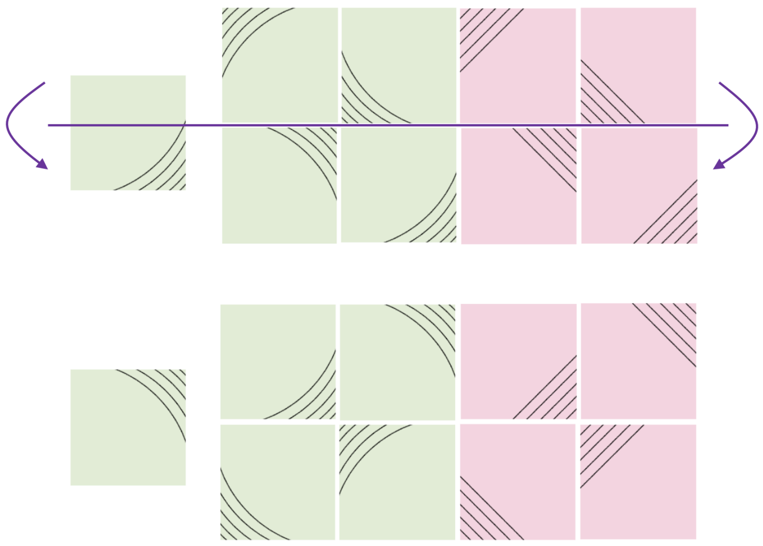
$$\mathsf{sum}(\text{twice-evens}(L)) + \mathsf{sum}(\text{twice-odds}(L)) = 3\,\mathsf{sum}(L)$$

You are free to cite parts (a-c) in your calculation since we know that $L = \mathsf{cons}(a, \mathsf{cons}(b, \mathsf{nil}))$ holds on the line with the `return` statement.

3. Skinny Flipping (22 points)

The following parts consist of a mix of written and coding work: parts (b,e,h) are coding and are submitted with "HW3 Coding", while parts (a,c,d,f,g) are written and are submitted with "HW3 Written".

In this problem, we will write a function that "flips a quilt vertically, as if spun around a horizontal line through the center". Here is an example (the bottom quilt is the result of vertically flipping the top quilt):



(a) The problem definition was in English, so our first step is to formalize it.

Start by writing a mathematical definition of a function "sflip-vert" that flips a single **square** vertically.

(b) Translate your mathematical definition into TypeScript code in the `quilt_ops.ts` file.

Write tests for your function in `quilt_ops_test.ts`.

For this function, and all functions in problems 3 and 4, we have provided one random example test. They are a good example of test formatting, but you'll need to determine how they fit into our testing heuristics, and add tests to cover all the cases our heuristics require.

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

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

(c) Next, we will define a mathematical function "rflip-vert" that flips a **row** vertically.

Let's start by writing this out in more detail. Let a, b, and c be squares. Fill in the blanks showing the result of applying rflip-vert to different rows, which we will write as lists of squares.

Feel free to abbreviate sflip-vert in your answer as "s".

rnil ______________________

rcons(a, rnil) ______________________

rcons(a, rcons(b, rnil)) ______________________

rcons(a, rcons(b, rcons(c, rnil))) ______________________

...

(d) Write a mathematical definition of a function rflip-vert using recursion.

(e) Translate your mathematical definition into TypeScript code in the `quilt_ops.ts` file.

Write tests for your function in `quilt_ops_test.ts`.

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

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

(f) Now, we are ready to define a function "qflip-vert" that flips a **quilt** vertically. Note that this operation flips individual rows vertically, but also *reverses* their order!

Again, let's start by writing this out in more detail. Let u, v, and w be rows. Fill in the blanks showing the result of applying qflip-vert to different quilts, which we will write as lists of rows.

Your answers should use rflip-vert (not sflip-vert), which you can abbreviate as just "r".

qnil	____________________
qcons$(u,$ qnil$)$	____________________
qcons$(u,$ qcons$(v,$ qnil$))$	____________________
qcons$(u,$ qcons$(v,$ qcons$(w,$ qnil$)))$	____________________

...

(g) Write a mathematical definition of a function "qflip-vert".

Hint: it may be useful to review definition of the function rev, from lecture. Also, remember that the function qconcat, which concatenates two quilts, is already provided for you.

(h) Translate your mathematical definition into TypeScript code in the `quilt_ops.ts` file.

Write tests for your function in `quilt_ops_test.ts`.

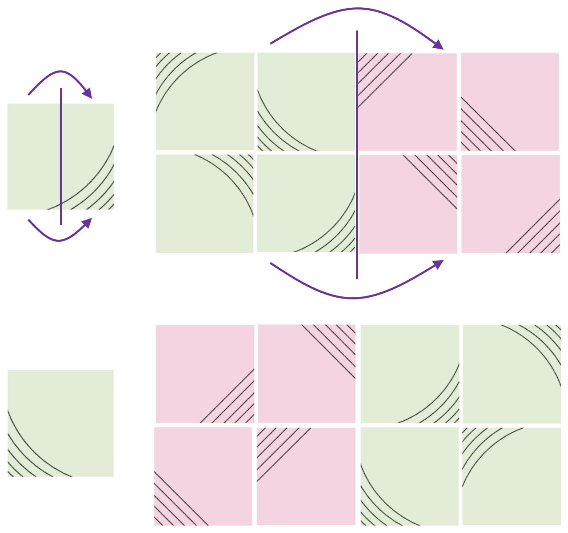
Include a brief comment above each test case justifying why you chose it.

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

4. Flip Service (24 points)

The following parts consist of a mix of written and coding work: parts (b,d,f,g,i) are coding and are submitted with "HW3 Coding", while parts (a,c,e,h) are written and are submitted with "HW3 Written".

In this problem, we will write a function that "flips a quilt horizontally, as if spun around a vertical line through the center". Here is an example (the bottom quilt is the result of horizontally flipping the top quilt):



(a) The problem definition was in English, so our first step is to formalize it.

Let's start by writing a mathematical definition of a function "sflip-horz" that flips an individual square horizontally.

(b) Translate your mathematical definition into TypeScript code in the `quilt_ops.ts` file.

Write tests for your function in `quilt_ops_test.ts`.

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

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

(c) Next, write a mathematical definition of a function "rflip-horz" that flips a row horizontally. You should use sflip-horz in your definition.

Note that this operation not only flips the individual squares horizontally but also *reverses* their order. Remember that the function rconcat, which concatenates two rows, is already provided for you.

(d) Translate your mathematical definition into TypeScript code in the `quilt_ops.ts` file.

Write tests for your function in `quilt_ops_test.ts`.

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

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

(e) Now, write a mathematical definition of a function "qflip-horz" that flips a quilt horizontally. You should use rflip-horz in your definition.

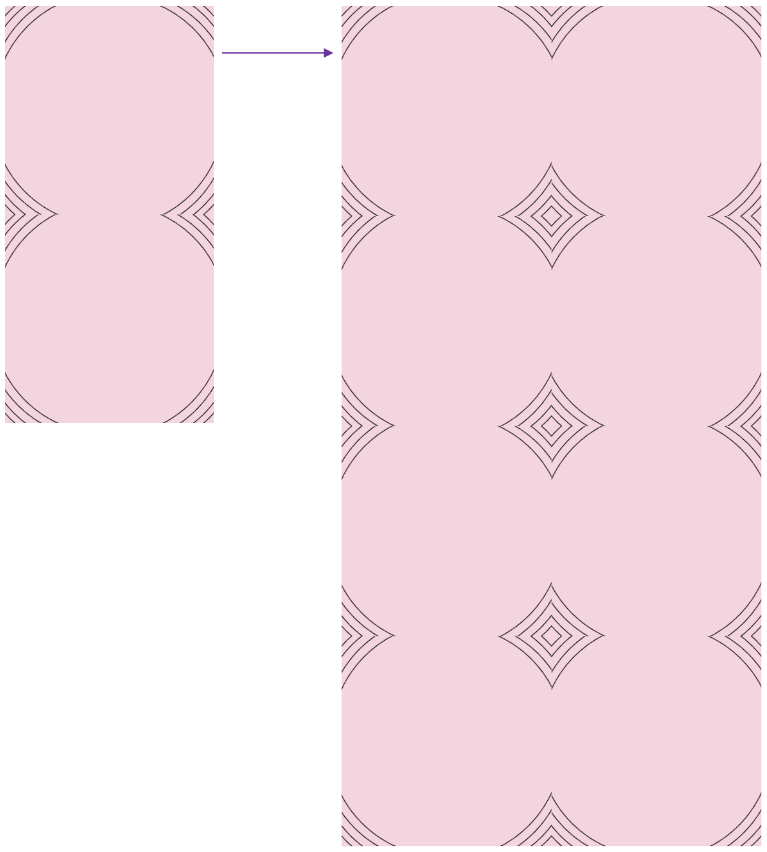
(f) Translate your mathematical definition into TypeScript code in the `quilt_ops.ts` file.

Write tests for your function in `quilt_ops_test.ts`.

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

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

(g) The code provided in `quilt_ops.ts` includes a function called "`symmetrize`" that takes a quilt as an argument and returns one that is symmetric, both horizontally and vertically. It does this by sewing the given quilt and its horizontally-flipped version and then sewing the result together with a vertically flipped version of that. (The code is just two lines. Have a look at it if this description was unclear.)



Modify `index.tsx` to look for a query parameter called "`symmetrize`" and, if it is present, call the `symmetrize` function on the quilt before displaying it. You can see if a parameter is present by calling the "`has`" method on the `URLSearchParams` object. In this case, we call `params.has("symmetrize")`.

You should then be able to see the symmetrized version by adding this parameter to the URL, e.g.: `http://localhost:8080?pattern=D&color=red&rows=4&symmetrize`.

(h) What steps are required to ensure that that this code from part (g) is correct? (Remember the `symmetrize` function is already tested.) Explain your answer, and check that your code is correct by doing those steps.

(i) As a last step to polish off our app and further improve the experience for our users, we will add a `<form>` (like we used in HW2) to allow users to display different quilts without modifying the URL directly.

In `quilt_form.tsx`, we provided a function, `QuiltForm()`, that returns an HTML form with selectors for each quilt customization. In `index.tsx`, replace the contents of the `root.render` in the branch that checks for undefined pattern to instead call this function to display the form (rather than rendering a default PatternA quilt, as we did previouly). Like other functions that produce HTML, `QuiltForm` takes a "_props" parameter, but in this case we don't need any props, so when your function can just pass in an empty record: `{}`.

Congratulations! You have a functioning app.

5. Swap, Drop, and Roll (16 points)

The following parts consist entirely of written work. It should be submitted with "HW3 Written".

In this problem, we will use the following function, which swaps adjacent pairs of elements in a list:

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

Lists of length 0 and 1 are left as is, whereas if the list has length 2 or more, the order of the first two elements are swapped before we recurse on the rest of the list after those two elements.

Suppose you see the following snippet in some TypeScript code. It uses `len` and swap, which are TypeScript implementations of the mathematical functions "len" and "swap".

```
if (len(L) === 3) {
  return cons(1, cons(2, L));  // = swap(swap(cons(1, cons(2, L))))
}
```

The comment shows the definition of what should be returned, but the code is not a direct translation of those. Below, we will use reasoning prove that the code is correct.

(a) Let x be any integer. Prove that $\mathsf{swap}(\mathsf{swap}(\mathsf{cons}(x, \mathsf{nil}))) = \mathsf{cons}(x, \mathsf{nil})$.

(b) Let x and y be any integers and R be any list. Prove that

$$\mathsf{swap}(\mathsf{swap}(\mathsf{cons}(x, \mathsf{cons}(y, R)))) = \mathsf{cons}(x, \mathsf{cons}(y, \mathsf{swap}(\mathsf{swap}(R))))$$

(c) Let $L = \mathsf{cons}(s, \mathsf{cons}(t, \mathsf{cons}(u, \mathsf{cons}(v, \mathsf{cons}(w, \mathsf{nil})))))$ for some integers $s, t, u, v, w : \mathbb{Z}$, i.e., L is some list of length 5. Prove that $\mathsf{swap}(\mathsf{swap}(L)) = L$.

You should apply part (a) once and part (b) multiple times (with different choices of x and y) rather than performing the same calculation again here.

(d) Prove that the code is correct by showing that $\mathsf{swap}(\mathsf{swap}(\mathsf{cons}(1, \mathsf{cons}(2, L)))) = \mathsf{cons}(1, \mathsf{cons}(2, L))$, using the fact that L has length 3, i.e., that $L = \mathsf{cons}(a, \mathsf{cons}(b, \mathsf{cons}(c, \mathsf{nil})))$ for some integers a, b, c.

Feel free to apply prior parts, if useful, rather than performing calculations again.

6. Extra Credit: Cycle-babble (0 points)

The following problem consist entirely of written work. If you complete it, submit it with "HW3 Written".

For this last problem, we define "cycle", which returns a list with each element moved forward one index (e.g., the element at index n moves to index $n-1$), except for the first element, which is moved to the end.

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

where concat is recursively on its first argument by

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

For example, we have $\text{cycle}(\text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil})))) = \text{cons}(2, \text{cons}(3, \text{cons}(1, \text{nil})))$.

Let a be any integer and L be any list. In this problem, you will prove, **without induction**, that we have $\text{len}(\text{cycle}(\text{cons}(a, L))) \geq 1$. Do so in the following steps.

(a) Prove by cases that $\text{cycle}(\text{cons}(a, L)) \neq \text{nil}$.

(b) Use part (a) to prove that $\text{len}(\text{cycle}(\text{cons}(a, L))) \geq 1$. You may need to use the following facts:

- If a list $L \neq \text{nil}$, then we know that $L = \text{cons}(b, R)$ for some $b : \mathbb{Z}$ and $R : \text{List}$.
- We have $\text{len}(L) \geq 0$ for any list L since, by definition, len returns a natural number.

Quilt Patterns

The following mathematical functions PatternA, ..., PatternE, formally define the correct quilt to display with that pattern for different numbers of rows. Specifically, the function call PatternX(n, c) returns the correct quilt with pattern X having n rows with squares in color c. Note that these patterns are defined only for $n \geq 0$ and, in cases C and D, only for even n.

Patterns A & B

We define the A pattern, for any **natural number** of rows, recursively as follows:

$$\begin{aligned} \textbf{func } \mathsf{PatternA}(0, c) &:= \mathsf{qnil} \\ \mathsf{PatternA}(n+1, c) &:= \mathsf{qcons}([s_c, s_c], \mathsf{PatternA}(n, c)) \end{aligned}$$

where c is the color parameter and s_c is the square $\{\mathsf{shape} : \mathsf{ROUND},\ \mathsf{color} : c,\ \mathsf{corner} : \mathsf{NE}\}$. Remember that $[s_c, s_c]$ is shorthand for $\mathsf{rcons}(s_c, \mathsf{rcons}(s_c, \mathsf{rnil})))$. The function call PatternA(n, c) will return a quilt with n rows, with each row having exactly 2 squares.

We define the B pattern, for any **natural number** of rows, recursively as follows:

$$\begin{aligned} \textbf{func } \mathsf{PatternB}(0, c) &:= \mathsf{qnil} \\ \mathsf{PatternB}(n+1, c) &:= \mathsf{qcons}([s_c, t_c], \mathsf{PatternB}(n, c)) \end{aligned}$$

where c is the color parameter and s_c is the square $\{\mathsf{shape} : \mathsf{STRAIGHT},\ \mathsf{color} : c,\ \mathsf{corner} : \mathsf{NE}\}$ and t_c is the square $\{\mathsf{shape} : \mathsf{STRAIGHT},\ \mathsf{color} : c,\ \mathsf{corner} : \mathsf{SW}\}$.

Patterns C & D

We define the C pattern, for any **even number** of rows, recursively as follows:

$$\begin{aligned} \textbf{func } \mathsf{PatternC}(0, c) &:= \mathsf{qnil} \\ \mathsf{PatternC}(n+2, c) &:= \mathsf{qcons}([s_c, t_c], \mathsf{qcons}([u_c, v_c], \mathsf{PatternC}(n, c))) \end{aligned}$$

where the four squares mentioned above are

$$\begin{aligned} s_c &:= \{\mathsf{shape} : \mathsf{ROUND},\ \mathsf{color} : c,\ \mathsf{corner} : \mathsf{SE}\} \\ t_c &:= \{\mathsf{shape} : \mathsf{ROUND},\ \mathsf{color} : c,\ \mathsf{corner} : \mathsf{SW}\} \\ u_c &:= \{\mathsf{shape} : \mathsf{ROUND},\ \mathsf{color} : c,\ \mathsf{corner} : \mathsf{NE}\} \\ v_c &:= \{\mathsf{shape} : \mathsf{ROUND},\ \mathsf{color} : c,\ \mathsf{corner} : \mathsf{NW}\} \end{aligned}$$

Note that PatternC(n, c) also returns a quilt with n rows, with each row containing 2 squares.

We define the D pattern, for any **even number** of rows, the same way:

$$\begin{aligned} \textbf{func } \mathsf{PatternD}(0, c) &:= \mathsf{qnil} \\ \mathsf{PatternD}(n+2, c) &:= \mathsf{qcons}([s_c, t_c], \mathsf{qcons}([u_c, v_c], \mathsf{PatternD}(n, c))) \end{aligned}$$

except that, now, the four squares mentioned above are

$$\begin{aligned} s_c &:= \{\mathsf{shape} : \mathsf{ROUND},\ \mathsf{color} : c,\ \mathsf{corner} : \mathsf{NW}\} \\ t_c &:= \{\mathsf{shape} : \mathsf{ROUND},\ \mathsf{color} : c,\ \mathsf{corner} : \mathsf{NE}\} \\ u_c &:= \{\mathsf{shape} : \mathsf{ROUND},\ \mathsf{color} : c,\ \mathsf{corner} : \mathsf{SW}\} \\ v_c &:= \{\mathsf{shape} : \mathsf{ROUND},\ \mathsf{color} : c,\ \mathsf{corner} : \mathsf{SE}\} \end{aligned}$$

Pattern E

We define the E pattern, for any **natural number** of rows, recursively as follows:

$$\begin{aligned}
\textbf{func}\ \mathsf{PatternE}(0, c) &:= \mathsf{qnil} \\
\mathsf{PatternE}(1, c) &:= \mathsf{qcons}([s_c, t_c], \mathsf{qnil}) \\
\mathsf{PatternE}(n+2, c) &:= \mathsf{qcons}([s_c, t_c], \mathsf{qcons}([u_c, v_c], \mathsf{PatternE}(n, c)))
\end{aligned}$$

where the four squares mentioned above are

$$\begin{aligned}
s_c &:= \{\mathsf{shape} : \mathsf{STRAIGHT},\ \mathsf{color} : c,\ \mathsf{corner} : \mathsf{NE}\} \\
t_c &:= \{\mathsf{shape} : \mathsf{STRAIGHT},\ \mathsf{color} : c,\ \mathsf{corner} : \mathsf{SW}\} \\
u_c &:= \{\mathsf{shape} : \mathsf{STRAIGHT},\ \mathsf{color} : c,\ \mathsf{corner} : \mathsf{SE}\} \\
v_c &:= \{\mathsf{shape} : \mathsf{STRAIGHT},\ \mathsf{color} : c,\ \mathsf{corner} : \mathsf{NW}\}
\end{aligned}$$

Note that we have defined a pattern for any natural number of rows, even though, as in patterns C and D, the odd and even rows are different!