CpSc 418 # Homework 2 Due: Oct. 9, 2018, 11:59pm
Early Bird: Oct. 7, 2018, 11:59pm

105 points.

Please submit your solution using the handin program. Submit your solution as
cs418 hw2
Your submission should consist of three files:

- hw2.erl: Erlang source code for the coding parts your solution.
- hw2.pdf PDF for the written response parts of your solution and the plots.
- hw2_tests.erl: EUnit tests for your solution.

Templates for hw2.erl and hw2_tests.erl are available at
http://www.ugrad.cs.ubc.ca/~cs418/2018-1/hw/2/code.html.

The tests in hw2_tests.erl are not exhaustive. If your code doesn't work with these, it will almost certainly have problems with the test cases used for grading. The actual grading will include other test cases as well.

Please submit code that compiles without errors or warnings. If your code does not compile, we might give you zero points on all of the programming problems. If we fix your code to make it compile, we will take off lots of points for that service. If your code generates compiler warnings, we will take off points for that as well, but not as many as for code that doesn't compile successfully.

We will take off points for code that prints results unless we specifically asked for print-out. For this assignment, the functions you write should return the specified values, but they should not print anything to stdout. Using lists:format when debugging is great, but you need to delete or comment-out such calls before submitting your solution. Printing an error message to stdout when your function is called with invalid arguments is acceptable but not required. Your code must fail with some kind of error when called with invalid arguments.

1. Calculate with ProcState. (**24 points**)
   In the September 10 lecture, we gave the following code for an accumulator process:

```
acc_proc(Tally)
    when is_number(Tally) ->
  receive
    N when is_number(N) ->
      acc_proc(Tally+N);
    {Pid, total} ->
      Pid ! Tally,
      acc_proc(Tally);
    exit -> Tally
  end.

accumulator() ->
  spawn(fun() ->
    acc_proc(0)
  end).
```

   (a) Adding more than just adding to the calculator. (**8 points**)
   Rather than having a simple accumulator that can only add, we would like to have a general purpose calculator. Our calculator will be a process that responds to messages with the patterns shown below:

   {plus, X}: Add X to the current Tally. We allow X to be an Erlang number, that means either an integer or floating point number.

`{minus, X}`: Subtract `X` from the current `Tally`.

`{times, X}`: Multiply the current `Tally` by `X`.

`{divide, X}`: Divide the current `Tally` by `X`.

`{set, X}`: Set the `Tally` to `X`.

`{tally, Pid}`: Send a message of the form `{tally, self(), Tally}` to `Pid`. Continue with the current `Tally`.

`exit`: The calculator process terminates.

The function call `hw2:calculator(q1a)` creates the calculator process. I have written a skeleton of the `q1a` function. Your task is to complete the `q1a` function and add a few test cases to `hw2_tests.erl`. Here's a simple test case:

```
1> Calc = hw2:calculator(q1a).
<0.65.0>
2> Calc !  {plus, 3}.
{plus, 3}
3> Calc !  {times, 14}.
{times, 14}
4> hw2:fetch(Calc).
42
5> Calc !  {minus, 10}.
{minus, 10}
6> Calc !  {divide, 4}.
{divide, 4}
7> hw2:fetch(Calc).
4
8> Calc !  exit.
exit
9> hw2:fetch(Calc).
<0.71.0>:  receive timed out
<0.71.0>:  I was looking for
<0.71.0>:    {tally, <0.65.0>, V}
<0.71.0>:  My pending messages are
<0.71.0>:    no more messages
<0.71.0>:  giving up.
failed
```

(b) Remembering to remember. (**8 points**)
We would like our calculator to be able to store and recall values. We will do this by adding a `ProcState` parameter to the calculator. `ProcState` should be an Erlang key-list. See lists:keyfind.
We will add one new command to our calculator: `{store, Key}`, where `Key` must be an Erlang atom. This adds an entry to `ProcState` that makes a mapping from `Key` to the current accumulator value. Furthermore, we will modify the arithmetic commands, `plus`, `minus`, `times`, `divide`, and `set` so that the argument can be an atom. In this case, the calculator looks up the value of associated with the atom from `ProcState`. Here's an example where I use the calculator to evaluate `(2 + 3)*(11 - 4)`:

```
10> f(Calc).  % forget the q1a calculator
ok
11> Calc = hw2:calculator(q1b).
<0.112.0>
12> Calc !  {set, 11}.
{set,11}
13> Calc !  {minus, 4}.
{minus,4}
14> Calc !  {store, a}.
{store,a}
15> Calc !  {set, 2}.
{set,2}
16> Calc !  {plus, 3}.
{plus,3}
17> Calc !  {times, a}.
{times,a}
```

```
31> hw2:fetch(Calc).
35
18> hw2:fetch(Calc, debug).
[a,7]
```

I have written a skeleton of the `q1b` function and its helper `q1b_op`. Your task is to complete the `q1b` and `q1b_op` functions and add a few test cases to `hw2_tests.erl`.

(c) Fun with functions. (**8 points**)
Our calculator is nice, but it only supports simple arithmetic. We will now let the user add functions. If we send a calculator process a message of the form
`F when is_function(F,2)`
The calculator should call `F(ProcState, Tally)` to get the tuple `{NewProcState, NewTally}`. The calculator continues with the new values for `ProcState` and `Tally`. For example, here's an example where I use the calculator to find the roots of $x^2 + 17x - 38 = 0$:

```
19> f(Calc).  % forget the q1b calculator
ok
20> Calc = hw2:calculator(q1c).
<0.197.0>
21> hw2:calculate(Calc,
      [{set, 1}, {store, a}, {set, 17}, {store, b}, {set, -38}, {store, c}]).
ok
22> hw2:calculate(Calc,
      [{set,4}, {times,a}, {times,c}, {store,four_a_c}]).
ok
23> hw2:calculate(Calc,
      [ {set,b}, {times,b}, {minus,four_a_c},
        fun(ProcState, Tally) -> {ProcState, math:sqrt(Tally)} end,
        {store, d} ]).
ok
24> hw2:calculate(Calc,
      [{set,2}, {times,a}, {store,two_a}]).
ok
25> hw2:calculate(Calc,
      [{set,-1}, {times,b}, {plus,d}, {divide,two_a}, {store, r1}]).
ok
26> hw2:calculate(Calc,
      [{set,-1}, {times,b}, {minus,d}, {divide,two_a}, {store, r2}]).
ok
27> Calc !  {set, r1}.
{set,r1}
28> hw2:fetch(Calc).
2.0
29> Calc !  {set, r2}.
{set,r2}
30> hw2:fetch(Calc).
-19.0
```

Your task is to write the function`q1c(ProcState, Tally)` and add a few test cases to `hw2_tests.erl`.

2. Linear functions. (**30 points**)
Function $hw2 : f/2$ is defined as shown below:

```
f([A,B], X) -> A*X + B.
```

The function $hw2 : f([A, B], X)$ computes a *linear* function of `X` according to the coefficients `[A,B]`. For example:

```
f([5/9,-32*5/9], FahrenheitTemperature) -> CelsiusTemperature.
f([9/5,32], CelsiusTemperature) -> FahrenheitTemperature.
```

(a) (**8 points**)
Write a function, `g([A2, B2], [A1, B1]) -> A3, B3` such that for any choice of numbers for `A1`, `B1`, `A2` and `B2`,

```
f([A2,B2], f([A1,B1], X)) = f(g([A2,B2], [A1,B1]) X)
```

Add a few test cases for `g` to `hw2_tests.erl`. We allow `A1`, `B1`, `A2` and/or `B2` to be floating point numbers. When testing your solution, we will ignore round-off error; i.e., we will test using something similar to the `?assertClose` macro from `hw1_tests.erl`.

Hint: If you are stuck, try writing a formula for `f([A2,B2], f([A1,B1], X))`. From that, figure out what `A3` and `B3` must be.

(b) Is `g` commutative? (**8 points**)

- If your answer is "yes", you need to prove it. It is sufficient to write the formulas for `g([A2,B2], [A1,B1])` and `g([A1,B1], [A2,B2])` and use algebra to show that they are equivalent.
- If your answer is "no", you need to give a counterexample to justify your answer.

(c) Is `g` associative? (**8 points**)

- If your answer is "yes", you need to prove it. It is sufficient to write the formulas for `g([A3,B3], g([A2,B2], [A1,B1]))` and `g(g([A3,B3],[A2,B2]), [A1,B1])` and use algebra to show that they are equivalent.
- If your answer is "no", you need to give a counterexample to justify your answer.

(d) Our Linear calculator (**6 points**)
Consider the calculator from Question 1a. Show that the calculator operations for each of the messages `{plus,X}`, `{minus,X}`, `{times,X}`, `{divide,X}`, and `{set,X}` had an equivalent representation as a linear operator of the form `[A,B]`. For example, `{plus, 5}` directs the calculator to add `5` to the current `Tally`. The corresponding linear operation is `f([1,5], Tally)`.

Complete the implementation of the function `h/1` in `hw2.erl` that translates calculator commands to linear operators. Add at least one test case for each calculator operation to `hw2_tests.erl`.

3. A parallel calculator. (**25 points**)
The function `hw2:calc_seq(Ops, Tally0)` implements a *sequential* version of the calculator – no processes are spawned by executing `calc_seq`. `Ops` is a list of operations to perform; each operation is a tuple of the form `{Op, Y}` where `Op` is one of `plus`, `minus`, `times`, `divide`, or `set`, and `Y` is a number. `Tally0` is the initial value for the accumulator. For this problem, you need to implement and test a parallel version of the calculator:

```
calc_reduce(Wtree, Key, Tally0)
```

where `Wtree` is a worker tree; `Key` is the key for a distributed list of operations, and `Tally0` is the initial value for the accumulator. The desired result is

```
calc_reduce(Wtree, Key, Tally0) == calc_seq(Ops, Tally0).
```

where `Ops = lists:append(workers:retrieve(Wtree, Key))`, and the usual remarks about ignoring floating point overflow, and we'll use `close` or something similar to check "equality".

- Implement `calc_reduce` (**12 points**).
Notes: the functions `f`, `g`, and `h` from Question 2 are helpful for this problem. You can probably get a performance advantage (and maybe even get superlinear speed-up) if you make your own versions optimized just for this problem, but that is not required. **Warning:** `calc_seq` works through `OpList` in left-to-right order, but `g(AB2, AB1)` constructs an operator that corresponds to applying AB1 first and AB2 second – i.e. it works in right-to-left order. This isn't a show-stopper, but you need to think about this when you implement your combine function.

- (**3** points). Add some test cases for calc_reduce to hw1_tests.erl.
- Speed-up vs. number of processes (**5 points**).
  Measure the time when the total number of operations is 1,000,000 using calc_seq and using calc_reduce with 1, 4, 16, 32, 64, and 128 processes running on thetis.ugrad.cs.ubc.ca. Report your timing measurements and speed-ups (relative to the calc_seq) in hw2.pdf.

  Note: it is important that you use thetis. It has 32 physical cores, that Erlang sees as 64 cores due to multi-threading. thetis is the most parallel shared memory machine that we have in the ugrad.cs.ubc.ca environment.
- Speed-up vs. problem size (**5 points**)
  Choose the number of processes that resulted in the greatest speed up in the previous question. Plot the speed-up versus the total number of operations for when the total number of operations ranges from 10,000 to 1,000,000.

4. A scanning calculator. (**20 points**)
For every reduce problem, there is a corresponding scan problem, and vice-versa. For this question, let Tally0 be the initial tally for the calculator. Given a list, OpList that is distributed across the workers of Wtree, in the same way as OpList. Here's the declaration for the function you need to write:

```
calc_scan(Wtree, SrcKey, DstKey, Tally0) -> your_answer([Wtree, SrcKey, DstKey, Tally0], calc_scan).
```

To be more specific, hw2.erl defines a function,

```
calc_scan_seq(OpList, Tally0) -> ResultList
```

that is the sequential version of calc_scan. The desired result is that if we first do

```
31> hw2:calc_scan(Wtree, SrcKey, DstKey, Tally0).
32> OpList = lists:append(workers:retrieve(Wtree, SrcKey)).
33> Ans_par = lists:append(workers:retrieve(Wtree, DstKey)).
34> Ans_seq = calc_scan_seq(OpList, Tally0).
```

then Ans_par == Ans_seq with the usual remarks about ignoring floating point overflow using close or something similar to check "equality".

- (**5 points**) Create a table like the one below in your hw2.pdf. Complete the right column with the element of Ans_par or Ans_seq that corresponds to the element of OpList in the left column:

OpList	Ans
{plus, 3}	
{times, 14}	
{minus, 10}	
{divide, 4}	
{minus, 5}	
{times, 7}	

  Give a one or two sentence informal description of what calc_scan computes.

  Hints: you should be able to use the function calc_scan_seq once you have completed the function h in hw2.erl – see Question 2d. The list for OpList in the table above is the list returned by hw2:calc_ops().
- (**15 points**) Implement the function calc_scan in hw2.erl and add a few test cases to hw2_tests.erl. It's easiest to *see* the results of the scan for small examples. We will test your code with small examples and large ones too.

Why?

Question 1: **Calculate with `ProcState`**

This question is intended to show how a process in Erlang can maintain "state" by passing information from one recursive call of its function to the next. This is a simplified version of the `ProcState` used in the `workers` and `wtree` libraries. It also gives you a bit more experience with processes and messages, and an example of sending a function to a process to give it a task to do.

This question also introduces the calculator, which provides an example for the reduce and scan questions.

Question 2: **Linear functions**

This question sets out the properties of the calculator operations that you need for the reduce and scan questions.

Question 3: **A parallel calculator**

This combines the ideas from the first two questions to get an interesting parallel program. Why is it interesting?

- First, it says that if you had a million operations to do with a calculator, you could give a thousand operations to each of a thousand of your friends, and they could compute "summaries" that can be combined with a reduce to get the complete answer. What is "surprising" about this is that your friends can start in the middle of the calculation without know the result from earlier operations. Think about that. If you weren't surprised at first, maybe you should be.

  While the result isn't initially obvious, if you're comfortable with linear algebra, you can see that each calculator operation can be represented by a matrix A, where A depends on the operation and operand. We then get

$$\begin{bmatrix} \texttt{NewTally} \\ 1 \end{bmatrix} = A \begin{bmatrix} \texttt{OldTally} \\ 1 \end{bmatrix}$$

  If linear algebra isn't your thing – **don't panic**. That's why I walked you through how to compose the linear operators of the calculator one step at a time.

  The other fun thing about this problem is that you can get pretty good speed-up. Yes, parallel algorithms do run faster than their sequential counterparts.

Question 4: **A scanning calculator**

Reduce and scan are closely related. Having done a reduce problem, I tossed in the scan version so you can see this connection in your own code.