CSE 331: Software Design & Implementation Spring 2025

Homework 7

Due: Wednesday, May 21st, 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.

Note: The `<A>` in this function declaration stands for the generic type just like you've seen with lists but the function itself also has `<A>` before the parameters meaning that the types of the L, R and returned list should be the same.

```
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 an integer 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:

$$\text{even: List}\langle\text{Digit}\rangle \to \mathbb{B}$$

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

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:

$$\text{even-acc: List}\langle\text{Digit}\rangle, \mathbb{B} \to \mathbb{B}$$

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

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.

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

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

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

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

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

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/cse331-25sp/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"**:

`stack_ops.ts` `stack_ops_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.

Postfix and Infix Notation

In the following portion of this assignment, we will work with arithmetic expressions (namely a mix of addition and multiplication) in a structure that you have likely not seen before, **postfix** notation.

The typical way that arithmetic expressions are written is in **infix** notation which looks like this:

$$x * y + z$$

where the $x * y$ takes precedence over the $+ z$ due to our precedence rules that are modeled after order of operations. The main difference with postfix is that the operators come after the operands. Since $+$ and $*$ are operators that both take two numbers as operands, we'd structure their postfix notations like $x\ y\ +$ and $x\ y\ *$ respectively. Where x, y are the operands and $+$, $*$ are the operators in their respective expressions.

Note that postfix works without implicit precedence rules since there is only one way to evaluate a postfix expression. So the hard part of going from infix to postfix is understanding how precedence rules work with the infix expression then translating them over to postfix. But, to make sure that order of operations persists in this notation, you need to be careful about the order of your operators and expression in general. Take the infix expression above for example, for it to be valid in postfix notation, the $*$ should happen before the $+$ so this:

$$x\ y * z\ +$$

is correct. But, if you take a different expression like $a + b + c * d$ and encoded it as $a\ b\ +\ c\ +\ d\ *$, that would be incorrect as it would evaluate $a + b$, then add that to c and then finally multiply it by d.

It's helpful to envision the "invisible" parentheses around the expressions showing the order of operations like so:

Infix: $(a + (b + (c * d)))$
Postfix: $(((c\ d\ *)\ b\ +)\ a\ +)$

Calculator Stack

In the app for this week's homework, you'll implement a Calculator Stack that represents the state of integers that can be added or multiplied. Let's formalize this by converting numbers and $+$ or $*$ operations into Token values making up our Stack Abstract Data Type (Notice that we are using our generic Lists to implement the stack, so we'll be able to use some of the familiar list concepts we've been working with).

Suppose we define our Calculator Stack Abstract Data Type as so in TypeScript:

```
type Token = bigint | "+" | "*";

const stack: List<Token> = nil;
```

This ADT represents arithmetic operations in postfix notation (e.g. $[5,\ 2,\ *,\ 3,\ +] = 13$) instead of infix notation (e.g $3 + 2 * 5 = 13$).

Task 4 – Rally The Loops [15 pts]

In this problem, we will implement functions that convert from BigInts to a list of string (which is just each digit).

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

```
const convertBigIntToString = (n: bigint): List<string>
```

Its specification says that, for all n, it returns the list of digit characters rev(bigint-to-string(n)), where bigint-to-string is defined recursively as

$$
\begin{array}{lcll}
\text{bigint-to-string}(n) & := & \text{undefined} & \text{if } n < 0 \\
\text{bigint-to-string}(n) & := & \text{from-digit}(n) :: \text{nil} & \text{if } n < 10 \wedge n >= 0 \\
\text{bigint-to-string}(n) & := & \text{from-digit}(n \ \% \ 10) :: \text{bigint-to-string}(n/10) & \text{otherwise}
\end{array}
$$

The reason for returning rev(bigint-to-string(n)) rather than bigint-to-string(n) is that bigint-to-string returns a list in little endian format, but we want to display them in big endian. The reversal switches between the two representations. [1]

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

Write down a correct invariant for your loop. 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` which handles converting the integer into a string as well as the error handling of making sure we entered a base-10 digit.

(b) Write tests for `convertBigIntToString` in `stack_ops_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, `convertStrToBigInt`, which is an implementation of the following recursive definition:

$$
\begin{array}{lcll}
\text{string-to-bigint}(\text{nil}) & := & 0 & \\
\text{string-to-bigint}(\text{“0”} :: \text{nil}) & := & 0 & \\
\text{string-to-bigint}(c :: cs) & := & \text{to-digit}(c) + \text{string-to-bigints}(cs) * 10 & \text{if } c \neq \text{“0”}
\end{array}
$$

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

[1] Storing higher order digits at the front of a list is called 'big-endian'. For example, 120 is represented as 1::2::0::nil in 'big-endian'. 'Little-endian', or storing lower order digits at the front and higher order digits at the back, is more convenient for our calculations. In this case, 120 would be represented as 0::2::1::nil.

Task 5 – See You Later, Validator [20 pts]

In this problem, we will implement a function to verify the state of our stack ADT. This function has the following declaration in `stack_ops.ts`:

```
const isStackValid = (stack: List<Token>): boolean
```

For this function we want to **rev**erse the postfix stack before we call this function since we want to start at the front of the stack instead of the end meaning that we need to reverse the list so that we see the top last.

The reason we have to reverse the stack is because we start by counting the operands and then if there is ever an operator with less than 2 operands, then it should return false because the stack cannot be evaluated if an operator doesn't have two operands to work with.

We can formalize the algorithm for validating our stack as follows:

is-stack-valid(nil, count)	:=	false	if count < 1
is-stack-valid(nil, count)	:=	true	if count $>= 1$
is-stack-valid($x :: xs$, count)	:=	is-stack-valid(xs, count $+ 1$)	if x is a bigint
is-stack-valid($x :: xs$, count)	:=	false	if x isn't a bigint and count < 2
is-stack-valid($x :: xs$, count)	:=	is-stack-valid(xs, count $- 1$)	if x isn't a bigint and count $\geqslant 2$

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.

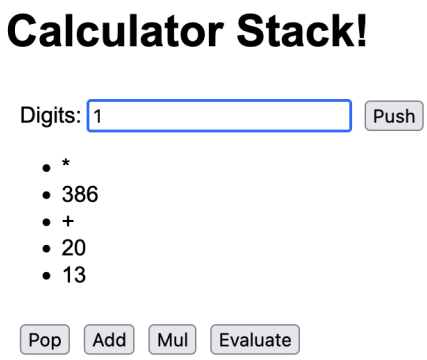
(b) Implement the base cases of the function above after the loop. Think about which cases in the math definition don't have recursive calls and those conditions should be your base cases.

(c) Write tests for `isStackValid` in `stack_ops_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.

Task 6 – Ahead of the Stack [20 pts]

In this task, we will finish an implementation of a stack calculator app. 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 push add and multiply operators to the stack with the add and multiply buttons. The last button, "Evaluate", calls the `evaluateStack` function after verifying that your current stack is valid using the function you implemented in the last part.



The code provided in `App.tsx` already allows the user to type in numbers and push them onto 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 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 $*$ to the stack, and pop numbers from the stack along with the ability to evaluate 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 evaluate operation doesn't make sense if the stack isn't valid. 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.

An example of a possible (but not required) method for organizing error messages is using 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*.

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

Since we have tests, we already know that validating and evaluating 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!)