CpSc 418

Homework 1

Getting started with Erlang

Due: January 24, 2023, 11:59pm
Early-Bird: January 22, 2023, 11:59pm

Prelude

Please submit your solution using:
`handin cs-418 hw1` Your solution should contain two files:

`hw1.erl`: Erlang source code for your solutions to the questions.

`hw1.pdf`: Your solutions to written questions.

Templates for `hw1.erl` and `hw1_tests.erl` are available at
`http://www.students.cs.ubc.ca/~cs-448/2022-2/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. Any question about whether or not your code compiled as submitted will be determined by trying it on CS department linux machines.

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 io: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.

This assignment is based on the first thirteen sections of *Learn You Some Erlang* – up through More on Multiprocessing.

The Questions (88 points)

1. Maps, Folds, and List Comprehensions (20 points)

   (a) `sum` (8 points)
   Consider the function

   ```
   sum(0) -> 0;
   sum(N) when is_integer(N), 0 < N ->
     N + sum(N-1).
   ```

   i. Heads or tails (2 points): Is `sum` head-recursive or tail-recursive?
   Revise the body of the function `sum_i` in the `hw1.erl` code template to return the atom `head_recursive` if `sum` is head-recursive, and `tail_recursive` otherwise.

   ii. (3 points) Write the other version – i.e. if `sum` is head-recursive, write `sum_tr(N)` that is tail recursive; likewise, if `sum` is tail-recursive, write `sum_hr(N)` that is head recursive. Of course, for any `N` (including invalid arguments), you should have `sum_tr(N) =:= sum(N)`, or `sum_hr(N) =:= sum(N)` (depending on which one you need to write). You may need to write a helper function in addition to writing `sum_hr` or `sum_tr`.

   iii. (3 points) Write another version, `sum_iii` using lists:foldl, lists:foldr, or lists:map. You can also use lists:seq.

(b) `sq` (8 points)
Consider the function

```
sq([]) -> [];
sq([Hd | Tl) -> [Hd * Hd | sq(Tl)].
```

   i. Heads or tails (2 points): Is `sq` head recursive or tail recursive?
   Anwer "head recursive" or "tail recursive" and give a one sentence justification for your answer.

   ii. (3 points) Write another version using, `sq_ii` using lists:foldl, lists:foldr, or lists:map.

   iii. (3 points) Write another version using, `sq_iii` using a list comprehension.

(c) Mind reading (4 points)

   i. (2 points) Why didn't I ask you to implement a version of `sum` using a list comprehension?

   ii. (2 points) Why didn't I ask you to implement the other kind of recursion for `sq`?
   I.e. why not write `sq_tr` if `sq` is head recursive, or `sq_hr` if `sq` is tail recursive?

Note: this really isn't a mind-reading exercise. Any reasonable answer will be accepted, even if it's not the one I was thinking of when I wrote the question. You just need to show that you're thinking about why you use particular Erlang constructs or not.

2. **GCD** (10 points):
Write a function, `gcd(A, B)` that returns the greatest common divisor of `A` and `B`. Your function should give the correct answer for any integers for `A` and `B` as long as they aren't both `0`, and fail if either `A` or `B` is not an integer, or if both `A` and `B` are `0`.

3. **What is π?** (28 points)
Pick two integers, `A` and `B`, uniformly at random with $1 \leq$ `A`, `B` $\leq N$. What is the probability that codeA and `B` are co-prime (i.e. that `gcd(A, B) == 1`)? In the limit that $N \to \infty$, this probability converges to $6/\pi^2$.

   (a) (8 points) Write a function, `p(N)` whose guard requires `N` to be a positive integer (i.e. `N > 0`) and returns the number of distinct pairs of integers, `{A, B}` with $1 \leq$ `A`,`B` $\leq N$ such that `A` and `B` are co-prime. For example, `p(6) -> 23`.

   (b) (4 points) Write a function, `pi(N)` that estimates the value of π based on the value returned from `p(N)`.

   (c) (10 points) To get a good estimate, we need a large value of `N`, and the run-time for `p(N)` grows quadratically with `N` (or you probably did something wrong). Now write `p(N, M)` that generates `M` samples of the form `{A, B}` with `A` and `B` integers that are uniformly and independently distributed with $1 \leq$ `A`,`B` $\leq N$ and returns the number that it generated with `A` and `B` co-prime.

   (d) (2 points) Write a function, `pi(N, M)` that estimates the value of π based on the value returned from `p(N, M)`.

   (e) (4 points) Try some tests of both `pi(N)` and `pi(N,M)`. Make a few observations about the quality of the estimates and the execution times. Nothing profound is required here. Just make it clear that you ran your code and thought a little about the results.

4. **Generators** (30 points)
Many languages provide iterators or generators for producing sequences of values. In Erlang, we can use list comprehensions to process lists of values, and functions such as lists:seq to generate lists of index values. However, it's annoying that lists:seq actually constructs the list and may use a large amount of memory, even if we don't need to have the entire list at once. For example,

```
lists:sum(lists:seq(1, N))
```

When I try it on my laptop (a 2015 Macbook Pro), I get a run time of about `N*36ns` `N` $\leq$ `10,000,000`, about that twice per value for `N = 200,000,000`, and about 11× slower per value when `N = 1,000,000,000`. Let's use processes to implement generators!

Here's the plan. The generator process will generate the sequence of values and send them to the consumer process. The generator sends a special atom `done` to indicate that all values have been sent. The consumer process will receive each value, process it, and continue. We want to keep it simple for the user; so, we'll make an example similar to lists:foldl.

In more detail, a generator function is of the form `GenFun(GenArg)`. If there is another value to send to the consumer, `GenFun(GenArg)` returns a tuple of the form `{V, NextArg}`, where `V` is the value to be sent to the consumer process, and `NextArg` is the argument for the next call to `GenFun`. Otherwise, the sequence is exhausted, and `GenFun(GenArg)` returns the atom `done`.

The consumer function is of the form `AccFun(Value, Acc) -> NewAcc`. Given the next value in the sequence, `Value`, and the current accumulator value, `Acc`, `AccFun(Value, Acc)` computes the next accumulator value. When all values from the generator have been consumed, our generator based fold will return the final value of `Acc`.

In the template file `hw1.erl`, I provide an example of using a generator with the function `gen_test(N)`. This function computes the sum of the first `N` integers. The generator function is


```
fun(I) when I =< N -> {I, I+1}; % {V, NextArg}
   (I) -> done; % sequence exhausted end
```

The generator is given an initial value for `I` of `1`. The consumer function, is


```
fun(V, Acc) -> V + Acc end % that was simple.
```

The consumer is given an intial value of `0` for `Acc`. The complete code for `gen_test` is

```
gen_test(N) when is_integer(N) ->
  gen_fold(fun(V, Acc) -> V + Acc end, 0, % the consumer, and its initializer
           fun(I) when I =< N -> {I, I+1}; % the generator...
              (I) -> done;
           end, 1).  % ...and its initializer
```

Your task, of course, it to write the function

```
gen_fold(ConsumerFun, ConsumerInit, GeneratorFun, GeneratorInit)
```

In the hopes of promoting learning rather than frustration with this assignment, I've sketched an implementation in the template code for `hw1.erl`. The function `gen_fold(ConsumerFun, ConsumerInit, GeneratorFun, GeneratorInit)` spawns a generator process, `GenProc`, and calls

```
gen_fold(Fun, Acc, GenProc)
```

The function `gen_fold/3` is a tail recursive function that sends `next` requests to `GenProc`. It `GenProc` replies with `{gen, GenProc, done}`, then `gen_fold/3` returns the last value from `Fun`, i.e. `Acc`. and `gen_fold/3` uses `Fun` to combine `V` with the current value of `Acc` to get the new `Acc` and continues.

Finally,

```
gen_produce(GenFun, GenArg)
```

handles interprocess communication for the producer. It waits to receive a message of the form `{next, ConsumerPid}` and then calls `GenFun` to get a new value or `done`.

Let's turn all of these function descriptions into a problem statement:

(a) (25 points) Implement (or complete the implementations) of the functions:

  i. `gen_produce/2` (10 points)
  ii. `gen_fold/3` (10 points)
  iii. `gen_fold/4` (5 points)

(b) (5 points) Are your functions `gen_fold/3` and `gen_produce/2` tail-recursive? Why is it a *very* good idea to write tail-recursive implementations of these functions? **Hint:** think about what happens if the user provided generator produces a *huge* number of values. **Note:** For full credit, your implementations of `gen_fold/3` and `gen_produce/2` must be tail-recursive. I'll take off even more points if you claim that they are tail recursive but they aren't.

(c) **Just for fun:** When I tested my implementation, I had a few surprises. First, `gen_test(N)` was about 400× slower than

```
lists:sum(lists:seq(1, N))
```


Erlang messages are supposed to be fast – what happened?! The issue appears to be with the lambda expression `fun(V, Acc) -> V + Acc end` and `fun(I) -> ... end`. In particular,

```
lists:foldl(fun(V, Acc) -> V + Acc end, 0, lists:seq(1, N))
```


is about 50% *slower* than the `test_gen` version. Why? I'm don't know. If you want to experiment with it, have fun. The Erlang installation on the CS department machines is release OTP/22. Later releases may have fixed this. I plan to install OTP/25 on my laptop and give it a try. If lambdas are still slow, I'll send a report to the Erlang developers.

Why?

1. **Maps, Folds, and List Comprehensions:** These are intended to be easy problems to let you check to make sure that you're comfortable with the Erlang features that we'll use throughout the first half of the class.

2. **GCD:** Another simple example, but one where you need to think a bit about the logic of the function. If you haven't done programmed in a functional language before, this is a simple "hello world" to write a recursive function. I assume that everyone has seen (or can look up) Euclid's GCD algorithm. Yeah, I just cited wikipedia rather than the original Greek – deal with it.

3. π**:** A fun (maybe that depends on your sense of fun) example from number theory. The problem is embarrassingly parallel, and a great candidate for the "reduce" algorithm that we saw with count 3's. You can look forward to a parallel π problem on HW2.

4. **Generators:** Erlang is functional programming plus message passing. We're using it here because it lets us get into the message passing paradigm quickly and fairly easily. Given that, you need to see Erlang processes and messages. This problem does that. As described in *Learn You Some Erlang*, the typical Erlang program makes extensive use of processes and message, and it hides most of the details behind functions that provide a nice abstraction for the problem you want to solve. This problem gives an illustration of that for

The Library, Errors, Guards, and other good stuff

The CPSC 418 Erlang Library: your code *must* run on the CS department linux machines.

This assignment doesn't explicitly use the course library, but some functions like those in the `time_it` module may be helpful. To access this library from the CS department machines, give the following command in the Erlang shell:

```
1> code:add_path("/home/c/cs-418/public_html/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/bin/erl erl -eval 'code:add_path("/home/c/cs-418/public_html/resources/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.students.cs.ubc.ca/~cs-418/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.

Compiler Errors: if your code doesn't compile, it is likely that you will get a zero on all coding questions. 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 instructor. 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. For example, to have separate cases for `N > 0` and `N < 0`.
- 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. an exported function) it's good to have your assumptions about arguments clearly stated. Ideally, you this with guards, that is great.
  - Often, a function can only be implemented for *some* values of its arguments. For example, we might have:

```
sendSquare(Pid, N) -> Pid !  N*N.
```

A call such as `SendSquare([1, 2, 3], cow)` doesn't make sense. Bad calls should throw an error (e.g. a `badarg` error). Please, don't silently ignore bad arguments, for example

```
sendSquare(Pid, N) ->
  if is_pid(Pid) and is_number(N) -> Pid ! {square, N*N};
     true -> "messed up actual arguments" % Don't do this.
  end
end.
```

The caller might very well ignore the return value of `SendSquare`, and `Pid` might end up blocking, waiting for a message that will never arrive. Furthermore, putting tests to see if the return value is an error code is so C, but throwing explicit exceptions and writing error handlers (if you want to do something other than killing the process that threw the error) is so much easier to write, read, and maintain.

We will test your code on bad arguments and make sure that an error gets thrown.

- 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, 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 you can use comments; comments don't slow down the code. Any exported function should throw an error when called with bad arguments.

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.