CpSc 418 **Midterm – Take Home Part** Due: February 23, 2023, 11:59pm

100 points

Prelude

Please submit your solution using:
`handin cs-418 midterm-home` Your solution should contain two files:

`midterm.erl`: Your solutions to coding questions.

`midterm.pdf`: Your solutions to written questions.

The instructors solution to homework 2 along with more test cases are available at:
http://www.students.cs.ubc.ca/~cs-418/2022-2/exams/midterm-1/midterm.html.

The tests in `midterm_tests.erl` are not exhaustive, but better than the `hw2_tests.erl` that I provided with Homework 2. If any of you submit solutions with weird bugs not caught by these tests, I'll add more tests.

As always your code should compile without errors or warnings, and your functions should only print stuff that is explicitly requested in the questions – in other words, nothing.

Your midterm score will be the average of the take-home and in-class components. Both will be graded on a scale of 100. Bug bounties will be in effect for both: bug bounties for the take-home component will be awarded in homework points; bug-bounties for the in-class component will be awarded in midterm points.

The Questions

0. **Collaboration!** (5 points): Collaboration is *required* for this take-home midterm. Discuss your code and the solution with at least one other student in the class. Read their reflection on their code; read their old and new code; and give them feedback. I realize that due to the break week, some of you may be out of town, and this can make collaboration difficult. If you really can't find another student to collaborate with, please come to office hours. I will hold my regular office hours, and run a zoom session during them – that's Monday from 12–1pm. If you're still stuck, let me know, and I'll either set-up another office hour, or arrange an in-person or zoom meeting.

   List your collaborators here. You can give a list of names or CWLs.
   Note: while collaboration is required, the code you submit and your write-ups must be written by you and only you. The collaboration is to get feedback and ideas.

1. **Scan Reflection** (25 points): Based on the code you submitted for homework 2, comparing with the solution set code, and interactions with your collaborators, what did you learn?

   - It is possible that your code is already as good or better than the solution set. If it's better, confirm that with your collaborators, and tell me why – a good way to get a high score. If it's "as good" note any differences; of you might just say that "We came to pretty much the same solution, and the differences are minor, such as...", and give two or three examples of minor differences.
   - If you did not complete HW2, or didn't have a solution or partial solution to HW2, Q3, then contact me. Some of your classmates have generously given me permission to give you an anonymized version of their code, and I will send you such code as your starting point. You will need to read their code, compare it with the solution, and write a reflection and comparision based on that.

2. **Scan Time Warp** (10 points): If you could give three, short, simple hints to a clone of you who was about to start the `scan_bu` (or `reduce_bu`) problem, what would they be? Each of your three hints should be short – preferably less than 20 words. I may take off points if any of the three is longer than 50 words, and you can be sure that I'll take off points if they are all longer than 50 words.

3. **Scan Revision** (20 points): Given your reflection and hints, revise you implementation of `scan_bu`. Your solution will be graded on:

   - Correctness – it should pass all the tests in the new `midterm_tests.erl`. I will read all solutions. If I spot errors or think of new things to test, I will add more tests.
   - Clarity – your code should be easy to read. The structure of the code should reveal the structure of the computation. A few short comments may help. If you need a longer explanation, write a block-comment before the function.
   - Style – write it in Erlang. Don't write it in some dialect of C or Java that you've abused into being accepted by the Erlang compiler.

4. **Scan Extension** (20 points): Consider `scan_bu(ProcState, AccIn, LeafVal, Combine)`. The value `AccIn` is the starting value for the accumulation by the entire tree (a.k.a. the value of "everything to the left" of the tree). When the root is reached, we have computed the `GrandTotal` for the entire tree. Consider scenarios where we would like to do something with `GrandTotal` – I'll provide two such examples in Question 5. In this case we could let `AccIn` be a *function* and evaluate `AccIn(GrandTotal)` to get the value to use at the start of the downward pass of the scan.

   Implement `scanx_bu(ProcState, AccIn, LeafVal, Combine)` such that if `AccIn` is a function, then the value used at the beginning of the downward pass is `AccIn(GrandTotal)`. Otherwise, the value used at the beginning of the downward pass is `AccIn`. Other than this added functionality, `scanx_bu` should behave like `scan_bu`. The Erlang built-in-function `is_function/1` or `is_function/2` will help you determine which kind of `AccIn` you have.

5. **Extended Examples** (20 points):

   (a) (10 points) Use your `scanx_bu` to implement `cumsum_and_notify(W, SrcKey, DstKey)` where:

   `W` is a worker tree from `red:create(...)`.

   `SrcKey` is a key that each worker uses to retrieve a list from its `ProcState`. This list is distributed across all of the workers of `W`. The workers compute the cumulative sum of the elements of this distributed list using `scanx_bu`, of course. Furthermore, the `AccIn` function returns `0` (the starting element for the cumulative sum) and sends `GrandTotal` to the master process (i.e. the process that called `cumsum_and_notify`), with suitable tags to avoid getting messages confused. The `cumsum_and_notify` function receives the message with the `GrandTotal` and returns `GrandTotal`.

   `DstKey` is a key that each worker uses to store its segment of list for the cumulative sum. Each worker has the segment of the cumulative sum corresponding to its segment of the original list.

   Here's an example. Let's W has four workers, and

```
W0:  red:get(ProcState, src) -> [1, 2, 3]
W1:  red:get(ProcState, src) -> [4, 5]
W2:  red:get(ProcState, src) -> [6, 7, 8, 9]
W3:  red:get(ProcState, src) -> [14, -23, 2.5, 11.5]
```

   Then, a call `cumsum_and_notify(W, src, dst)` will return 50.0, and

```
W0:  red:get(ProcState, dst) -> [1, 3, 6]
W1:  red:get(ProcState, dst) -> [10, 15]
W2:  red:get(ProcState, dst) -> [21, 28, 36, 45]
W3:  red:get(ProcState, dst) -> [59, 36, 38.5, 50.0]
```

In addition to using `scanx_bu`, `cumsum_and_notify` will `red:broadcast/2` – at least my version does!

(b) (10 points) Use your `scanx_bu` to implement `normalized_cumsum(W, SrcKey, DstKey)` where:

`W` and `SrcKey` are as describe for Question 5a.

Goal: If `Src` is the list that is distributed across the workers of `W` and associated with `SrcKey`, then let

$$\texttt{lists:nth(K, Dst)} \quad = \quad \frac{1}{\texttt{GrandTotal}} \sum_{\texttt{I=0}}^{\texttt{K}} \texttt{lists:nth(I, Src)}$$

`DstKey` is a key that each worker uses to store its segment of `Dst`. Each worker has the segment of the `Dst` corresponding to its segment of the original list.

The implementation of `normalized_cumsum` doesn't send the `GrandTotal` back to the `normalized_cumsum`. Instead, it passes back down the tree a *pair* of values: the total of everything to the left of the current subtree, and `GrandTotal`. For example, my solution uses a tuple: `{LeftTotal, GrandTotal}`. You can do the same, or you can use a different representation if you think it works better. You will need to modify your `Combine` function as well. I was going to give more hints, but I'll encourage you to collaborate, post piazza questions, etc.

With the same example for the list associate with the key `src` as in Question 5a, we could run `normalized_cumsum(W, Src, Dst2)` to reach a state where

```
W0:  red:get(ProcState, dst) -> [0.02, 0.06, 0.12]
W1:  red:get(ProcState, dst) -> [0.2, 0.3]
W2:  red:get(ProcState, dst) -> [0.42, 0.56, 0.72, 0.8]
W3:  red:get(ProcState, dst) -> [1.12, 1.72, 1.77, 1.0]
```

with the usual disclaimer about round-off error.