CpSc 418

Homework 1

Due: Jan. 17, 2018, 11:59pm
Early Bird: Jan. 15, 2018, 11:59pm

56 points.

Please submit your solution using the handin program. Submit your solution as
cs418 hw1
Your submission should consist of three files:

- `hw1.erl`: Erlang source code for the coding parts your solution.
- `hw1.pdf` PDF for the written response parts of your solution and the plots.
- `hw1_tests.erl`: EUnit tests for your solution.

Templates for `hw1.erl` and `hw1_tests.erl` are available at
`http://www.ugrad.cs.ubc.ca/~cs418/2017-2/hw/1/code.html`.
The tests in `hw1_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. `f_to_c` (15 points)
   I was visiting my dad in Bellingham and he said that it was 45 degrees outside – "broiling hot!", I thought. But it was not.

   It turns out that they use an arcane temperature system called "Fahrenheit". We can convert from Fahrenheit temperatures to the more sensible Celsius measure with the formula:

   $$\texttt{CelsiusTemp} = (5/9) * (\texttt{FahrenheitTemp} - 32)$$

   (a) Convert one temperature (3 points) Write a function, `f_to_c1(F)` that takes an argument `F`, a temperature in Fahrenheit as an Erlang number (i.e. integer or float) and returns the corresponding Celsius temperature. For example,
   `f_to_c1(68.0) -> 20.0.`

   Note: this part of the problem asks for you to call the function `f_to_c1` – make sure you remember the '`1`' at the end. That's because I'll ask you to write a more general version in question 1b. Your solution needs to include `f_to_c1` for this question **and** `f_to_c` for question 1b.

   (b) Convert lists of temperatures (5 points) Write a function, `f_to_c(F)` that is a generalization of `f_to_c1` in the following way:

   - If `F` is a number, it should be interpreted as temperature in Fahrenheit and `f_to_c(F)` should return the corresponding temperature in Celsius. Example, `f_to_c1(68.0) -> 20.0.`
   - If `F` is a list of numbers, `f_to_c(F)` should return a list, `C`, where each element of `C` is the Celsius equivalent of the corresponding element of `F`. Example,

```
f_to_c([-50, -40, -30, -20, -10, 0, 10, 20, 30, 40, 50]) ->
```
$[-45.\overline{5}, -40.0, -34.\overline{4}, -28.\overline{8}, -23.\overline{3}, -17.\overline{7}, -12.\overline{2}, -6.\overline{6}, -1.\overline{1}, 4.\overline{4}, 10.0]$

- If `F` is a nested-list of numbers, your function can either fail, or it can return a corresponding nested lists of converted temperatures. For example,

  ```
  f_to_c([-50, [-40, -30, [], -20, [-10, 0], 10], 20, [30], 40, 50]).
  ```

  could fail (i.e. throw an error) or it can return

  $[-45.\overline{5}, [-40.0, -34.\overline{4}, [], -28.\overline{8}, [-23.\overline{3}, -17.\overline{7}], -12.\overline{2}], -6.\overline{6}, [-1.\overline{1}], 4.\overline{4}, 10.0]$

  Question 1c asks you which one you chose, and why.

(c) Explain your design (2 points) If `F` is a nested-list of numbers, for example
`F = [-50, [-40, -30, [], -20, [-10, 0], 10], 20, [30], 40, 50]`,
does your implementation return the corresponding nested-lists of converted temperatures, or does it throw an error? Give a short (one or two sentence) explanation of why you chose the implementation that you did.

(d) Efficiency (3 points) Your implementation of `f_to_c` should run in linear time. Use time_it:t/1 from the course library to measure the run time for your implementation with lists of length 1,000, 10,000, 100,000, and 1,000,000. For consistent results, you should run your tests on `bowen.ugrad.cs.ubc.ca` – of course, you are welcome to write and debug your code on any machine you like. time_it:t/1 reports the mean and standard-deviation for multiple runs, with a total runtime of about one second. For lists with 1,000,000 or fewer elements, your implementation of `f_to_c` should run in under one second.

You can use the function misc:rlist to generate random input lists. You should generate each random list *before* calling time_it:t/1 – for example,

```
N = SomeBigNumber,
RandomList = [X - 50 || X <- misc:rlist(N, 100.0)],
TimingData = time_it:t(fun() -> f_to_c(RandomList) end),
{mean, Mean} = lists:keyfind(mean, 1, TimingData),
{std, StandardDeviation} = lists:keyfind(std, 1, TimingData),
io:format("N=~7b, mean time = ~10.3e (seconds), standard deviation = ~10.3e (seconds)~n",
          [N, Mean, StandardDeviation]).
```

(e) Tests (2 points) Add some test cases for `f_to_c` to your copy of `hw1_tests.erl`. Make sure that you cover the corner cases such as `f_to_c(F)` where `F` is the empty list, of where `F` is neither a number nor a list, etc.

Full credit will be given for any reasonable set of tests. We will also implement a more relaxed collaboration policy *just for* `hw1_tests.erl`. If you collaborate with others, you can jointly write one `hw1_tests.erl` and all share it. **Of course,** everyone in your collaboration group must submit a copy of the file, and you each must clearly state who your collaborators were in a comment within the first 5 lines of your `hw1_tests.erl` file.

2. `linmap` (7 points)
The function `f_to_c` is a special case of a *linear mapping*: each element, X, of the input list is replaced with $A * X + B$ in the result list.

(a) Implement `linmap(A, B, X)` (3 points)

- If `X` is an Erlang number, then `linmap` should return `A*X + B`.
- If `X` is a list of Erlang numbers, then `linmap` should return a list where each element of the return list is linear mapping for the corresponding element of the input list. For example,

  ```
  linmap(2.0, -3.0, [0, 1, 2, 3, 42]) -> [-3.0, -1.0, 1.0, 3.0, 81.0].
  ```

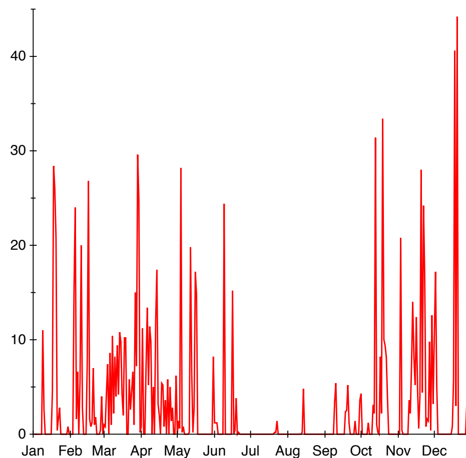

Figure 1: Daily precipitation in Vancouver in 2017

- If `X` is a nested list of Erlang numbers, then `linmap` can fail (i.e. throw an error) or return the corresponding nested list of the linear mappings for each number in the input – just like in question 1b.

(b) Implement `c_to_f(CelsiusTemperature)` (2 points)
That converts Celsius temperatures to their Fahrenheit equivalents, and handles lists as for `f_to_c`. Your implementation *must* use `linmap`.

(c) Test cases and efficiency (2 points). Add test cases for `linmap` and `c_to_f` to `hw1_tests.erl`. Your implementation of `linmap` (and therefore `c_to_f`) should run in linear time – we will check the execution time.

3. How wet is Vancouver? (14 points)
The file
`http://www.ugrad.cs.ubc.ca/~cs418/2017-2/hw/1/src/rain.erl`
gives daily precipitation (in millimeters) for Vancouver in 2017. The function `rain:plot(Filename)` creates a SVG file that can be displayed to produce a plot like the one shown in Figure 1. `Filename` should be a string and you probably want it to end with `.svg`. On the CS department linux machines, the command
`display` *file.svg*
will create a window and display image described in the svg file.

The daily precipitation data produces a rather jagged plot. This question and the next one examine how we can smooth the data to make it easier to for a human understand the plot.

For this problem, you will write a function `simple_smooth` that computes a weighted average of the value for each element of a list with the the element just before and the element just after. In more detail, let
`SmoothData = simple_smooth(RawData).`
For $1 < I <$ `length(RawData)`, `SmoothData` should satisfy

$$\texttt{lists:nth(I, SmoothData)} = \frac{1}{4}*\texttt{lists:nth(I-1, RawData)} + \frac{1}{2}*\texttt{lists:nth(I, RawData)} + \frac{1}{4}*\texttt{lists:nth(I+1, RawData)}.$$

This is a *mathematical definition* of what the return result should satisfy. **Do not** implement `simple_smooth` with all those calls to lists:nth or you'll end up with a quadratic time implementation and lose a bunch of points.

We need to take care of the end cases as well. If `RawData` is an empty list or a list of a single element, `simple_smooth` should just return `RawData`. For lists of two or more elements, the first and last values of `SmoothData` should satisfy

$$\texttt{lists:nth(1, SmoothData)} = \frac{3}{4}*\texttt{lists:nth(1, RawData)} + \frac{1}{4}*\texttt{lists:nth(2, RawData);}$$

$$\texttt{lists:nth(}N\texttt{, SmoothData)} = \frac{1}{4}*\texttt{lists:nth(}N\texttt{-1, RawData)} + \frac{3}{4}*\texttt{lists:last(RawData).}$$

where N = `length(RawData)`. This means that we compute the first element of `SmoothData` as if the (non-existent) 0th element of `RawData` were the same as the first element of `RawData`. Likewise, we compute the last element of `SmoothData` as if the (non-existent) $N+1^{\text{st}}$ element of `RawData` were the same as the Nth element.

Here are a few examples:

```
simple_smooth([]) -> [];
simple_smooth([123.4]) -> [123.4];
simple_smooth([1,1,2,3,5,8]) -> [1.0,1.25,2.0,3.25,5.25,7.25]
```

(a) (8 points) Implement `simple_smooth(RawData)`.

(b) (2 points) Plot the smoothed precipitation graph and include the plot in the `hw1.pdf` file that you submit. To get the data for the precipitation graph in figure 1, use the function `rain:daily()`. This function returns a list of `{X,Y}` pairs. The `X` value is the day of the year, $1 \leq$ `X` ≤ 365. The `Y` value is the total precipitation (in millimeters) on that day. Use your `simple_smooth` function to create a smoothed version of the data.

To plot the data, you can use the function `rain:plot2(FileName, Data)`. This function produces an SVG file with the name given by `FileName`. `Data` should be a list of `X,Y` tuples where `X` is the day of the year, and `Y` is the smoothed precipitation for that day. The plot generated by `rain:plot2` shows the original data in red and the smoothed data in blue.

(c) (4 points) Test cases and efficiency. Add test cases for `simple_smooth` `hw1_tests.erl`. Your test cases should include corner cases. Your function should fail (i.e. throw an error) if `RawData` is not a list of numbers. You can explicitly throw the error, or it can be thrown by the Erlang runtime because your code attempts an illegal operation – either is OK. It is **not** acceptable to return some value if `RawData` is not a list of numbers.

Your implementation of `simple_smooth` should run in linear time – we will check the execution time for lists much larger than one-year of rain data. Your implementation should run in less than one second for an input list with 100,000 elements. Use time_it:t/1 to check the execution time.

4. Convolving in the rain (20 points)
The smoothing function in question 3 was nice, but we can do better. A generalization of this approach is called *convolution*. The idea is that for the Ith value of `Data`, we will compute a weighted average of the Ith value itself and the M values before and after the Ith value. The set of weights for the average is called the *convolution kernel*. The formula for the convolution is:

$$\texttt{SmoothData}[I] = \sum_{J=-M}^{M} \texttt{Kernel}[J] * \texttt{Data}[I-J]$$

As in question 3, we will use `Data[1]` when $I - J < 0$, and $Data[N]$ when $I + J > N$, where N is the number of elements in `Data`.

Of course, we are going to do this in Erlang. You will write a function,

```
conv(Kernel, Data) -> Smooth.
```

Where `Kernel` and `Data` are lists of numbers. The length of `Kernel` must be an odd number – in other words $2M + 1 =$ `lengthKernel`. `Smooth` is a list of numbers with the same length as `Data`. `Smooth` is the list where for all $1 \leq \texttt{I} \leq N$,

$$\texttt{lists:nth(I, Smooth)} = \sum_{\texttt{J}=-M}^{M} \texttt{lists:nth(M+1+J, Kernel) * fetch(I-J, Data).}$$

```
fetch(I, Data) when is_integer(I), 1 =< I, I =< length(Data) ->
  lists:nth(I, Data);
fetch(I, Data) when is_integer(I), I < 1 -> hd(Data);
fetch(I, Data) when is_integer(I), I > length(Data) -> lists:last(Data).
```

As in question 3, this is a *mathematical definition* of what the return result of `conv` should satisfy. Don't write an implementation with all of these calls to lists:nth – it will be way too slow and you will lose points on the performance tests.

(a) (12 points) Implement `conv(Kernel, Data)`.

(b) (2 points) Plot the smoothed precipitation graph and include the plot in the `hw1.pdf` file that you submit as your solution. To get the data for the precipitation graph in figure 1, use the function `rain:daily()` as described in question 3. For the `Kernel`, use `gauss_kernel(3)` from the `hw1.erl` template file. This is a Gaussian smoothing kernel with a standard deviation of about ± 3 – it means the smoothing kernel is roughly a weekly average.

To plot the data, you can use the function `rain:plot2(FileName, Data)` as described in question 3.

(c) (4 points) Run-time analysis
What is the $\mathcal{O}$ run-time for your implementation of `conv`. Your answer should be in terms of M, the kernel size, and N, the number of elements in `Data`. Give a short (one to three sentence) justification for your answer.

(d) (2 points) Run-time measurement
Make a few measurements (10 to 20 should be enough but more are fine) varying the size of `Kernel` and the number of element in `Data`. Do the measurements match your analysis?

(e) (2 points) Test cases.
Be sure to add test cases to `hw1_tests.erl`. We will test your code, you should to.

Why?

This is a follow-up to PIKA 1 to make sure that everyone is confident programming in Erlang. Often, people who are new to functional programming find that they miss their loops. The questions in this assignment require various forms of traversing data structures, obviously without loops. They are arranged roughly in an order from simple programming patterns to some more general ones. These questions also introduce examples such as convolution that we will revisit in the term.

Question 1: This is one of the simplest patterns: a map. Each element of the input list is mapped to a new value for the result list. The function is overloaded to work on numbers or lists. This should give you some experience with pattern matching.

Question 2: Linear maps are a common computation – the `f_to_c` function was one example of such a map. With `linmap`, we replaced the hard-coded constants of `5/9` and `32` that were used in `f_to_c` with parameters.

The `linmap` function will come up again when we study GPUs and CUDA. The people who write CUDA books like the BLAS function `saxpy` – that's just `linmap` wrapped up in a bunch of contorted acronyms. "BLAS" stands for "basic linear algebra subprograms" and `saxpy` stands for "`s`ingle-precision codea*x `plus` `y`". "Linear map" seems easier to say and remember. By giving an example here, hopefully the arcane names of BLAS functions will be a bit less intimidating.

Question 3: This generalizes the map operations of the first two problems to a computation that involves adjacent elements of a list. A common hurdle for people who use languages with loops, is "How do I keep track of what comes before and after the current element when writing my recursive function?".

Imperative languages such a C and Java allow any iteration of the loop to access any element of an array. On the one hand, this can seem convenient. On the other hand, it's a common cause of errors – programmers often don't think about what information they are using from previous iterations to perform the current iteration, and that leads to errors in the first few iterations or the last few.

With functional programming, you *must* identify what values you will use from previous iterations. You need to pass them along as parameters to your recursive function. On the one hand, this can seem like annoying "clutter" in the code. On the other hand, making the dependencies explicit can help avoid errors.

This class is about parallel computing. So, the imperative vs. functional styles isn't the central point – except it is. For parallel computing, we need to divide the computation between processes that execute in parallel. Communication is expensive. Any idea that any process can access any value will collide with the reality of real parallel computers. The functional style has us thinking about what values are needed where from the very beginning. It's not the only way to write parallel code, but it's a good place to start.

The `simple_smooth` function traverses the `Data` list *and* it has special cases at the beginning and end. I found it helpful to write a `simple_smooth` as a wrapper function that handles the special cases for short-lists and then calls a helper for the main recursive computation. This is a common pattern when writing functional code.

Question 4: This question generalizes the smoothing problem from question 3. In this case, you have a nested recursion: `conv(Kernel, Data)` needs to traverse `Data`, and for each value of `Data` it needs to traverse the list `Kernel`. As in `q:wet`, there are special cases at the beginning and end of `Data`, but for `conv`, the special cases depend on the length of `Kernel`.

As with question 2, this question was also inspired by what we will do with CUDA. The previous two offerings of this course have included implementing convolution in CUDA. We might have that again this term. I'm introducing the idea here.

The Library, Errors, Guards, and other good stuff

The CPSC 418 Erlang Library: your code *must* run on the CS department linux machines. Some of the functions used in this assignment such as time_it:t/1 and plotting functions called by `rain:plot` and `rain:plot2` are from the course library. To access this library from the CS department machines, give the following command in the Erlang shell:

```
1> code:add_path("/home/c/cs418/resources/erl").
```

You can also set the path from the command line when you start Erlang. I've included the following in my `.bashrc` so that I don't have to set the code path manually each time I start Erlang:

```
function erl { /usr/local/bin/erl erl -eval 'code:add_path("/Users/mrg/classes/cs418/2017-1/src/erl")'
```

See http://erlang.org/doc/man/erl.html for a more detailed description of the `erl` command and the options it takes.

If you are running Erlang on your own computer, you can get a copy of the course library from
http://www.ugrad.cs.ubc.ca/~cs418/resources/erl/erl.tgz
Unpack it in a directory of your choice, and use code:add_path as described above to use it. Changes may be made to the library to add features or fix bugs as the term progresses. I try to minimize the disruption and will announce any such changes. I *do* plan to modify the `plot` and `svg` modules – they are both very new, and I'll want to add some more features for subsequent homework assignments. I'd like to add edoc comments to the source so that the `plot` and `svg` modules will be included in the on-line documentation for the course library. Currently, they lack documentation – you can use them by calling `rain.plot` or `rain.plot2` as described in questions 3 and 4. Or, you can be brave and look at the code and figure out how to use it. If you're brave, be warned that I reserve the right to change anything that isn't documented! In that case, diff is your friend.

Compiler Errors: if your code doesn't compile, it is likely that you will get a zero on the assignment. Please do not submit code that does not compile successfully. After grading all assignments that compile successfully, we *might* look at some of the ones that don't. This is entirely up to the discretion of the instructors and TAs. If you have half-written code that doesn't compile, please comment it out or delete it.

Compiler Warnings: your code should compile without warnings. In my experience, most of the Erlang compiler warnings point to real problems. For example, if the compiler complains about an unused variable, that often means I made a typo later in the function and referred to the wrong variable, and ended up not using the one I wanted. Of course, the "base case" in recursive function often has unused parameters – use a `_` to mark these as unused. Other warnings such as functions that are defined but not used, the wrong number of arguments to an io:format call, etc., generally point to real mistakes in the code. We will take off points for compiler warnings.

Printing to stdout: please don't unless we specifically ask you to. If you include a *short* error message when throwing an error, that's fine, but not required. If you print *anything* for a case with normal execution when no printing was specified, we will take off points.

Guards: in general, guards are a good idea. If you use guards, then your code will tend to fail close to the actual error, and that makes debugging easier. Guards also make your intentions and assumptions part of the code. Documenting your assumptions in this way makes it much easier if someone else needs to work with your code, or if you need to work with your code a few months or a few years after you originally wrote it. There are some cases where adding guards would cause the code to run much slower. In those cases, it can be reasonable to use comments instead of guards. Here are a few rules for adding guards:

- If you need the guard to write easy-to-read patterns, use the guard.
- If adding the guard makes your code easier to read (and doesn't have a significant run-time penalty), use the guard.
- If a function is an "entry point" into your code (e.g. and exported function) it's good to have your assumptions about arguments clearly stated. If you can do this with guards, that is great.
- Adding lots of little guards to every helper function can clutter your code. Write the code that you would want others to write if you are going to read it.
- In some cases (discussed below), guards can cause a severe performance penalty. In that case, it's better to use a wrapper function so you can test the guards once and then go on from there, or to use comments. Comment don't slow down the code.

The rest of this discussion of guards is an example showing how a poorly considered guard can change an $O(N)$ time algorithm to $O(N^2)$. It does this by calling `length` in a guard – `length(List)` take time that is linear in the length of `List`. My example for guards that make for terribly slow code is a function `allLess(L1, L2)` that is a check that all elements of `L1` are less than the corresponding elements of `L2`.

```
allLess([], []) -> true; allLess([H1 | T1], [H2 | T2])
    when is_number(H1), is_number(H2),
         is_list(T1), is_list(T2), length(T1) == length(T2) ->
  (H1 < H2) andalso allLess(T1, T2).
```

I tried `allLess(lists:seq(0,999), lists:seq(1,1000))`. Using `time_it:t`, it takes about 1.4ms (on my laptop) to execute the call to `allLess`. If I change the guard to:

```
when is_number(H1), is_number(H2)
```

Then it takes about 14μs – it's 100 times faster. That's because the function `length` traverses the list and takes time that is linear with the list length. Because the guard is invoked for each recursive call to `allLess`, the guard changes an $O(N)$ computation to $O(N^2)$. In cases like this, it's fine to use the simpler guard. If you call `allLess` with lists of different lengths, this means you'll get an error message showing a call that is different than the one you originally made – it's the call that happened after a bunch of recursive calls, and that can make debugging a little harder – or a lot harder, depending on the details.

A common case for omitting guards occurs with tail-recursive functions. We often write a wrapper function that initializes the "accumulator" and then calls the tail-recursive code. We export the wrapper, but the tail-recursive part is not exported because the user doesn't need to know the details of the tail-recursive implementation. In this case, it makes sense to declare the guards for the wrapper function. If those guarantee the guards for the tail-recursive code, and the tail recursive code can only be called from inside its module, then we can omit the guards for the tail-recursive version. This way, the guards get checked once, but hold for all of the recursive calls. Doing this gives us the robustness of guard checking **and** the speed of tail recursion.

Quick review question: is `allLess` tail recursive?

A remark for those who are really into analyzing the code. The behaviour of

```
allLess([1], [0, 0])
```

changes with the guard. The call to `allLess` fails with

```
** exception error: no function clause matching
   hw1:allLess([1],[0,0]) (hw1.erl, line 89)
```

when I use the version with the guards for `T1` and `T2`. When those guards are omitted, `allLess` returns false.

Unless otherwise noted or cited, the questions and other material in this homework problem set is copyright 2018 by Mark Greenstreet and are made available under the terms of the Creative Commons Attribution 4.0 International license http://creativecommons.org/licenses/by/4.0/