CpSc 418 **Homework 1** Due: Sept. 26, 2013, 11:59pm

80 points + 6 extra credit.
5% extra credit for problems 1, 2, and 4 if solution submitted by 11:59pm on Sept. 23.

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

`hw1.erl` – Erlang source (ASCII text).

`hw1.txt` or `hw1.pdf` – plain, ASCII text or PDF.

The first file, `hw1.erl`, will be your solution to the programming part of the assignment.
Unless otherwise stated, you may use any functions that you like from the Erlang library, for example, functions from the `lists` module.
The second file, `hw1.txt` or `hw1.pdf` should give your solutions to the written parts of the assignment.
No other file formats will be accepted.

1. **Tail Recursion Optimization(25 points).** Consider the following two functions to compute the sum of the elements of a list:

```
% sum_hr(List) -> number – sum of the elements of List, head-recursive version
sum_hr([]) -> 0;
sum_hr([Hd | Tl]) -> Hd + sum_hr(Tl).

% sum_tr(List) -> number – sum of the elements of List, tail-recursive version
sum_tr(List) -> sum_tr(0, List).
sum_tr(S, []) -> S;
sum_tr(S, [Hd | Tl]) -> sum_tr(S+Hd, Tl).
```

The first version, `sum_hr` is the one that most people find easier to read. This form is called "head recursion" because the recursive call occurs at the beginning of the function – note that `sum_hr` needs make the recursive call *before* it can perform the addition to produce the final result.

The second version, `sum_tr` is called "tail recursion" because the recursive call occurs at the very end of evaluating the function. Thus, the value returned by `sum_tr` is the value returned by the recursively called function. This allows the compiler to perform a very useful optimization:

> **Instead of creating a new stack frame for the recursive call, the compiler generates code to re-use the existing stack frame, but sets it up for the next iteration of the function.**

Effectively, the compiler converts tail-recursive functions into while loops. This is why programmers often use the tail-recursive form for a function, even though the head-recursive version may be easier to read.

With this optimization, I would expect `sum_tr` to be *faster* than `sum_hr`. So, I tried it. We can measure how long it takes a piece of code to execute by using the `time_it:t(Fun, HowLong)` function from the CpSc 418 Erlang library. See notes about getting this library at the end of this assignment. For example,

```
1> L = lists:seq(1,100), ok.   % The list [1, 2, ..., 100]
ok % But I suppressed the long Erlang spew by appending , ok
2> time_it:t(fun() -> hw1:sum_hr(L) end, 1.0)
[{mean,3.450424401342413e-6},{std,9.348702447983752e-7}]
3>
```

`time_it:t(fun() -> hw1:sum_hr(L) end, 1.0)` calls `hw1:sum_hr` repeatedly with the argument `L` until the total time reaches 1.0 seconds. It then reports the average time per call and standard deviation. For the example above, running Erlang on my laptop, the average time per call of 3.45μs, with a standard deviation of 0.935μs. Then, I tried the tail-recursive version.

```
3> time_it:t(fun() -> hw1:sum_tr(L) end, 1.0)
[{mean,3.5942721793006116e-6},{std,1.1329852144921657e-6}] 4>
```

The average time per call is 3.59μs – the tail recursive version is 4% *slower* than the head-recursive version!

OK, the compiler is doing some serious optimizations. So, I had to revise the problem to show why tail recursion matters (and when it doesn't).

(a) (**10 points**): write a function

$$\texttt{add\_hr(N) ->} \sum_{I=1}^{\texttt{N}} I$$

using head-recursion. Actually add up all the numbers. **Don't** just solve the recurrence and write the simple formula, $\frac{1}{2}(N^2 + N)$. Then, write `time_add_hr(N)` that uses `time_it:t` to determine the average time per call of `add_hr(N)` with a total time of 1.0 seconds.

Now write the tail recursive version

$$\texttt{add\_tr(N) ->} \sum_{I=1}^{\texttt{N}} I$$

and `time_add_tr(N)` that reports the average time per call of `add_tr(N)`.

In `hw1.txt` or `hw1.pdf` briefly summarize your observations about which one is faster and by how much.

(b) (**10 points**): Even though `sum_tr` was not any faster than `sum_hr`, it does save on stack space – it just took me a bit of time to figure out how to measure this. I wrote two functions and have included them in the template for `hw1.erl`:

```
stack_size() -> current size of the stack for this process

sum_help(N, {Tally, MaxStack}) -> {NewTally, NewMaxStack}
% NewTally = Tally + N
% NewMaxStack = max(MaxStack, current stack size)
```

Thus, `sum_help` does the addition for the sum, and keeps track of the largest stack-size encountered so far. To get an initial value for `{Tally, MaxStack}`, my code uses `{0, stack_size()}`. Stack size is reported in Erlang words.

Write a function `sum_hr_smem(List) -> {Total, MaxStack}` where `Total` is the sum of elements of `List` (computed with a head-recursive implementation), and `MaxStack` is the maximum stack size that occurred while doing the computation.

Likewise, write a function `sum_tr_smem(List) -> {Total, MaxStack}` that returns the sum of the elements of the list and the maximum stack size for a tail-recursive implementation.

Try both functions for various sized lists. Report your data in `hw1.txt` or `hw1.pdf`. For each of `sum_hr(L)` and `sum_tr(L)`, give a simple formula for the stack size as a function of the length of `L`.

(c) (**5 points**): In Erlang, it is common to write a function for a "reactive process" with the form:

```
my_process(Args) ->
    receive % wait until we get a message requesting us to do something
        A_Message ->
            handle the message,
            send responses,
            compute NewArgs
    end, my_process(NewArgs).   % continue with the next message
```

Why is it very important to write this as a tail-recursive function? In other words, give a **brief** explanation of what can go wrong if `my_process` is *not* tail recursive. Include your answer in `hw1.txt` or `hw1.pdf`.

2. List Flattening (0 points).

   Problem cancelled: I accidentally included the solution in hw1.erl.

3. **Counting Corners (30 points).** Let `L` be a list. We will say that the N^{th} element is "ascending" if it is less than its successor, descending if it is greater than its successor, and "flat" if it is equal to its successor. Note that the last element is neither ascending, descending, nor flat. So, we can map a list of `N` elements to a list of `N-1` atoms of values `a` (for ascending), `d` (for descending), or `f` (for flat). For example,

```
Let L   =  [1, 2, 3, 0, 18, 16, 25, 32, 42, 42, 42,-5]
    ADF =  [a, a, d, a, d,  a,  a,  a,  f,  f,  d]
```

   `ADF` gives the ascending, descending, flat characterization for the corresponding elements of `L`.

   An element is a "corner" if:

   - it is an ascending element, followed by zero or more flat elements followed by a descending element, or
   - it is a descending element, followed by zero or more flat elements followed by an ascending element.

   For example, `L` has five corners as shown below.

```
[1, 2, 3, 0, 18, 16, 25, 32, 42, 42, 42, -5]
    ↑  ↑  ↑   ↑              ↑
```

   (a) (**10 points**) Write a function, `count_corners(L) -> Ncorners` that returns the number of corners in list `L`. If `L` is the empty list, then `count_corners(L) -> 0`. Your implementation of `count_corners(L)` should run in linear time in the length of `L`.

   (b) (**10 points**) Write a function,
   `check_corners(L) -> {First, StartSlope, Ncorners, EndSlope, Last}`
   where

   `First` is the first element of `L`;
   `StartSlope` is
   `+1` if `L` starts with zero or more flat elements followed by an ascending element;
   `-1` if `L` starts with zero or more flat elements followed by an descending element;
   `0` if all elements of `L` are flat;
   `Ncorners` is `count_corners(L)`;
   `EndSlope` is
   `+1` if `L` ends with an ascending element followed by zero or more flat elements;
   `-1` if `L` ends with a descending element followed by zero or more flat elements;
   `0` if all elements of `L` are flat;
   `Last` is the last element of `L`.

   If `L` is the empty list, then `check_corners(L)` should return the atom `empty`.

   (c) (**10 points**) Write a function, `merge_corners(C1, C2) -> C12` where `C1`, `C2`, and C12 are tuples of the form `{First, StartSlope, Ncorners, EndSlope, Last}` or the atom `empty` as returned by `check_corners`. If `L1` and `L2` are lists, then
   `merge_corners(check_corners(L1), check_corners(L2))`
   should produce the same result as `check_corners(L1 ++ L2)`.

Note: Erlang has some special rules when mixing integer and floating point values. In particular, `3 == 3.0` produces the value `true`, but `3` does not *match* `3.0`. You can use the `=:=` operator to check for matching equivalence. For example:

```
4> 3 == 3.0.
true
5> 3 = 3.0.
** exception error:  no match of right hand side value 3.0
6> 3 =:= 3.0.
false
```

I've written my solution so that all comparisons are equivalent to using `<`, `==`, and `>`. I will not write test cases that mix integer and floating point numbers in a way that the difference between `==` and `=:=` matters. However, I'm pointing it out here because there are ways you could write code that would cause "strange" things to happen. For example:

```
f(X, Y) when X < Y -> ...; % less-than case
f(X, X) -> ...; % equals case
f(X, Y) -> ....  % greater-than case – NOT NECESSARILY
```

Note that `f(3, 3.0)` and `f(3.0, 3)` will *both* evaluate the case that is supposedly for `X > Y`. The "right" way to write this is:

```
f(X, Y) when X < Y -> ...; % less-than case
f(X, Y) when X == Y -> ...; % equals case
f(X, Y) -> ....  % greater-than case
```

Now `f(3, 3.0)` and `f(3.0, 3)` will both evaluate the "equals" case.

4. **Reduce (25 points).** For integers N > 0 and $M \geq 0$, let

```
collatz(N, 0) -> N;
collatz(N, M) when (N rem 2 == 0) -> collatz(N div 2, M+1);
collatz(N, M) when (N rem 2 == 1) -> collatz(3*N + 1, M+1).
```

The Collatz Conjecture states that for any N > 0 there exists an M ≥ 0 such that `collatz(N,M) = 1`. As of today, the conjecture remains open (and probably will stay that way for quite a while).

In the template for `hw1.erl`, I have provided functions:

```
collatz_len(N) when is_integer(N) -> the smallest M such that collatz(N, M) = 1.

collatz_max(Lo, Hi) -> {N, K}.  % Where Lo ≤ N ≤ Hi; collatz_len(N) = K;
    % and for every J with Lo ≤ J ≤ Hi, collatz_len(J) ≤ K. In the case of "ties",
    % the smallest value of N that achieves the longest collatz_len is returned for N.
```

along with some other versions that take lists as arguments or provide a default value for `Lo`. Note: if you call `collatz_len(N)` with a choice of `N` that *never* reaches one, the `collatz_len` function will tail-recurse forever. If you find such a value for `N`, please let me know – you will get lots of extra credit. ☺

Using these functions, we can find that the integer that is less than or equal to 10 with the longest Collatz-path to `1` is nine, with a length of 19. The integer less than or equal to a million with the longest Collatz-path to `1` is 837,799, with a path length of 524.

Use the functions in the `workers` and `wtree` modules from the course library (see `http://www.ugrad.cs.ubc.ca/~cs418/resources/erl/doc/index.html`), to implement:

```
collatz_max_par(Lo, Hi, W) -> {N, K}
```

where `W` is a worker-pool from `wtree:create`. Your solution should be an example of using the `wtree:reduce` function.

5. **(6 points, extra credit).** For each of problems 1, 3, and 4 on this assignment:
   (a) How long did it take you to solve the problem?
   (b) Please rate each problem on a scale of 0 to 5 where
      0 – Worthless tedium.
      1 – Too much work, and I learned little.
      2 – A typical homework problem.
      3 – Definitely had a favorable learning/effort ratio.
      4 – I learned a lot and had fun doing so.
      5 – Wow! I've discovered a new way to think!

   Feel free to add other comments as well.

The CpSc 418 Erlang Library

Getting the library: Go to
`http://www.ugrad.cs.ubc.ca/~cs418/resources/erl/source.html`
You can download the modules individually, or get a tar'd, gzipped archive at
`http://www.ugrad.cs.ubc.ca/~cs418/resources/erl/erl.tgz`

Getting the documentation: Go to
`http://www.ugrad.cs.ubc.ca/~cs418/resources/erl/doc/index.html`
This is generated by the edoc utility from comments in the source files.

Setting your Erlang path:
Use the function: `code:add_path(Directory)` where `Directory` is a string that gives the pathname to the directory with Erlang modules that you want on your path. The modules need to be compiled – the Erlang run-time looks for `.beam` files, not `.erl` files.

- I've found it convenient to write an Erlang module called `path` that exports one function with no arguments called `path` that sets my path. Then, I just give the command

  ```
  1> path:path().
  ```

  at the beginning of my Erlang session and I get the class library modules in my path.
- Or, you could set your `ERL_PATH` environment variable.
- Or, you can use the `-pa` or `-pz` options when you give the `erl` command. See `http://www.erlang.org/doc/man/erl.html`

Why?

Question 1: Introduce the concept of tail-recursion. It's important to understand that compilers for functional languages perform tail-call optimization. As seen in this problem, the impact on time is fairly small (for Erlang). However, the impact on memory usage can be much more important.

It used to be that tail-call optimization produced code that was **way** faster than head-recursive code. This is why you can come across code from experienced functional programmers who go way out of their way to make a function

tail-recursive. More advanced compilers have closed the speed gap, and as we saw here, in many cases pretty much eliminate it. Of course, the memory usage still matters. See
http://www.erlang.org/doc/efficiency_guide/myths.html
for a more complete discussion.

Note that tail-call elimination goes beyond converting a recursive call to a while loop. Anytime the final expression of a function is a call to another function, the tail-call optimization can replace the call with the operations or replacing the current stack frame with the context for the next function and then "jumping" to the entry point of the next function.

Question 2: **CANCELLED**
This was supposed to be a follow-up to question 1 to help you understand the performance implications of common operations in a functional language. The short version is:

- If you construct a list with `[Hd | Tl]`, the Erlang run-time can perform this operation in $O(1)$ time. Basically, it creates a new list cell, sets the "value" pointer to point to `Hd`, and sets the "next" pointer to point to `Tl`. Note that we can have as many *incoming* pointers to a list as we want. Erlang is functional. That list will never change. Adding another pointer to it doesn't change its value. So, many lists can share a suffix.
- Likewise, for a non-empty list, `L`, the Erlang run-time evaluates `hd(L)` and `tl(L)` in $O(1)$ time.
- If you construct a list with `L1 ++ L2`, the Erlang run-time can needs to make a copy list `L1`; let's call this list `L1b`. The last list cell of `L1` has its next pointer set to null. The last cell of the copy, `L1b`, gets its next pointer set to point to `L2`. Creating `L1b` takes time $O($`length(L1)`$)$, and the time to evaluate `L1 ++ L2` is $O($`length(L1)`$)$. Note that this can be smaller than $O($`length(L1)` $+$ `length(L2)`$)$ in the case that `length(L1)` $\ll$ `length(L2)`. Note that `L1b` is a "shallow" copy of `L1`. If some of the elements of `L1` are themselves lists or tuples, list `L1b` will have pointers to the same lists or tuples. Again, the run-time is taking advantage of the fact that Erlang is functional and it knows that these lists and tuples will never change.

If you miss question 2, you can get it at:
http://www.ugrad.cs.ubc.ca/~cs418/2013-1/hw/1/q1.pdf

Question 3: This is an opportunity to do something a little more involved using functions, lists, tuples, and pattern matching. I also plan to combine it with reduce in homework 2. I found writing `merge_corners` to be the one that needed the most thought. There is an "obvious" solution that enumerates the patterns for all of the combinations of `L_endslope`, `R_startslope`, and `sign(R_first - L_last)`, but that ends up with 27 cases plus two more for the cases when the left or right arguments are the atom `empty`.

Hint: I originally used atoms for the slopes: `up`, `down`, and `flat`. I found that using the integers $+1$, -1, and 0 made it possible to combine many of the cases together. My solution has two patterns for `empty` arguments, and one for everything else. My "everything else" case has four if statements.

Question 4: I wanted to make sure that there was something with processes, communication, and parallel computing on this assignment. I tried to make a problem that wasn't trivial, but that won't require large amounts of code.

Question 5: Let me know what you find educational and what you enjoy. I'll take that into account as I devise other homework problems.

I did extensive revisions of the material for this week's lectures to incorporate the mini-assignment approach. Unfortunately, I ended up needing to print this assignment before I had time to thoroughly proof-read it. My apologies for the typos and corrections.