COMP 6411: PROJECT DESCRIPTION

Important Info:

1. *Projects can be done either individually or in a group of 2 or 3 students (extra work is required for group submissions).*

2. *Projects must be submitted on time in order to received full value. Appropriate late penalties (< 1 hour = 10%, < 24 hours = 25%) will be applied, as appropriate. . You **MUST** verify the contents of your assignment AFTER you submit. You will be graded on the version submitted at the deadline – no other version will be accepted at a later date.*

3. *The graders will be using a standard distribution of Erlang (the version in the docker distribution is OTP 23 (or higher), but any recent version should be fine). You cannot use any Erlang libraries and/or components that are not found in the standard distribution. The graders will not have these libraries on their systems and, consequently, your programs will not run properly.*

OBJECTIVE: For the final project, we will take a look at Erlang. Note that while Erlang is a functional language, our focus here is the concurrency model provided by Erlang. In particular, this assignment will require you to gain some familiarity with the concept of message passing. In fact, Erlang does this more effectively than any other modern programming language.

Briefly, your objective is to create a (really) simple gaming tournament. Specifically, you will be simulating the classic Rock/Paper/Scissors game (RPS). If you haven't heard of this game before, it will be described below. The basic idea is that you will be given a list of players, each with a unique number of game *credits*. Players will interact with one another, making random RPS guesses, and wins and losses will be determined by a "master" process. Players will gradually exhaust their game credits and will be forced to leave the game. Eventually, only one player will remain – this person will become the tournament winner. That's it.

DETAILS: So now for the details. To begin, you will need a collection of players. These will be supplied in a very simple text file. While Erlang provides many file primitives for processing disk files, the process is not quite as simple as Clojure's `slurp()` function. So your input file will contain records that are already pre-formatted. In other words, they are ready to be read directly into standard Erlang data structures.

A sample player file might be:

```
{sam,26}.
{jill,12}.
{ahmad,17}.
```

In other words, the file simply contains a set of Erlang *tuples*. The first element of each tuple is an *atom* (note that atoms start with a lower-case letter). The atoms will represent the names of the players (so no string processing is required). You will see that each atom/label is associated with a number. This is the total number of game credits that they will have when the tournament begins.

To read the file, we use the `consult()` function in the `file` module. This will load the contents of the file directly into an Erlang structure (i.e., a list of tuples). Note that NO error checking is required. The text files are guaranteed to contain valid data.

Because you (and the graders) will want to run the program with different input files, we don't want to hard-code the file name. Instead, it will be passed as a command line parameter. While this isn't especially difficult to do, I don't want you to waste time trying to get this to work. So after compiling your source files with `erlc`, you will run your program with the following invocation on the command line:

```
erl -noshell -run game start p1.txt -s init stop
```

This will run a program in which the main module (i.e., source file) is called `game`. Within `game`, the initial function (e.g., like *main* in Java) will be called `start`. The `start` function will accept a list of parameters, in this case just the file name *p1.txt*. This, of course, is the name of the player file. You can call it anything you like (the graders will provide their own test files).

Inside the `game.erl` source file, you will have the following code at the top of the file:

```
-module(game).
-export([start/1]).

start(Args) ->
      PlayerFile = lists:nth(1, Args),

      {ok, PlayerInfo} = file:consult(PlayerFile),
      …
```

Note that the **…** at the end simply means that the `start` function will have more logic after this point. This is just the code to read the data files.

In summary, the code above provides the module name for the current source file (`game`), and it indicates that the module exports a single function called `start`. The `start` function will take one parameter which, in this case, is going to be a list of the command line args: `["p1.txt"]`. We just

assign "p1.txt" to `PlayerFile` and then use the `consult` function in the `file` module to read the content of the file and bind it to a label. So `PlayerInfo` will be a list of player tuples `[{sam,26}, {jill,12}, {ahmad,17}]`. At this point, you have all of your input data and you are ready to proceed.

So your job now is to take this information and create an application that models the tournament environment. Because each player is a distinct entity in this world, they will be modeled as separate tasks/processes. When the application begins, it will therefore generate a new process for *each* player. Because you do not know how many players there will be (there could be dozens or even hundreds), or even their names, you cannot "hard code" this phase of the application.

The player tasks will then start up and begin the tournament.

You may want to make each new player sleep for 200 milliseconds or so, just to make sure that all the player tasks have been created and are ready to be used - this can be done trivially with an expression like `timer:sleep(200)`. Otherwise, the application may crash if a player tries to contact another player that does not yet exist.

Okay, so now for the game logic itself:

1. Players will make requests to one another to begin a new RPS game. To do so, they will randomly select one of the players from the full list (excluding themselves). In order for this to work, of course, the players must be provided with the complete player list.

2. To ensure that everything doesn't happen at once, each request is made after a random delay of between 10 and 100 milliseconds. Again, this can be done with something like `timer:sleep(N)`, where `N` is a number between 10 and 100.

3. Since all players can make requests, then it should be obvious that all players can receive invitations as well. When a player receives an invitation, it will send a confirmation back to the first player to confirm that they will have a game (unless the player has lost all of his/her credits, in which case they must reject the invitation…as described below).

4. When the first player receives confirmation that the second player has accepted their request, they will send a message to the master process to ask for a new game to be scheduled. Note that the *master* process is just the initial Erlang process – the one that created all of the player tasks. It is NOT an additional process called "master".

5. When the master receives this message, it will create a new ID for the game. This is just a unique integer, starting from 1, that will be incremented each time a new game request is made. This ID must be sent to both players so that they know who they are playing in this round.

   IMPORTANT: It might not be obvious thus far, but a player can be involved in multiple games at the same time, since they can receive multiple requests from other players during the time they are making their own requests. In fact, it is possible to play multiple games

with the same player at the same time. Moreover, some games will take longer than others so games may overlap. This is why we need unique ID numbers for each game.

6. Once the ID number arrives, each player will randomly select their RPS "move" for this game. This is probably a good time to explain how RPS works, just in case you have never heard of it before. So the idea is as follows:
    a. RPS is a two-person game in which simultaneously reveals one of three possible "weapons": a rock, a piece of paper, or scissors. The rules for victory are simple:
        i. Rock/Scissors: rock wins because a rock breaks scissors
        ii. Rock/Paper: paper wins because paper covers a rock
        iii. Scissors/Paper: scissors win because they cut paper
    b. Based upon the options above, victory is obvious, unless…
    c. If there is a tie (e.g., rock/rock), then the process is repeated as many times as necessary, until a winner is found.

7. So back to the game. Clearly, the players can't be trusted to determine their own winner. So each must send its random selection (rock, paper, or scissors) to the master process. Because these two RPS moves are coming from two different processes, they will arrive at different times. Once the second message arrives (i.e., one with the same game ID), the master will determine the winner and record this information for future use.

    Important: The master does not "block" while waiting for the second message. When the master receives the first message, it will immediately wait for game messages from other players. Only when the second, matching, RPS move arrives will the master determine a winner for that game.

8. In practice, the players don't need to be informed about the outcome of an individual game since they will just keep playing by making/answering requests for new games. However, there are two cases when a response is definitely required:
    a. A tie (e.g., rock/rock). In this case, the master must send a message back to both players to indicate that a tie has occurred. When this happens, the players will randomly choose a new move and send it back to the master (along with the same ID). This process will be repeated until a victory occurs.
    b. Eventually, a player will exhaust their credits and be forced to stop. So what is a credit? In this tournament, credits are not used for new games. Instead, they represent the number of games that you can lose before you are disqualified. If you have 4 credits at the start of the tournament, for example, you can only lose up to 4 games.

9. The master process must keep track of player losses. Once a player's credits reach 0, the master must inform that player that they are disqualified. At this point, the disqualified player cannot (i) make any new game requests or (ii) accept any new invitations. There are two important points to note:
    a. If the player is currently playing other games, those games should continue to completion. Trying to cancel partially completed games would just add unnecessary complexity.

b. When a player is disqualified, they must continue to listen to messages, since other players may still send them requests for new games – which MUST be denied. In this case, the active player should avoid sending a request to that same player again.

10. Eventually, the master process will recognize that only one player is left (all others have no credits remaining). At this point, the master will notify all players that the tournament is over. All player tasks will then properly terminate (i.e., they don't crash or run forever).

11. Finally, the master will display a tournament summary and the application will finish (a standard command prompt will appear).

And that's it. Woo hoo.

Of course, we need a way to demonstrate that all of this has worked properly. To begin, it is important to understand that this is a multi-process Erlang program. The "master" process will be the initial process that, in turn, *spawns* processes for each of the players. So, in our little example above, there will be 4 processes in total: the master and 3 players.

To confirm the validity of the program, we need a series of info messages to be printed to the screen. This will include:

- A new game is scheduled and an ID is assigned
- An individual game has completed and a summary is provided
- A player has used his/her credits and is disqualified

In effect, these messages provide a real-time transaction log as the program is running.

IMPORTANT: The "master" process is the only process that should display anything to the screen. Player processes NEVER do any I/O themselves. In production applications this would also be true since (1) concurrent I/O from multiple tasks would get interleaved together, creating an unreadable mess, and (2) large applications would often use networked/distributed nodes that would not even use the same console/screen. The graders will check your source code to ensure that all printing is done from the master.

On the next page, we can see partial output for a small example.

```
** Rock, Paper Scissors World Championship **

Starting game log...

+ [1] new game for sam -> jill
$ (1) sam:rock -> jill:paper = jill loses [11 credits left]
+ [2] new game for sam -> ahmad
+ [3] new game for jill -> john
$ (2) sam:paper -> ahmad:scissors = sam loses [25 credits left]
+ [4] new game for ahmad -> sam
+ [5] new game for sam -> ahmad
$ (4) ahmad:scissors -> sam:rock: ahmad loses [16 credits left]
$ (3) jill:scissors -> sam:paper: sam loses [24 credits left]
+ [6] new game for sam -> jill
$ (5) sam:rock -> ahmad:rock, sam:rock -> ahmad:paper = sam loses [23 credits
left]

…

+ [36] new game for sam -> jill
+ [37] new game for ahmad -> sam
- (36) sam:paper -> jill:paper, sam:rock -> jill:scissors = jill loses [0
credits left]
+ [38] new game for ahmad -> sam

…
+ [45] new game for sam -> ahmad
- (45) sam:paper -> ahmad:rock = ahmad loses [0 credits left]


We have a winner…


** Tournament Report **

Players:
 sam: credits used: 16, credits remaining: 10
 jill: credits used: 12, credits remaining: 0
 ahmad: credits used: 17, credits remaining: 0
 -----
 Total games: 45

winner: sam


See you next year...
```

Let's review the output. After the application starts up, we begin to see messages indicating that new games are about to begin. These are the lines like:

```
+ [1] new game for john -> sue
```

Here, the "+" indicates that this is a new game, while the "[1]" tells us that this is game #1. The remaining text shows that this is a game between john and sue that was initiated by john.

Other lines summarize a completed game:

```
$ (2) sam:paper -> ahmad:scissors = sam loses [25 credits left]
```

In this case, the "$" symbol indicates a game victory. The "(2)" shows the ID number of the game. This will match a previous "new game" message. We then see a summary of the game play: `sam:paper -> ahmad:scissors.` Sam has chosen "paper", while ahmad has chosen "scissors". This implies that sam loses, leaving him with 25 credits.

When ties occur in games, multiple moves will be displayed, as in:

```
$ (5) sam:rock -> ahmad:rock, sam:rock -> ahmad:paper = sam loses [23 credits
left]
```

Here, sam and ahmad both played "rock" initially, so another round was required in order to produce a victory.

In the final group of messages, the master indicates that certain players have been disqualified:

```
- (36) sam:paper -> ahmad:rock = ahmad loses [0 credits left]
```

At the end, there are no players remaining (jill and ahmad have both been disqualified) and the tournament can come to a conclusion. At this stage, the report is printed. The content is fairly simple but it is worth noting that the contents of the report must be completely consistent with the log listing. This is fairly easy to confirm on a small example.

As a final point, the use of randomness guarantees that the output will be slightly different on each run. In general, the player starting with the highest number of credits will be the winner. However, if credit counts are more similar, and the number of players is larger, it will be hard to predict the outcome in advance.

OTHER THINGS: As noted, this is an assignment that focuses on concurrency, not general programming issues. Consequently, I want to minimize the time spent on other things. So please keep the following things in mind.

- With Erlang, you send messages to other process by using their process IDs (this is the value returned by the `spawn` function. In some situations, you can just bind this value to a label and then use this later in a send. However, when processes have been spawned in another task, you don't necessarily know what their IDs are. Erlang provides a very simple mechanism for this. Specifically, you can *register* a name/processID pair using the `register(name, ID)` function. Later, you can use the `whereis(name)` lookup function - in another process - to get the processID of your target process.
- The `self()` function gives the process ID of the current process

- Erlang uses *if* for its conditional processing. Note that at least one of the conditional checks MUST return true; otherwise, you will get a runtime error. If needed, you can use something like the expression below. Basically, this says that if the first condition(s) isn't true, the default is just to match "true" and essentially do nothing.
```
if
  X > 3 -> do_something();
  true -> false
end.
```
- The `lists:foreach` function is a way to quickly process a list by performing a function on each of the values in the list.
- The `lists:foldl` function is Erlang's equiavalent to Clojure's `reduce` function (i.e., reduce all values in a list to a single value with a function like *sum*).
- While not absolutely required, it is good practice to send your message content as a tuple. Use an atom/label as the first element of the tuple to easily distinguish one message from another!
- Erlang has modules for *lists* and *maps* (and other things). Each contains many functions for manipulating the associated data structures. The online docs give many examples of their use.
- Erlang's *list comprehensions* can be used to easily build lists from other lists. For example, the code below makes a new list from the second element of each tuple in the original list:
`FruitList = [Fruit || {_ , Fruit} <- GroceryList]`
- Erlang allows you to use many expressions in a single function. They just execute one after another. You simply have to separate each expression with a comma. The function itself will end with a period.
- Some expressions – like `receive` and `if` expressions – use a semi-colon to end each block (though no semi-colon is used after the final block).
- The `rand` module is used for random number generation. `rand:seed(exsss)` can be used to initialize the generator and `rand:uniform(myInteger)` can be used to get a random number from 1 to myInteger.
- The function `length(myList)` can be used to get the length of a list. `lists:nth(n, myList)` can be used to extract the value at position `n`. Note, however, that list indexing in this case begins at 1, not 0.
- Basic printing can be done with `io:fwrite()` or `io:format()`. These functions work very much like `printf()` in the C language. You provide a list of values that are mapped into a formatted string. Again, the online docs provide many examples.

I think that covers most of the obvious issues. If you build on these ideas and utilize the documentation on the main Erlang website, which is pretty good, you should be able to mostly focus on the logic of the communicating tasks.

GROUP VERSION If you are working in a group, you will create the same Erlang application. However, you will also be creating a second version of the application in Java. The comparable Java program will produce exactly the same result.

You will begin by reading the same data file. In this case, you will use Java's IO classes to extract the player data. Once you have the data, you will replicate the functionality of the Erlang program.

In this case, you will use Java's basic *thread* mechanism to create individual threads to represent each player. So just like the Erlang app, this will be a multi-threaded program. Each player will be constructed as a thread, and the player threads will exchange the same info (requests, confirmation, RPS moves) with both each other and the master thread, using the same logic/order described in the Erlang description above. Output to the screen (log and final report) will also take the same form.

Important: You are free to implement your Java code however you like. However, you can only use Java classes contained in the Java Standard libraries - NO external third-party libraries, including any message frameworks that do the communication work for you. The real purpose of the Java component is to see how a general imperative language compares to a language designed for a particular purpose.

GRADING The graders will provide their own player text files. A very simple file will be used to assess basic functionality (and award most of the point value). Then some larger examples will be used to see if your application performs well under different conditions. As noted, the test files will ALWAYS use valid data. Again, no error checking of any kind is required on the input. All game credits will be recorded as positive, non-zero integers. Basically, if your code works well in your own testing, it should work properly for the graders.

Note that Erlang automatically uses the current folder when searching for your modules. So nothing special has to be done to find them. For groups, your Java program will also read its input from the current folder and must compile and run from the command line without any additional configuration settings or files.

Finally, you do not have to provide any error checking on the file names passed on the command line. It is the markers' job to correctly input the names of their test files.

In terms of group grading, please note that although two programs will be written, they will not be given equal weight. This is primarily an Erlang project and, as such, the Erlang application will receive 80% of the total grade. The remaining 20% will be associated with the Java program. So, if you want a good grade, you cannot simply do the Java version and quickly throw together a non-functional Erlang application

DELIVERABLES: Your Erlang submission will have just 2 source files. The "main" file will be called `game.erl` and will correspond to the master process. The second file will be called `player.erl` and will include the code associated with the "player" processes. Module names will be identical to the file names (minus the .erl extension). Do not include any data files, as the markers will provide their own.

For groups, the Java version can have multiple source files, depending on how you choose to organize your code. However, the main “driver” class must be called `game.java`. It will have the *main* method and will be the entry point for running and testing your code.

Before preparing your submission, please create a simple README.txt file. Inside, indicate whether this is a solo project or a group project. For group projects, include the name(s) of the other member(s). Only one submission on Moodle will be made for the group and each person in the group will receive the same grade.

Once you are ready to submit, place the README.txt file and the .erl (and possibly .java) source files into a zip file. The name of the zip file will consist of "project" + last name + first name + student ID + ".zip", using the underscore character "_" as the separator (for groups, the last name will coincide with the name of the person doing the actual submission). For example, if your name is John Smith and your ID is "123456", then your zip file would be combined into a file called project_Smith_John_123456.zip". The final zip file will be submitted through the course web site on Moodle. You simply upload the file using the link on the project web page.

Good Luck