Homework 1: DSSL2 Warmup

The purpose of this assignment is to get you programming fluently in DSSL2, the language that we'll be using for the course.

On Blackboard, you will find starter code (`warmup.rkt`) including some definitions, headers for the methods and functions that you'll need to write, along with an insufficient number of tests.

Installing DSSL2

To complete this homework assignment, you will first need to install the DrRacket programming environment, version 8.7 (available from `racket-lang.org`). Then you will need to install the DSSL2 language within DrRacket.

Once you have DrRacket installed, open it and choose "Package Manager" from the "File" menu. Type `dssl2` as the source, then click the "Install" button and wait for installation to finish. When it's finished, the "Install" button should change to "Update"; then close the window.

Class practice

In `warmup.rkt` we provide a partial definition for the class `Account`, which represents bank accounts. Accounts include

- an `id`entifier, which is a natural number that uniquely identifies the account,
- an account `type`, which can be either `"checking"` or `"savings"`,
- and a `balance`, which must be a non-negative number.

We also provided the constructor, which checks the above constraints of `type`s and initial `balance`s, as well as getter methods for the fields.

The `Account` class also has three additional methods, which you must write:

1. `Account.deposit(num?) -> NoneC`, which adds an amount to the balance of the account. The amount deposited must be non-negative, otherwise your code should call `error` instead.
2. `Account.withdraw(num?) -> NoneC`, which subtracts the given amount from the balance. If the requested withdrawal exceeds the balance, this method must raise an error. The amount withdrawn must be non-negative, otherwise your code should call `error` instead.
3. `Account.transfer(num?, Account?) -> NoneC` withdraws the given number from this account's balance and deposits it in the given account. The amount transferred must be non-negative, and must not exceed the first account's balance, otherwise your code should call `error` instead.

These methods must all return `None`, as indicated by the `NoneC` contracts.

You will also want to write additional tests for the above methods.

Customers

The next set of functions you will write work with vectors of `customer` structs. Customer structs have two fields, `name` which should be a string, and `bank_account` which should be an `Account` built using the `Account` class described above. Multiple `customer`s can have the same name.

5. `max_account_id(VecC[customer?]) -> nat?` takes a vector of customers, and returns the highest account `id` found in any of the given customers' accounts. This function must raise an error when no customers are given.
6. `open_account(str?, account_type?, VecC[customer?]) -> VecC[customer?]` takes the name of a customer, an account type, and a vector of customers, and produces a new vector of customers with a new customer added. That new customer must have the provided name, and have a newly created account with the given type, a balance of 0, and an `id` one higher than the highest previously-used id. If this new account would be the first account in the ledger, start with an `id` of 1.
7. `check_sharing(VecC[customer?]) -> bool?` takes a vector of customers, and checks whether any of the customers in the vector have accounts with identical `id`. If that's the case, return `True`, otherwise return `False`.

As before, you will want to write additional tests for the above operations.

Grading

Please submit your completed version of `warmup.rkt`, containing:

- definitions for the methods and functions described above,
- sufficient tests to be confident of your code's correctness,
- and the honor code.

Functional Correctness

We will use four separate test suites to test your submission:

- **Basic account**: `Account.deposit` and `Account.withdraw` on positive inputs.
- **Advanced account**: `Account.deposit` and `Account.withdraw` edge and error cases, `Account.transfer` (including edge and error cases).
- **Basic customer**: `max_account_id` and `open_account`, including edge and error cases.
- **Advanced customer**: `check_sharing`, including edge and error cases.

To get credit for a test suite, your submission must pass *all* its tests.

The outcome your submission will earn will be determined as follows:

- **Got it**: passes all four test suites.
- **Almost there**: passes both basic test suites and fails a single advanced test suite.
- **On the way**: either passes both basic test suites, or passes both the basic and advanced test suite for one of the two topics.
- **Not yet**: does not achieve "on the way" requirements.
- **Cannot assess**: we could not successfully run our grading tests on your submission (usually due to a crash), which also means *we could not give you feedback*. **Please run your code before submitting!**

Non-Functional Correctness

For this assignment, the self-evaluation will be specifically looking for:

- Thorough testing, including edge cases
- Rigorous checking of error cases
- Reuse of code (*i.e.*, not copy-paste) where pertinent