CSE 331: Software Design & Engineering Winter 2025

Homework 7

Due: Wednesday, February 26th, 11pm

Written

Submission

After completing all parts below, submit your solutions as a PDF on Gradescope under "**HW7 Written**".

Don't forget to check that the submitted file is up-to-date with all written work you completed!

Make sure your work is legible and scanned clearly if you handwrite it, or compiled correctly if you choose to use LaTeX. Match each HW problem to the page with your work when you turn in. If your work is not readable or pages are not assigned correctly, you will receive a point deduction.

Task 1 – List and Shout [10 pts]

The following problems involve translation of loops into tail recursive functions.

(a) The following TypeScript function calculates concat(L, R) using a loop.

```
const concat = <A>(L: List<A>, R: List<A>): List<A> => {
  S = rev(L);
  while (S.kind !== "nil") {
    R = cons(S.hd, R);
    S = S.tl;
  }
  return R;
};
```

Define a mathematical definition for a tail-recursive function concat-acc that has identical behavior to the loop body. Then, give a mathematical definition of a function concat that calls concat-acc in such a manner that it matches the overall behavior of the `concat` code.

Don't forget to add type declarations for your functions.

(b) This TypeScript function uses a loop to calculate the value of a descending order polynomial given a list of integer coefficients and a value of x, such that calc$(c_0 :: c_1 :: ... :: c_n ::$ nil$, x) = c_0x^{n-1} + c_1x^{n-2}...c_nx^0$.

```
const calc = (C: List<bigint>, x: bigint): bigint => {
  let s: bigint = 0n;
  while (C.kind !== "nil") {
    s = (s * x) + C.hd
    C = C.tl;
  }
  return s;
};
```

Notice that the length of the coefficients determines the number of polynomial terms, and that each power is strictly 1 less than the previous term. For example, for coefficients $5 :: 2 :: 3 ::$ nil, the polynomial to evaluate would be $5x^2 + 2x^1 + 3x^0 = 5x^2 + 2x + 3$, and with some input $x = 2$, the result would be $5(2^2) + 2(2) + 3 = 27$.

Define (mathematically) a tail-recursive function calc-acc that has identical behavior to the loop. Then, give a new definition of calc that calls calc-acc in such a manner that it matches the overall behavior of the `calc` code.

Don't forget to add type declarations for your functions.

Task 2 – Loops, I Did it Again [15 pts]

In HW6 Task 5, we looked a function "even" that takes a list of base-3 digits, List⟨Digit⟩, and determines if the value they represent is even or odd. Recall the definition of a Digit in base-3:

$$\textbf{type}\ \text{Digit} := 0 \mid 1 \mid 2$$

That function even was defined as follows:

$$\begin{aligned} \text{even}(\text{nil}) &:= \text{true} \\ \text{even}(0 :: \text{ds}) &:= \text{even}(\text{ds}) \\ \text{even}(1 :: \text{ds}) &:= \text{not even}(\text{ds}) \\ \text{even}(2 :: \text{ds}) &:= \text{even}(\text{ds}) \end{aligned}$$

In this problem, we will do the same calculation using tail recursion. To do so, we first define a tail-recursive function, even-acc, of two arguments as follows:

$$\begin{aligned} \text{even-acc}(\text{nil}, b) &:= b \\ \text{even-acc}(0 :: L, b) &:= \text{even-acc}(L, b) \\ \text{even-acc}(1 :: L, b) &:= \text{even-acc}(L, \text{not } b) \\ \text{even-acc}(2 :: L, b) &:= \text{even-acc}(L, b) \end{aligned}$$

That would allow us to *re*-define even by invoking even-acc (for clarity here, we'll call this redefinition "even-new"):

$$\text{even-new}(L) := \text{even-acc}(L, \text{true})$$

giving us a potentially more memory efficient implementation.

(a) Translate the mathematical definitions for even-acc and the definition of even-new into one TypeScript function, even(L), that behaves identically to a tail-call optimized version of these definitions by using a loop.

Be sure to include the invariant of the loop. Your invariant should be in the form shown in class for loops translated from tail recursion and refer to the definition for even-acc. Your code must be correct with that invariant.

(b) Prove the following, which we'll refer to as "equation (1)," by induction

$$\text{even-acc}(L, b) = (\text{even}(L) = b) \tag{1}$$

where the "=" on the right is the usual equals operator on booleans, which is true if both have the same value and false if they have different values.

Note that $(b = \text{true})$ is equivalent to just b, and that $(\text{not } a = b)$ is equivalent to $(a = \text{not } b)$ since both mean $(a \neq b)$.

Hint: You will likely need a proof by cases within your inductive step.

(c) Using equation (1), which we proved holds in the last part, we can now rewrite our invariant without reference to even-acc ("Destroy the evidence"). This is useful to us because we can describe the behavior of our loop without reference to our intermediate tail-optimized version of the math definitions.

Show that $\text{even}(L_0) = (\text{even}(L) = b)$ holds using the original invariant and equation (1).

Task 3 – In One Fell Loop [20 pts]

We have seen the definition of a function contains(L, y) that returns true if the value y appears in the list L. Here is a variation on that function, excludes(L, y) that returns true if the value y *does not* appear in the list L.

$$\text{excludes}(\text{List}\langle\mathbb{Z}\rangle, \mathbb{Z}) \to \mathbb{B}$$

$$\begin{array}{lcll} \text{excludes}(\text{nil}, y) & := & \text{true} & \\ \text{excludes}(x :: L, y) & := & \text{false} & \text{if } x = y \\ \text{excludes}(x :: L, y) & := & \text{excludes}(L, y) & \text{if } x \neq y \end{array}$$

This function (as well as contains) is already tail recursive.

(a) Translate it into a TypeScript function, excludes(L, y), that behaves identically to a tail-call optimized version of these definitions by using a loop.

Be sure to include the invariant of the loop. Your invariant should be in the form shown in class for loops translated from tail recursion, and your code must be correct with that invariant.

(b) It is also possible to define excludes as follows

$$\text{excl}(\text{List}\langle\mathbb{Z}\rangle, \mathbb{Z}) \to \mathbb{B}$$

$$\begin{array}{lcl} \text{excl}(\text{nil}, y) & := & \text{true} \\ \text{excl}(x :: L, y) & := & (x \neq y) \text{ and } \text{excl}(L, y) \end{array}$$

This definition is more concise and uses pattern matching which may make it more obviously correct to someone else reading the code, but since it is not tail recursive, a direct implementation would be less memory efficient. We need to show that our tail-recursive function definition is equivalent to this alternate definition that we'd like to use to document the function instead.

Prove that the following, which we'll refer to as "equation (2),"

$$\text{excl}(L, y) = \text{excludes}(L, y) \tag{2}$$

holds for all lists of integers L and integer values y. Your proof should be by structural induction on L.

Note that, for all boolean values b, we have "false and b = false" and "true and $b = b$".

(c) Using the original invariant and equation (2), rewrite your loop invariant to use excl, instead of the original excludes, and show that it holds.

Coding

To get started, check out the starter code for this assignment:

```
git clone https://gitlab.cs.washington.edu/cse331-25wi/materials/hw7-calculator.git
```

Navigate to the `hw7-calculator` directory and run `npm install --no-audit`. Run tests with the command `npm run test` and run the linter with the command `npm run lint`. You can also run `npm run start` and open `localhost:8080` to see the application for this assignment.

Submission

After completing all tasks to follow, submit your solutions on Gradescope. The following completed files should be submitted to **"HW7 Code"**:

`natural.ts` `natural_test.ts` `Calculator.tsx`

Wait after submitting to make sure the autograder passes, and leave yourself time to resubmit if there's an issue. The autograder will run your tests, additional staff tests, and the linter.

Number Bases

In the coding portion of this assignment, we will work with natural numbers represented as lists of digits, as in HW6 Task 5, but this time in an arbitrary base from 2 to 36. In the code, we represent a number as the following type:

```
type Natural = {digits: List<number>, base: number};
```

In addition to the type listed above, we require that `base` is between 2 and 36 (inclusive) and that every digit in the list is not just any number but specifically a number between 0 and the `base` -1 (inclusive).

Starting at base 11 and continuing up the digits representing 10 and up are no longer numerical, but alphabetical characters like a(10), b(11), c(12), since we run out of single digits after 9. Keep this in mind as you work through the tasks below and begin writing test cases!

When we write a base-10 number like "120", the first digit "1" represents the 1×10^2, so it is the "highest order" digit. Storing the high order digits at the front of a list is a called "big endian" representation.

While the big endian representation matches how we write numbers, it is less convenient for our calculations, so instead, we will store the "lowest order" digits at the front of the list. For example, 120 would be stored as $0 :: 2 :: 1 ::$ nil rather than $1 :: 2 :: 0 ::$ nil. This is called a "little endian" representation, and it is what we will use in `Natural`.

We can formalize the discussion above by defining the value of the base-b digits in "ds" as

$$\begin{aligned} \text{value}(\text{nil}, b) &:= 0 \\ \text{value}(d :: \text{ds}, b) &:= d + b \cdot \text{value}(\text{ds}, b) \end{aligned}$$

This simple, recursive definition encodes the fact that the digits are in the little endian representation. (A recursive function defining the value in the big endian representation would not be so simple!) Also note that "d" is the head of the list of digits "ds".

Note that we are not using `bigint` in this assignment. Instead, we have provided the above definition of `Natural` as an alternative. Also, note that we use `number` in the code to represent digits and bases, both of which should always be integers. In the starter code, we use some TypeScript functions that accept `number` as the parameter type, even though the inputs represent integers, so we use `number` everywhere for simplicity. You can assume digits and bases will be integers in your implementation. We won't test your code on non-integer inputs.

Before continuing with the following tasks, read these notes on "bottom up" recursion which you will use for the loops in Tasks 4 and 5. These are also linked under Topic 7.

Task 4 – Rally The Loops [15 pts]

In this problem, we will implement functions that convert from strings to the `Natural` type above.

(a) Implement the body of the function `convertNatToStr` in `natural.ts`. The function has the following declaration

```
const convertNatToStr = (n: Natural): List<string>
```

Its specification says that, for all lists of digits ds (except nil, a special case), it returns the list of characters rev(digits-to-string(n)), where digits-to-string is defined recursively as

$$\begin{aligned} \text{digits-to-string}(\text{nil}) &:= \text{nil} \\ \text{digits-to-string}(d :: ds) &:= \text{from-digit}(d) :: \text{digits-to-string}(ds) \end{aligned}$$

The reason for returning rev(digits-to-string(n)) rather than digits-to-string(n) is that digits-to-string returns characters in little endian format, but we want to display them in big endian. The reversal switches between the two representations.

You should implement this function with a loop using the "bottom up" template.

Write down a correct invariant for your loop. Remember that we are not writing a loop to calculate `digits-to-string`(n), but rather rev(`digits-to-string`(n)). The invariant must reflect this. Your code must be correct with the invariant you write! (It's not enough to just behave properly when run, another programmer must be able to see why it is correct by reading your comments.)

A function that calculates from-digit is already provided in the code as `fromDigit`.

(b) Write tests for `convertNatToStr` in `natural_test.ts`. Do not move on until you are certain that your code is correct. (Debugging it later on will be much more painful!) Write comments for your tests justifying which test coverage areas they satisfy.

We have provided the reverse of this operation, `convertStrToNat`, which is an implementation of the following recursive definition:

$$\begin{aligned} \text{string-to-digits}(\text{nil}, b) &:= \text{nil} && \\ \text{string-to-digits}(c :: cs, b) &:= \text{to-digit}(c) :: \text{string-to-digits}(cs, b) && \text{if to-digit}(c) < b \\ \text{string-to-digits}(c :: cs, b) &:= \text{undefined} && \text{if to-digit}(c) \geqslant b \end{aligned}$$

We implemented this function using recursion, but we could implement it with a loop using a different recursion to loop template.

Task 5 – Sticks Out Like a Sore Num [20 pts]

In this problem, we will implement a function to add two natural numbers. This function has the following declaration in `natural.ts`:

```
const add = (n: Natural, mat: Natural): Natural
```

This is the core part of our natural number library. The other functions, for multiplying and changing bases, are simple recursions that invoke this add function, provided in the starter code.

We will implement addition of digits as you learned in grade school, moving through the digits from the low order ones to the high order ones, bringing along a "carry" as we go. (This shows, again, why the little endian representation is easier for us.)

We can formalize the algorithm for adding two lists of digits in a fixed base B as follows:

$$
\begin{aligned}
\mathsf{add}(\mathsf{nil},\, \mathsf{nil},\, 0) &:= \mathsf{nil} \\
\mathsf{add}(\mathsf{nil},\, \mathsf{nil},\, 1) &:= 1 :: \mathsf{nil}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{add}(\mathsf{nil},\, b :: bs,\, c) &:= (b + c) :: bs && \text{if } b + c < B \\
\mathsf{add}(\mathsf{nil},\, b :: bs,\, c) &:= (b + c - B) :: \mathsf{add}(\mathsf{nil}, bs, 1) && \text{if } b + c \geqslant B
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{add}(a :: as,\, \mathsf{nil},\, c) &:= (a + c) :: as && \text{if } a + c < B \\
\mathsf{add}(a :: as,\, \mathsf{nil},\, c) &:= (a + c - B) :: \mathsf{add}(as, \mathsf{nil}, 1) && \text{if } a + c \geqslant B
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{add}(a :: as,\, b :: bs,\, c) &:= (a + b + c) :: \mathsf{add}(as, bs, 0) && \text{if } a + b + c < B \\
\mathsf{add}(a :: as,\, b :: bs,\, c) &:= (a + b + c - B) :: \mathsf{add}(as, bs, 1) && \text{if } a + b + c \geqslant B
\end{aligned}
$$

The first two and last two cases are the core of the algorithm. We need the middle cases to handle the fact that the lists of digits may not have the same length. We could handle this by adding some extra zeros to the end of the shorter list, but that is not necessary. The definition above handles those cases by behaving as if there was a 0 digit when one of the two becomes empty before the other.

Like the last task, this function follows the "bottom up" template. Make sure your code is correct with the provided invariant!

(a) Implement the body of the loop so that it implements the function above in its recursive cases. As usual, we should only enter the loop body if the function needs to make a recursive call.

(b) Implement the base cases of the function above after the loop, storing the result in the variable `rs` in the code. The code to put together the list from the base case with the list from the recursive cases, and to reverse the entire result, is already provided.

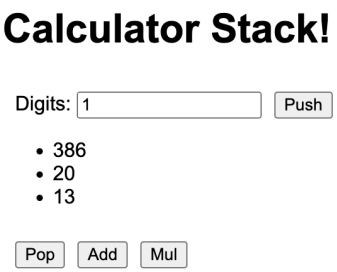
(c) Write tests for `add` in `natural_test.ts`. Do not move on until you are certain that your code is correct. Write comments for your tests justifying which test coverage areas they satisfy.

(d) Uncomment the tests for `scale`, `mul`, and `changeBase` in `natural_test.ts`. These should all pass now that `add` works. (If any of those failed, you'd have to figure out that code just to debug `add`. Ugh, can you imagine!)

Task 6 – Ahead of the Stack [20 pts]

In this task, we will finish an implementation of a stack calculator app that uses your `Natural` numbers. You can run the app with `npm run start`.

In the final app, the main page will be a "calculator" where users can input digits and push them to a stack. Then, it allows users to pop items from the stack and add and multiply the top two items (the result of which is pushed onto the stack in their place).



On the calculator page, there is a gear icon. Clicking on this will open the "Settings" page. The app is already so beautifully designed that the only option we could imagine a user wanting to set is the base of the numbers in the calculator. Any valid base (integers from 2 to 32) can be set here and on "Save" will update all calculator stack items to have the new base.

The code provided in `App.tsx` already allows the user to type in numbers, push them onto the stack, and change the base of the numbers in settings. If you haven't already, run `npm run start` to open up the app and see what you're working with, and read the provided code to make sure you understand how state is managed and how the code is organized.

This is a multi-page app where the `App` component is the parent to the `Calculator` and `Settings` components. It determines which page to display with a `Page` enum, similar to the one we had you create last week. Like last week, these pages can be constructed with just an identifier (there is no other crucial information that defines their construction).

```
type Page = {kind: "calculator"} | {kind: "settings"}
```

The app is missing the ability to add, multiply, and pop numbers from the stack. We will add these features in `Calculator.tsx`.

Note that the pop operation does not make sense when the stack is empty, and the add or multiply operations do not make sense when the stack has fewer than two items. When an operation does not make sense, you can either (1) not give the user a way to invoke it or (2) allow them to invoke it but show an error when the operation is not possible. You are free to choose whichever of these approaches you prefer. However, you cannot crash or silently fail when the operations do not make sense. That could be confusing to the user.

You can check out how error handling is done in `Settings.tsx` as an example of a possible (but not required) method for organizing error messages. `Settings` uses an enum to define whether the status of a user input is "valid" (no known errors) or "invalid". In the invalid case, an error message needs to be presented to the user which the enum helps with by guaranteeing that when an "invalid" state for the component is entered, there is always a corresponding error message.

This enum is defined in `utils.ts`, and you are welcome to import and use it in `Calculator.tsx` if you'd like. It looks like this:

```
export type ParseState = {kind: "valid"} | {kind: "invalid", error: string}
```

You can also give the html containing your error message a `className` to make the text red!

```
className=".error"
```

(a) Implement the body of `renderStackOps` to return some HTML displaying buttons that allow the user to pop the top item from the stack, add the top two items, or multiply the top two items.

(b) Implement event handlers for each of the buttons you created that, when clicked, perform the appropriate operations on the stack.

(c) Update `RenderDigits` and `doPushClick` in `Calculator` to present an error message if a user tries to push a value to the stack that is *invalid* for the set base.

(d) Manually test your UI to make sure that it properly invokes each of the operations.

Since we have tests, we already know that adding, multiplying, and setting bases works properly, so we are just testing that the UI properly invokes them. (It's a good thing too. Can you imagine debugging a problem in the UI that was actually a bug in `add`. Yuck!)