1 Lab 6 Erlang

Date: May 6, 2021

This document first describes the aims of this lab followed by exercises which need to be performed.

1.1 Aims

The aim of this lab is to introduce you to Erlang. After completing this lab, you should have some familiarity with the following topics:

- Pattern matching in Erlang.
- Decision making in Erlang functions.
- First class functions in Erlang.
- Erlang concurrency.

1.2 Exercises

1.2.1 Starting up

Follow the *provided directions* for starting up this lab in a new git `lab6` branch and a new `submit/lab6` directory.

For this lab, we will be using the Erlang shell. You can start it by typing erl at the Unix shell prompt. To terminate it type a ^C followed by an `a` to abort, or the command `q().` (note the period).

All functions should be defined in a file `lab6.erl` which should be submitted along with the usual log. You are being provided with a starting lab6.erl file. Copy that file over to your `submit/lab6` directory along with a `.gitignore` file.

```
$ cd ~/i471?/submit/lab6
$ cp ~/cs471/labs/lab6/lab6.erl .
$ cp ~/cs471/labs/lab6/.gitignore .
```

While in the Erlang shell, you can compile your `lab6.erl` using the following (note the terminating period):

```
N> c(lab6).
```

[The Erlang shell prompt includes the command number; this document shows those prompts simply as `N>`. Usually, this document does not show the output of the shell command.]

The compilation will produce a `lab6.beam` file which is a binary for the Erlang abstract machine. The `.gitignore` file you copied over will prevent `*.beam` files from being submitted.

1.2.2 Exercise 1: Erlang Pattern Matching

Erlang supports Prolog-like pattern-matching. Within a clean Erlang shell type in the following:

```
N> X = 123.
```

This will succeed because the semantics of `=` is to **evaluate** the expression on the RHS and then match that result with the LHS. So the above match evaluates the RHS `123` and matches that result with the unbound variable `X` on the LHS, initializing `X` to `123`.

Now type:

```
N> 123 = X.
```

This too will succeed because the match will evaluates its RHS consisting of the bound variable `X` to `123` and matches that with its LHS `123`.

Now try:

```
N> 123 = Y.
```

That should fail with an error since when the match attempts to evaluate the RHS it encounters an unbound variable `Y`.

[So Erlang's `=` is similar to Prolog's `is/2` which also evaluates its RHS and matches that result with its LHS.]

In addition to primitive data like numbers and atoms, Erlang also supports lists similar to those of Prolog and brace-enclosed tuples.

Try the following to understand how you can use pattern matching to extract the components of complex data:

```
N> Shapes = [ { square, 2 }, { circle, 1 }, { square, 1 } ].
N> [{_, Side1}, _, {_, Side3}|_] = Shapes.
N> Sides = [ Side1, Side3 ].
N> Sides.
```

Exercises

Set up the following data:

```
N> Grades = [ {bill, 82}, {sue, 95}, { john, 85} ].
```

1. Use pattern matching to extract the third grades `{ john, 85 }` to a variable `Grade3`. Display it to ensure that you have it correct.

2. Use pattern-matching to extract the points of the three grade items in `Grades` and set a `Points` variable to to the 3-element list `[82, 95, 85]` containing the extracted points.

   Note that this may take multiple steps and need the definition of auxiliary variables.

3. Create a variable `Grades2` having contents just like `Grades` but the points for each item should be incremented by 2 points; i.e. you should have `Grades2` set to `[ {bill, 84}, {sue, 97}, { john, 87} ]`.

You need to do all the above exercises by extracting the data from `Grades` using pattern-matching, not by simply typing in the data.

1.2.3 Exercise 2: Erlang Functions

Terminate your Erlang shell from the previous exercise and start a new one.

In this exercise, we will look at different constructs which can be used for decision making within an Erlang function:

Pattern Matching to Select Function Clauses Add the following function to your `lab6.erl` which calculates the perimeter of a shape:

```
perimeter({square, Side}) ->
  4 * Side;
perimeter({circle, Radius}) ->
  2 * 3.14159 * Radius.
```

The function is written using separate clauses for each shape, with pattern matching used to select the applicable clause. Test out the function:

```
N> lab6:perimeter({circle, 1}).
N> lab6:perimeter({square, 2}).
N> lab6:perimeter({rectangle, 2, 3}).
```

The last expression will result in an error because there is no clause for a `rectangle`.

Using Guards to Select Function Clauses Now write the function somewhat artificially using a single clause which uses a `when` guard (similar to the `|`-guards in Haskell):

```
guard_perimeter({Type, L}) when Type =:= square ->
  4 * L;
```

```
guard_perimeter({Type, L}) when Type =:= circle ->
  2 * 3.14159 * L.
```

Add the above to your `lab6.erl` file, compile and test.

Tests using if Within the Body You can move the guards into the body using an `if`:

```
if_perimeter({Type, L}) ->
  if Type =:= square -> 4 * L;
     Type =:= circle -> 2 * 3.14159 * L
  end.
```

Add the above to your `lab6.erl` file, compile and test.

Pattern Matching Using case Within the Body Use a `case` expression within the body to do pattern matching:

```
case_perimeter(Shape) ->
  case Shape of
    {square, Side} -> 4 * Side;
    {circle, Radius} -> 2 * 3.14159 * Radius
  end.
```

Add the above to your `lab6.erl` file, compile and test.

Exercises:

1. Based on the above guard example, write a function `letter_grade(¬ Points)` which returns the atom:

   ```
   'A' when 90 < Points
   'B' when 80 < Points and Points =< 90
   'C' when 70 < Points and Points =< 80
   'D' when 60 < Points and Points =< 70
   'F' otherwise.
   ```

   Hint: Erlang runs a guard only if both pattern matching and guards for all earlier clauses have failed. So you can write the clauses in a certain order and avoid checking for conditions which are implied by the failure of the earlier guards. So assuming that the clause for a `'B'` follows the clause for an `'A'`, your can write the guard for `'B'` as simply `when 80 < Points`.

   Alternately, you can write the clauses so that they do not depend on order; in that case, the guard for `'B'` would be written as `when 90 =< Points, Points < 80`.

Test with suitable data:

```
N> lab6:letter_grade(91).
N> lab6:letter_grade(90).
N> lab6:letter_grade(81).
N> lab6:letter_grade(80).
N> lab6:letter_grade(75).
N> lab6:letter_grade(65).
N> lab6:letter_grade(55).
```

2. Write a second function `if_letter_grade(Points)` which works identically to `letter_grade()` but is implemented using a single clause which uses an `if` within the body.

   Recompile and test as in the previous exercise.

1.2.4 Exercise 3: First-Class Functions

Since Erlang is a functional programming language, it supports anonymous functions and higher order functions like `map()` and `fold()`. Specifically:

- Anonymous functions are written using the syntax
  ```
  fun (Arg1, Arg2, ...) -> Expr end
  ```
- Functions which operate on lists are available in the lists module.

Restart your Erlang shell and compile `lab6.erl`:

Grab hold of shapes data:

```
N> Shapes = lab6:shapes_data().
```

Now add the following to your `lab6.erl` file:

```
shape_types(Shapes) ->
  lists:map(fun({Type, _}) -> Type end, Shapes).
```

This uses `lists:map()` with an anonymous function which uses pattern matching to extract the first component of each `Shape`-tuple.

Test using:

```
N> lab6:shape_types(Shapes).
```

The `lists:map()` function along with `lab6:perimeter()` can be used to get a list containing the perimeters of a list of shapes:

```
perimeters(Shapes) ->
```

```
lists:map(fun perimeter/1, Shapes).
```

Note the somewhat funky syntax for referring to a previously defined function.

Compile and test `perimeters()`.

The `lists:foldl()` and `lab6:perimeters()` functions can be used to sum the perimeters of a list of shapes:

```
sum_perimeters(Shapes) ->
  Perims = perimeters(Shapes),
  lists:foldl(fun (P, Acc) -> P + Acc end, 0, Perims).
```

We can get the average perimeter of a list of shapes using:

```
average_perimeter([]) -> 0;
average_perimeter([_|_]=Shapes) ->
  sum_perimeters(Shapes) / length(Shapes).
```

The function takes care to avoid a divide-by-0 error. Note the pattern matching in the second clause to make the clauses mutually exclusive, while also setting the Shapes variable to the overall non-empty list.

Compile and test.

Exercises:

Add in the following to your Erlang shell to give you some new data to work with:

`N> Grades = lab6:grades_data().`

Use `map()` and `foldl()` to implement the following functions in `lab6.erl`:

1. A function `grade_points(Grades)` which returns a list of all the points for all entries in `Grades`.

   ```
   N> lab6:grade_points(Grades).
   [82.0,95,85,73,65,55]
   ```

2. A function `letter_grades(Grades)` which returns a list of pairs `{ Name¬ , LetterGrade }` where `LetterGrade` is the letter grade for the points associated with `Name` in `Grades`.

   Hint: use `map()` along with the previously implemented letter_grade().

   ```
   N> lab6:letter_grades(Grades).
   [{bill,'B'},
    {sue,'A'},
   ```

```
        {john,'B'},
        {joe,'C'},
        {mary,'D'},
        {tom,'F'}]
```

1.2.5 Exercise 4: Concurrency

The raison d'etre for Erlang is concurrency. In this exercise, we will create a server which will store some data. Clients can run arbitrary functions on the stored data.

Add the following `data_server()` to your `lab6.erl` file:

```
data_server(Data) ->          % Data is stored data
  receive                     % receive a message
    { ClientPid, Fn } ->      % msg contains function Fn
      Result = Fn(Data),      % run arbitrary function on Data
      %io:format("Result is ~w\n", [Result]),
      ClientPid ! { self(), Result }, % send Result to client
      data_server(Data);      % loop back
    stop ->                   % got stop message
      true                    % terminate server
   end.
```

Then add a client function:

```
data_client(ServerPid, Fn) ->
  ServerPid ! { self(), Fn }, % send Fn to server
  receive
    { _, Result } -> Result    % return Result
  end.
```

Finally, add some control functions:

```
start_data_server(Data) ->
  spawn(lab6, data_server, [Data]).

stop_data_server(ServerPid) ->
  ServerPid ! stop.
```

Start a shapes server in the Erlang shell:

```
N> c(lab6).
```

```
N>  Shapes = lab6:shapes_data().
N> PID1 = lab6:start_data_server(Shapes).
```

The last line should have printed the PID of the server. This PID has been captured in the variable `PID1`.

Now use the client to run previously defined functions on the data stored by the server:

```
N> lab6:data_client(PID1, fun lab6:average_perimeter/1).
N> lab6:data_client(PID1, fun lab6:shape_types/1).
```

Now run the identity function to get back the data stored by the server:

```
N> lab6:data_client(PID1, fun (X) -> X end).
```

Finally, run another function which returns a new list of shapes having 3 times the dimensions of the shapes stored by the server:

```
N> F = fun ({T, S}) -> {T, S*3} end.
N> lab6:data_client(PID1, fun (Shapes) -> lists:map(F, Shapes) end).
```

Stop the server:

```
N> lab6:stop_data_server(PID1).
```

The above shows that functions in Erlang are truly first class in that they can even be transmitted between multiple processes.

Exercises:

1. Start a data server containing the `Grades` data.
2. Use the client to run the `letter_grades()` function on the server's data.
3. Use the client to retrieve the stored data.
4. Stop the grades server.