CSE 331: Software Design & Engineering **Spring 2025**

Sample Final Exam Solutions

Name: ____________________________

UW Email: _______________@uw.edu

This exam contains 18 pages (including this cover page) and 8 problems. Check to see if any pages are missing. Enter all requested information on the top of this page.

Instructions:

- Closed book, closed notes, no cell phones, no calculators.
- You have **1 hour and 50 minutes** to complete the exam. Once time is up, stop immediately.
- Answer all problems on the exam paper, or request scratch paper from a staff member. If any part of an answer is on scratch paper, clearly mark which problem it is for and staple it to the back of your exam.
- The **last 3 pages** contain definitions used in all of the problems. Feel free to separate those pages from the rest of the test (although they must also be turned in at the end).
- If you have a question, please ask. The worst we will say is "we can't answer that".

This page intentionally left blank. Do not put answers here.

Task 1 – The Has and the Has Nots **[4 pts]**

Before you start this problem, be sure to read the **last 3 pages** of the exam, which define the `IntSet` ADT, describe how we will implement it in the `CompactIntSet` class, and defines additional functions. Consider the following code, which implements "`has`" in `CompactIntSet`:

```
// @returns contains(obj, value)
has = (value: bigint): boolean => {
  const b = contains(this.list, value);
  {{ P: b = contains(this.list, value) }}
  {{ Post: b = contains(obj, value) }}
  return b;
}
```

(a) Explain, in 1–2 English sentences, why $\mathbf{Post}$ correctly states what is required for this code to be correct. In particular, why is there no RI included in $\mathbf{Post}$?

This data structure is immutable, and the method does not mutate its fields; we only must show that the RI is true after the constructor.

(b) Prove by calculation that $\mathbf{P}$ implies $\mathbf{Post}$.

$$\begin{aligned} \mathsf{b} &= \mathsf{contains}(\mathsf{this.list}, \mathsf{value}) && \text{from } \mathbf{P} \\ &= \mathsf{contains}(\mathsf{obj}, \mathsf{value}) && \text{by AF} \end{aligned}$$

Task 2 – Add It Up! [20 pts]

Consider the following code, which implements "add" in `CompactIntSet`:

```
// @returns S where contains(S, value) = true
//              and equiv(cons(value, obj), S) = true
add = (value:  bigint):  CompactIntSet => {
  let S: CompactIntSet;
  if (this.has(value)) {
    S = CompactIntSet(this.list);
    {{ P1 : contains(obj, value) = true and S = this.list }}
  } else {
    S = CompactIntSet(cons(value, this.list));
    {{ P2 : contains(obj, value) = false and S = value :: this.list }}
  }
  return S;
}
```

(a) To use the `CompactIntSet` constructor, we must satisfy its precondition. Explain, in English, why its precondition is satisfied in the else branch of the if statement.

> `this.list` is immutable; according to our RI, it has no duplicates. We must show that the argument to the constructor also does not have any duplicates. When we don't take the if statement, we know that `this.list` does not contain `value`. Thus, `cons(value, this.list)` also contains no duplicates.

(b) $\mathbf{P}_2$ was filled in using forward reasoning. Explain briefly, in English, why both facts within $\mathbf{P}_2$ are correct. (Feel free to cite prior questions)

> The first fact holds since we entered the "else" branch and applied the definition of has from Task 1. The second fact follows from the assignment statement and the postcondition of the constructor (which we can use because of part (a)).

(c) Prove that the postcondition holds when the else branch is taken, i.e.

$$\text{contains(S, value)} = \text{true and equiv(value :: obj, S)} = \text{true}$$

given that $\mathbf{P}_2$ holds.

We proceed with two distinct proofs. First, we show that contains(S, value) = true.

$$\begin{aligned} \text{contains}(S, \text{value}) &= \text{contains(value :: this.list, value)} && \text{from } \mathbf{P}_2 \\ &= \text{true} && \text{def of contains} \end{aligned}$$

Next, we show that equiv(value :: obj, S) = true.

$$\begin{aligned} \text{equiv(value :: obj, S)} &= \text{equiv(value :: this.list, S)} && \text{AF} \\ &= \text{equiv(S, S)} && \text{from } \mathbf{P}_2 \\ &= \text{true} && \text{reflexivity of equiv} \end{aligned}$$

(d) Suppose we repeated parts (a-c) for $\mathbf{P}_1$ as well. Explain why that would tell us that the postcondition holds at the end of the code above.

Parts (a-c) showed that the postcondition holds at the end of the else branch. Doing the same with $\mathbf{P}_1$ would tell us that it holds at the end of the then branch. Together, that would show that it holds after the if statement, no matter which branch we take.

(e) Now, suppose that we skipped part (a). In 1–3 sentences, explain what fact we could not use (and how that impacts our analysis).

To use a fact from a function call (i.e. the postcondition), we must show that the input satisfies the precondition. Without part (a), we would not have S = value :: this.list, which we use heavily in part (c).

Task 3 – Everybody Loops [16 pts]

Consider **removeAll** : $(\text{List}\langle\mathbb{Z}\rangle, \mathbb{Z}) \to \text{List}\langle\mathbb{Z}\rangle$, which removes all instances of $s : \mathbb{Z}$ from the list $L : \text{List}\langle\mathbb{Z}\rangle$:

$$\begin{aligned} \text{removeAll}(\text{nil}, s) &:= \text{nil} \\ \text{removeAll}(x :: L, s) &:= \text{removeAll}(L, s) && \text{if } x = s \\ \text{removeAll}(x :: L, s) &:= x :: \text{removeAll}(L, s) && \text{if } x \neq s \end{aligned}$$

Now, consider this tail-recursive definition of **removeAll-acc** : $(\text{List}\langle\mathbb{Z}\rangle, \mathbb{Z}, \text{List}\langle\mathbb{Z}\rangle) \to \text{List}\langle\mathbb{Z}\rangle$

$$\begin{aligned} \text{removeAll-acc}(\text{nil}, s, R) &:= R \\ \text{removeAll-acc}(x :: L, s, R) &:= \text{removeAll-acc}(L, s, R) && \text{if } x = s \\ \text{removeAll-acc}(x :: L, s, R) &:= \text{removeAll-acc}(L, s, x :: R) && \text{if } x \neq s \end{aligned}$$

Consider the following claim, where $s : \mathbb{Z}$ and $L : \text{List}\langle\mathbb{Z}\rangle$:

$$\text{removeAll-acc}(L, s, R) = \text{rev}(\text{removeAll}(L, s)) +\!\!+ R$$

Prove that this claim holds by structural induction <u>on L</u>.
As usual, you should cite the appropriate definitions in the right-hand column of your calculations.

Base Case. (a) Prove that $P(\text{nil})$ holds.

$$\begin{aligned} \text{removeAll-acc}(\text{nil}, s, R) &= R && \text{def of removeAll-acc} \\ &= \text{nil} +\!\!+ R && \text{def of concat} \\ &= \text{rev}(\text{nil}) +\!\!+ R && \text{def of rev} \\ &= \text{rev}(\text{removeAll}(\text{nil}, s)) +\!\!+ R && \text{def of removeAll} \end{aligned}$$

Inductive Hypothesis. Suppose that $\text{removeAll-acc}(L, s, R) = \text{rev}(\text{removeAll}(L, s)) +\!\!+ R$ holds for some list L and some arbitrary $s : \mathbb{Z}$ and list R.

Inductive Step. Let x be an arbitrary integer. Our goal is to show that $P(x :: L)$ holds, i.e., to show that $\text{removeAll-acc}(x :: L, s, R) = \text{rev}(\text{removeAll}(x :: L, s)) +\!\!+ R$. We continue by cases on x...

(b) Prove that $P(x :: L)$ holds when $x = s$.

$$\begin{aligned} \text{removeAll-acc}(x :: L, s, R) &= \text{removeAll-acc}(L, s, R) && \text{def of removeAll-acc} \\ &= \text{rev}(\text{removeAll}(L, s)) +\!\!+ R && \text{by I.H.} \\ &= \text{rev}(\text{removeAll}(x :: L, s)) +\!\!+ R && \text{def of removeAll} \end{aligned}$$

(c) Prove that $P(x :: L)$ holds when $x \neq s$.

$$
\begin{aligned}
\text{removeAll-acc}(x :: L, s, R) &= \text{removeAll-acc}(L, s, x :: R) && \text{def of removeAll-acc} \\
&= \text{rev}(\text{removeAll}(L, s)) +\!\!+ (x :: R) && \text{by I.H.} \\
&= \text{rev}(\text{removeAll}(L, s)) +\!\!+ (x :: (\text{nil} +\!\!+ R)) && \text{def of concat} \\
&= \text{rev}(\text{removeAll}(L, s)) +\!\!+ ((x :: \text{nil}) +\!\!+ R) && \text{def of concat} \\
&= \text{rev}(\text{removeAll}(L, s)) +\!\!+ ([x] +\!\!+ R) && \text{notation} \\
&= (\text{rev}(\text{removeAll}(L, s)) +\!\!+ [x]) +\!\!+ R && \text{associativity of } +\!\!+ \\
&= \text{rev}(x :: \text{removeAll}(L, s)) +\!\!+ R && \text{def of rev} \\
&= \text{rev}(\text{removeAll}(x :: L, s)) +\!\!+ R && \text{def of removeAll}
\end{aligned}
$$

These cases on x are exhaustive, so we have shown that $P(x :: L)$ holds in general.

Conclusion. $P(L)$ holds for all lists L by structural induction.

Task 3 Lemma This also implies that (by substituting $R = \text{nil}$ and using a property of rev),

$$\text{removeAll}(L, s) = \text{rev}(\text{removeAll-acc}(L, s, \text{nil}))$$

You may use this in future sections without additional proof, citing it as "Task 3 Lemma".

Task 4 – Chasing One's Tail In A Loop [24 pts]

Now, we convert our tail-recursive function to code.

(a) Given the following invariant, fill out the implementation of the "`removeAll`" function (using our typical conversion from a tail-recursive function to a loop). Then, fill in $\mathbf{P}_{\text{init}}$ and $\mathbf{P}_{\text{exit}}$ (using forwards reasoning), and $\mathbf{Q}_{\text{exit}}$ (using backwards reasoning).

```
// @returns list X = removeAll(L_0, s)
removeAll = (L: List<bigint>, s: bigint): List<bigint> => {
  let R: List<bigint> = nil;
  {{ P_init : R_0 = nil }}
  {{ Inv : removeAll-acc(L_0, s, R_0) = removeAll-acc(L, s, R) }}
  while (L.kind !== "nil") {
    if (L.hd !== s) {
      R = cons(L.hd, R);
    }
    L = L.tl;
  }
  {{ P_exit : removeAll-acc(L_0, s, R_0) = removeAll-acc(L, s, R) and L = nil }}
  {{ Q_exit : rev(R) = removeAll(L_0, s) }}
  const X: List<bigint> = rev(R);
  {{ Post : X = removeAll(L_0, s) }}
  return X;
}
```

(b) Explain, in 1–3 English sentences, why your choice of $\mathbf{P}_{\text{init}}$ implies $\mathbf{Inv}$.

At this point, $L = L_0$ and $R = R_0 = \text{nil}$, so both sides of the equality are exactly the same.

(c) Prove that your choice of $\mathbf{P}_{\text{exit}}$ implies $\mathbf{Q}_{\text{exit}}$. (Hint: you will want to use the Task 3 Lemma)

$$\begin{aligned}
\text{rev}(R) &= \text{rev}(\text{removeAll-acc}(\text{nil}, s, R)) && \text{def of removeAll-acc} \\
&= \text{rev}(\text{removeAll-acc}(L, s, R)) && \text{from } \mathbf{P}_{\text{exit}} \\
&= \text{rev}(\text{removeAll-acc}(L_0, s, R_0)) && \text{from } \mathbf{P}_{\text{exit}} \\
&= \text{rev}(\text{removeAll-acc}(L_0, s, \text{nil})) && \text{from } \mathbf{P}_{\text{init}} \\
&= \text{removeAll}(L_0, s) && \text{Task 3 Lemma}
\end{aligned}$$

(d) Consider the body of the loop when the if statement is taken.

$\{\{\, \mathbf{Inv}: \text{removeAll-acc}(L_0, s, R_0) = \text{removeAll-acc}(L, s, R) \,\}\}$

```
while (L.kind !== "nil") {
```

  $\{\{\, \mathbf{P_2}: \text{removeAll-acc}(L_0, s, R_0) = \text{removeAll-acc}(L, s, R) \text{ and } L = L.\text{hd} :: L.\text{tl} \,\}\}$

```
  if (L.hd !== s) {
```

    $\{\{\, \mathbf{P_3}: \text{removeAll-acc}(L_0, s, R_0) = \text{removeAll-acc}(L, s, R) \text{ and } L = L.\text{hd} :: L.\text{tl} \text{ and } L.\text{hd} \neq s \,\}\}$

    $\{\{\, \mathbf{Q_3}: \text{removeAll-acc}(L_0, s, R_0) = \text{removeAll-acc}(L.\text{tl}, s, L.\text{hd} :: R) \,\}\}$

```
    R = cons(L.hd, R);
  }
  L = L.tl;
```

  $\{\{\, \mathbf{Q_2}: \text{removeAll-acc}(L_0, s, R_0) = \text{removeAll-acc}(L, s, R) \,\}\}$

```
}
```

After copying your code from part (a), fill in the blank assertions $\mathbf{P_2}$ (from forwards reasoning) and $\mathbf{Q_2}$ (from backwards reasoning).

(e) Prove that your choice of $\mathbf{P_3}$ implies $\mathbf{Q_3}$.

$$\begin{aligned}
\text{removeAll-acc}(L_0, s, R_0) &= \text{removeAll-acc}(L, s, R) && \text{from } \mathbf{P_3} \\
&= \text{removeAll-acc}(L.\text{hd} :: L.\text{tl}, s, R) && \text{from } \mathbf{P_3} \\
&= \text{removeAll-acc}(L.\text{tl}, s, L.\text{hd} :: R) && \text{def of removeAll-acc} \\
& && (\text{as } L.\text{hd} \neq s)
\end{aligned}$$

Task 5 – I Never Would Have Test [8 pts]

Fill in the body of this unit test for `removeAll` so that it meets our coverage requirements.
You do **not** need to provide explanations for your choice of test inputs. You are encouraged to use the `nil` and `cons` list helpers and the function `assert.deepStrictEqual`.

```
it('removeAll', () => {


  assert.deepStrictEqual(
    removeAll(nil, 42n),
    nil
  );

  assert.deepStrictEqual(
    removeAll(cons(42n, nil), 42n),
    nil
  );

  assert.deepStrictEqual(
    removeAll(cons(23n, nil), 42n),
    cons(23n, nil)
  );

  const fib: List<bigint> = cons(1n, cons(1n, cons(2n, cons(3n, cons(5n, nil)))));

  assert.deepStrictEqual(
    removeAll(fib, 1n),
    cons(2n, cons(3n, cons(5n, nil)))
  );

  // Bonus!
  assert.deepStrictEqual(
    removeAll(fib, 42n),
    fib
  );


});
```

Task 6 – I Like To Remove It, Remove It [8 pts]

Consider extending the `CompactIntSet` with a new method, "`removeMultiple`". This method takes in a list of `bigint`s and removes each of them from the set, in order. It is described by the math function **removeMultiple** : $(\text{List}\langle\mathbb{Z}\rangle, \text{List}\langle\mathbb{Z}\rangle) \to \text{List}\langle\mathbb{Z}\rangle$

$$\begin{aligned} \text{removeMultiple}(L, \text{nil}) &:= L \\ \text{removeMultiple}(L, x :: R) &:= \text{removeMultiple}(\text{removeAll}(L, x), R) \end{aligned}$$

Fill in an implementation of `removeMultiple` *including* a corresponding loop invariant (as a comment), using the tail-recursion-to-loop conversion we have discussed in class.

You may use any functions that have been defined in previous tasks. You do not need to prove that this implementation (or this invariant) is correct.

```
// @returns removeMultiple(obj, R)
removeMultiple = (R: List<bigint>): CompactIntSet => {


  let L: List<bigint> = this.list;
  // Inv: removeMultiple(L_0, R_0) = removeMultiple(L, R)
  while (R.kind !== "nil") {
    L = removeAll(L, R.hd);
    R = R.tl;
  }

  return new CompactIntSet(L);
}
```

Task 7 – Alternative Factories [12 pts]

Finally, we consider the factory function for our implementation of `CompactIntSet`. As a reminder, you will want to look at the definition of the constructor in the last 3 pages of this exam.

(a) Consider this potential factory function for `CompactIntSet`:

```
const nilCompactIntSet = new CompactIntSet(nil);

const makeCompactIntSetA = (): IntSet => {
  return nilCompactIntSet;
}
```

In 1–3 sentences, explain why this factory function is correct (even in the presence of aliasing).

> The constructor for `CompactIntSet` has a precondition (that the list has no duplicates). This is true for the `nil` list! Furthermore, returning an alias to the same global `nilCompactIntSet` is okay as the data structure is immutable.

(b) Consider an alternative potential factory function for `CompactIntSet`:

```
const makeCompactIntSetB = (list: List<bigint>): IntSet => {
  return new CompactIntSet(list);
}
```

In 1–3 sentences, explain why this factory function is incorrect.

> The constructor for `CompactIntSet` has a precondition — that there are no duplicates — but we do not check this precondition.

(c) Write a new, correct implementation of makeCompactIntSetB that fixes the above errors. You may not change the header or precondition. You are not required to prove its correctness, and you do not need to provide a loop invariant.

```
const makeCompactIntSetB = (list: List<bigint>): IntSet => {


  let L: List<bigint> = nil;
  while (list.kind !== "nil") {
    L = cons(list.hd, L);
    list = removeAll(list.tl, list.hd);
  }
  return new CompactIntSet(rev(L));
}
```

Task 8 – Decisions, Decisions, Decisions... [8 pts]

Answer each of the following short-answer / multiple-choice questions.

(a) Your friend (enemy?) Matt is debugging a full-stack web application similar to HW8, where files are edited on the client and saved on the server. When he clicks "save file", he says that no change occurs on the page. However, when he refreshes the page, the file change "magically" appears (and is correct). Mark *all* boxes next to explanations that could apply.

- [ ] There is no click event handler registered on the save button.
- [ ] A request is sent, but the server has no route handler for that URL.
- [x] The server has a bug that causes it to crash before completing the save request.
- [x] The server saves the file, but sends back a malformed response to the client.
- [x] The server saves the file and returns a response, but the client's response handler has a bug that causes it to crash before updating the state.
- [ ] The server saves the file and returns a response, the client's state is updated properly, but Matt forgot to call `render` after updating the state.

(b) Consider the following four assertions:

- A: L is a list of one or more integers
- B: L is a list that contains only even integers
- C: L is a list of one or more integers and x is non-negative
- D: x is non-negative

Now, consider the following claims relating those assertions. Mark *all* of the claims that are true.

- [ ] A is stronger than B
- [ ] B is stronger than A
- [x] C is stronger than A
- [ ] C is stronger than B
- [x] C is stronger than D

(c) We discussed three key tenets of Floyd logic to show that programs with loops meet a specification:

   (a) initialization (the precondition implies the invariant)
   (b) preservation (the loop body preserves the invariant)
   (c) exit (the invariant and the loop exit condition imply the post condition)

   Is this sufficient to prove that a program with a loop is correct? Why or why not?

   No. We must also show that the loop terminates, which none of these proofs do. However, the combination of termination and these three proofs show that a loop is correct.

(d) In 1–3 sentences (or with a short code example), explain one case where complete statement coverage does not imply complete branch coverage.

   The typical case is when an if statement (or if-else-if chain) doesn't end with an "else", also called an "implicit else". The implicit else doesn't have a line of code (so there is no statement to cover), but the else branch itself still needs to be tested.

(e) In 1–3 sentences, explain a weakness of tail-call optimization.

   Tail-call optimization removes calls from the call stack, which makes debugging harder.

(f) In Java, the adaptor design pattern is common for type interoperability, especially across libraries. Is the adaptor pattern similarly helpful in TypeScript? Why or why not?

   No, as TypeScript uses structural typing, not nominal typing (what Java uses). Types with the same fields and methods in TypeScript already interoperate.

(g) In 1–3 sentences, explain a weakness of constructors compared to factory functions.

   All constructors must have the same name, even if there are multiple versions with a different number of arguments (and thus, different behaviours). Factory functions can differentiate different versions with different names.

(h) Recall that in JavaScript, the == operator has some unexpected behaviour with false:

```
false == 0   // true
false == "0" // true
false == ""  // true
false == " " // true
```

   Which property of an equality definition does == lack? Explain briefly using the above values.

   == is not transitive: "0" == false and false == " ", but "0" !=" "

ADT Specification

In these problems, we will implement the following `IntSet` ADT. While there are many ways to specify a set, for simplicity's sake we will treat a set as a $\text{List}\langle\mathbb{Z}\rangle$. The ADT is defined in TypeScript as follows:

```
/*  obj is a list of integers */
interface IntSet {
    /**
      * Checks if the given value is in the set (the list obj).
      * @returns contains(obj, value)
      */
    has: (value: bigint) => boolean;

    /**
      * Adds the given value to the set (the list obj).
      * @returns S where contains(S, value) = true
      *             and equiv(value :: obj, S) = true
      */
    add: (value: bigint) => IntSet;

    /**
      * Removes the given value from the set (the list obj).
      * @requires contains(obj, value) = true
      * @returns S where contains(S, value) = false
      *             and equiv(obj, value :: S) = true
      */
    remove: (value: bigint) => IntSet;
}
```

This specification relies on **equiv** : $(\text{List}\langle\mathbb{Z}\rangle, \text{List}\langle\mathbb{Z}\rangle) \to \mathbb{B}$, which returns true if (and only if) the two lists contain the same items (independent of ordering or duplicates). This appears in the postcondition of add and remove to ensure that the operation only changes the set in a certain way (without it, remove could just always return the empty set). The function is defined by:

$$\text{equiv}(L_1, L_2) \quad := \quad \text{subset}(L_1, L_2) \wedge \text{subset}(L_2, L_1)$$

equiv is defined in terms of **subset** : $(\text{List}\langle\mathbb{Z}\rangle, \text{List}\langle\mathbb{Z}\rangle) \to \mathbb{B}$, which returns true if (and only if) every element of the first list is also present in the second list. It is defined by:

$$\begin{aligned}
&\text{subset}(\text{nil}, L_2) && := \text{true} && \\
&\text{subset}(x :: L_1, L_2) && := \text{false} && \text{if contains}(L_2, x) = \text{false} \\
&\text{subset}(x :: L_1, L_2) && := \text{subset}(L_1, L_2) && \text{if contains}(L_2, x) = \text{true}
\end{aligned}$$

Importantly, **subset** and **equiv** are *reflexive*, i.e. for any list L,

$$\text{subset}(L, L) = true$$
$$\text{equiv}(L, L) = true$$

You may use these identities without proof, but you should cite them as "reflexivity of ___".

ADT Implementation

There are many ways to implement the `IntSet` ADT. For this exam, we will implement it with the following class, `CompactIntSet`. It stores the set as a linked list:

```
class CompactIntSet implements IntSet {
  // AF: obj = this.list
  // RI: noDuplicates(this.list) = true
  list: List<bigint>
```

Importantly, the representation invariant enforces that there are no duplicates in the field this.list. This RI is described by noDuplicates(List$\langle\mathbb{Z}\rangle$) $\to \mathbb{B}$, which is defined by:

$$\begin{array}{lll} \text{noDuplicates(nil)} & := & \text{true} \\ \text{noDuplicates}(x :: L) & := & \text{noDuplicates}(L) \quad \text{if contains}(L, x) = \text{false} \\ \text{noDuplicates}(x :: L) & := & \text{false} \qquad\qquad\quad\;\; \text{if contains}(L, x) = \text{true} \end{array}$$

In addition to the methods required by `IntSet`, the `CompactIntSet` class also includes the following constructor:

```
  /**
    * makes obj = list
    * @requires: noDuplicates(this.list) = true
  */
  constructor(list: List<bigint>) {
      this.list = list;
  }
```

The List type is implemented as a record type exactly as seen in lecture, section, and the homework:

```
type List<A> =
  | { readonly kind: "nil" }
  | { readonly kind: "cons", readonly hd: A, readonly tl: List<A> };
```

You may assume that the `nil` and `cons` helpers are defined as follows:

```
const nil: { kind: "nil" } = { kind: "nil" };

const cons = <A> (hd: A, tl: List<A>): List<A> => {
  return { kind: "cons", hd: hd, tl: tl };
};
```

Familiar List Functions

The function **len** : $\mathsf{List}\langle A\rangle \to \mathbb{N}$, which returns the length of a list, is defined by

$$\begin{aligned} \mathsf{len}(\mathsf{nil}) &:= 0 \\ \mathsf{len}(x :: L) &:= \mathsf{len}(L) + 1 \end{aligned}$$

The function **concat** : $(\mathsf{List}\langle A\rangle, \mathsf{List}\langle A\rangle) \to \mathsf{List}\langle A\rangle$, which takes two lists L and R and returns a single list with L followed by R (and is also abbreviated "$L + R$"), is defined as

$$\begin{aligned} \mathsf{concat}(\mathsf{nil}, R) &:= R \\ \mathsf{concat}(x :: L, R) &:= x :: \mathsf{concat}(L, R) \end{aligned}$$

You may assume (without proof) that $+$ is *associative*, i.e. $(a + b) + c = a + (b + c)$.

The function **rev** : $\mathsf{List}\langle A\rangle \to \mathsf{List}\langle A\rangle$, which returns the same numbers but in reverse order, is given by

$$\begin{aligned} \mathsf{rev}(\mathsf{nil}) &:= \mathsf{nil} \\ \mathsf{rev}(x :: L) &:= \mathsf{rev}(L) + [x] \end{aligned}$$

The function **contains** : $\mathsf{List}\langle \mathbb{A}\rangle \to \mathbb{B}$, returns if the list contains the second argument. It is defined by:

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

The function **at** : $(\mathsf{List}\langle A\rangle, \mathbb{Z}) \to (\mathbb{Z} \cup \{\mathsf{missing}\})$ takes a list L and an index j and returns the value at index j of L. It is abbreviated as "$L[j]$" and is defined by

$$\begin{aligned} \mathsf{at}(\mathsf{nil}, j) &:= \mathsf{missing} & & \\ \mathsf{at}(x :: L, j) &:= \mathsf{missing} & &\text{if } j < 0 \\ \mathsf{at}(x :: L, j) &:= x & &\text{if } j = 0 \\ \mathsf{at}(x :: L, j) &:= \mathsf{at}(L, j - 1) & &\text{if } 0 < j \end{aligned}$$