Homework 7

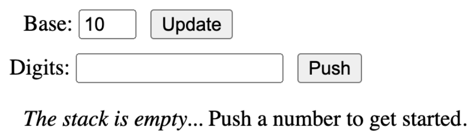
Due: Monday, November 18th, 11pm

If you have not completed HW7 written. Close this file and finish the written component before starting the coding parts in this worksheet.

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

```
git clone https://gitlab.cs.washington.edu/cse331-24au-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. Currently, there will be unused variable Errors, but if you dismiss them with the 'X' you should be able to see the following app. The existing buttons do not yet work as expected, but they will after you implement the first task.



The stack is empty... Push a number to get started.

Submission

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

`App.tsx` `natural.ts` `natural_test.ts`

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.

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 types 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 `base` $- 1$ (inclusive).

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

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

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!)

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 Task 1 and 2. These are also linked under Topic 7.

Task 1 – Rally The Loops [20 pts]

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

1. Implement the body of the function `naturalToString` in `natural.ts`. The function has the following declaration

   ```
   const naturalToString = (nat: 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-str(nat)), where digits-to-str is defined recursively as

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

   The reason for returning rev(digits-to-str(nat)) rather than digits-to-str(nat) is that digits-to-str 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-str`(nat), but rather rev(`digits-to-str`(nat)). 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`.

2. Write tests for `naturalToString` 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 justifiying which test coverage areas they satisfy.

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

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

We implemented this function using recursion, but we could implement it with a loop using a template similar to the "bottom up" template.

Task 2 – 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 = (nat: 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!

1. 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.
2. 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.
3. 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 justifiying which test coverage areas they satisfy.
4. 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 3 – Ahead of the Stack [20 pts]

In this problem, we will finish the implementation of the stack calculator that uses your `Natural` numbers.

The code provided in `App.tsx` already allows the user to type in numbers, push them onto the stack, and to change the base of the numbers on the stack. 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 `App.tsx` code to make sure you understand how state is managed and how the code is organized. The app is missing the ability to add and multiply numbers and to pop them from the stack which we will add in this part.

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.

1. 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.
2. Implement event handlers for each of the buttons you created that, when clicked, perform the appropriate operations on the stack.
3. Manually test your UI to make sure that it properly invokes each of the operations.

   We already know that each of the operations 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!)