CPSC 418: Parallel Computing
In Class Worksheet

Winter 2016–2017 Term 2

Introduction to Erlang

solution set

Name: Mark Greenstreet Student Number: 11111111

1. **Read You Some Erlang.** What is the value of each Erlang expression? (Don't just type it in—try to reason out the answer.)

   (a) `42 =:= 6 * 7.0 or 1 <= 2.`

   Ans. This code causes a syntax error: `<=` should be `=<`. Erlang uses a different symbol for "less-than-or-equal-to" than many common languages (C, Java, Python, ...). We can "fix" that and try

   ```
   42 =:= 6 * 7.0 or 1 =< 2.
   ```

   This also generates a syntax error, but the reason is more subtle. For grading purposes, we *might* expect you to know about `=<` on an exam. The rest of this answer isn't examinable material, but it may make it easier for you to write Erlang code. The syntax error in the second part is because of Erlang operator precedence. The comparison operators have lower precedence than boolean operations such as `or`. Here's the "fully parenthesized" version of the expression above following Erlang's precedence:

   ```
   42 =:= ((6 * 7.0) or 1) =< 2.
   ```

   If we try

   ```
   ((6 * 7.0) or 1).
   ```

   We get a "bad argument" error because `or` doesn't apply to numerical values. That makes sense. Why does

   ```
   42 =:= ((6 * 7.0) or 1) =< 2.
   ```

   produce a syntax error? That's because the comparison operations in Erlang can be "chained". For example,

   ```
   2 < 3 < true.
   ```

   is a syntax error, but `(2 < 3) < true.`
   evaluates to `false` because `(2 < 3)` evaluates to `true`, and `true < true` evaluates to false.

   Back to our original problem, we can just add parentheses to get the "intended" expression: `(42 =:= 6 * 7.0) or (1 =< 2).`
   has a value the value `true`.

   (b) `tl([1|2]).`

   Ans. The correct answer is:

   Make Ian sit in a comfy chair until he promises to never ask this question again.

I didn't know you can make a list in Erlang whose tail is not a list this. Now I'm fielding questions about it on piazza.

OK, the official answer is 2. Erlang allows the tail of a list to be any Erlang term. In practice, we will **always** construct "properly terminated" lists. If you write code that ends a list with something other than `[]`, expect us to deduct points.

Now, for the geeky stuff. I'm guessing the `[Head | NonList]` is a hangover from earlier functional languages. Lisp is the great* grandparent of pretty much all functional languages. Lisp allows the tail of a list to be a non-list because early implementations had to run on machines with very small amounts of main memory (e.g. 1Kbyte). Rather than "wasting" a whole word with `null`, early Lisp versions did some bit-fiddling to distinguish list pointers from other values, and allowed an arbitrary value in the tail of a list. Of course, programmers found ways to use this; so, "modern" Lisp dialects (i.e. Common Lisp) still support this, and I've seen Lisp APIs that rely on this feature.

OTOH, I have **never** seen this "feature" used in Erlang. Maybe it is. If you're really worried about memory usage, Erlang has "binaries". You might recall that we said you could skip that section of *Learn You Some Erlang*. That's because we're using Erlang to demonstrate fundamental concepts of parallel programming. If you really want to develop serious Erlang applications, then you'll want to go back and read that section, but you don't need it for this course.

(c) `tl([1,[ 2, 3, 4 ]]).`

Ans. The term `[1,[ 2, 3, 4 ]]` is a list of two elements: the first element is `1` and the second element is `[ 2, 3, 4 ]`. The value of `tl([1,[ 2, 3, 4 ]])` is the list consisting of the second through last elements of `[1,[ 2, 3, 4 ]]`. As noted above, `[1,[ 2, 3, 4 ]]` is a list of two elements; so the second element is the last element. That means that the value of `tl([1,[ 2, 3, 4 ]])` is a list of one element, where that element is `[2, 3, 4]`. That brings us to the answer. The value of `tl([1,[ 2, 3, 4 ]])` is

[[2, 3, 4]]If that outer pair of `[]`s bothers you, think if we asked for `tl[1,2]`. The list `[1,2]` has two elements, and `tl[1,2]` is the list of one element where that element is `2`. So, `tl[1,2] = 2`. That's not surprising. Now, replace `2` with `[2,3,4]` and you get the answer to the original question.

(d) `length([ 1 | [ 2 | [ 3, 4, 5 ]]]).`

Ans. We'll evaluate this one from the inside of the square-brackets to the full expression.

`[ 2 | [ 3, 4, 5 ]]).`

is a list whose head is `2` and whose tail is `[3, 4, 5]`. We can rewrite the original expression as

`length([ 1 | [ 2, 3, 4, 5 ]]).`

Using the same reasoning again, we get `length([ 1, 2, 3, 4, 5 ]).`

That's the length of a list with five elements. The answer is `5`.

(e) `x = [ 1 | [ [ 2, 3, 4 ], 5 ]].`

Ans. This generates an error because `x` is an atom. We are asking Erlang to "match" the list `[1 | [[2, 3, 4], 5]]` They don't match.

The "solution" is to use a real, Erlang variable. Erlang variable names must start with a capital letter of `_`. So,

```
X = [ 1 | [ [ 2, 3, 4 ], 5 ]].
```
is just fine.

Now what happens if after binding `X` to `[ 1 | [ [ 2, 3, 4 ], 5 ]].` we give the command
```
X = [ 1 , [2, 3, 4], 5].
```
That works just fine because `[ 1 | [ [ 2, 3, 4 ], 5 ]].` and `X = [ 1 , [2, 3, 4], 5].` are two ways of writing the same list. If instead we tried:
```
X = [ 1 , 2, 3, 4, 5].
```
We get a "no match" error again because Erlang can't match the value of the right side, `[ 1 , 2, 3, 4, 5].` to the value that `X` already has, i.e. `[ 1 | [ [ 2, 3, 4 ], 5 ]]`.. When using the Erlang shell, this can be a hassle. During a long session, we have keep thinking of new names for variables. Fortunately, Erlang lets us ask the shell to "forget" a variable. We can give the command:
```
f(X).
```
and now `X` is "forgotten". Having erased the binding for `X`, we can try
```
X = [ 1 , 2, 3, 4, 5].
```
again, and this time it works. Note that you can only use `f(Variable)` as a command to the Erlang shell; you can't use it in other Erlang code.

(f) `[ { value, Purple } || Purple <- [ 5, george, "22" ] ].`

Ans. This is a list comprehension. It builds the list of elements of the form `{value, Purple}` setting `Purple` to each element of the list `[5, george, "22"]`. The value of the expression is:
```
[{value, 5}, {value, geoge}, {value, "22"}].
```
Note that `value` is an atom – it's a kind of constant, and it's the same value in each tuple in the result list.

2. **Erlang Types.** What is the type of each of the following Erlang expressions (variable, atom, boolean, integer, float, list or tuple)?

(a) `jeopardy`

Ans. This is an Erlang atom. An atom is any token that starts with a **lower-case** letter and continues with letters (upper or lower case), digits, and underscores. You can also make an atom by putting any string of characters between a pair of single quotes. For example,
```
[is_atom(A) || A <- [x, x_or_15_Times_y, 'x or 15*y', '']].
```
has the value
```
[true, true, true, true]
```

(b) `True`

Ans. This is an Erlang variable. Erlang variable names start with a capital letter of `_` and continue with letters (upper or lower case), digits, and underscores.

(c) `"3.14159"`

Ans. This is an Erlang list. It's also a string because Erlang represents strings as lists of integers where each integer is a valid ASCII character. When deciding whether to print a list of integers as a list or a string, the Erlang shell considers the following characters to be valid, printable, ASCII values:

- Any value N with $8 \leq N \leq 13$ or $N = 27$. These are various cursor control characters (tab, newline, etc.).
- Any value N with $32 \leq N \leq 126$ – these are the "usual" printable characters.
- Any value N with $160 \leq N \leq 255$ – various special symbols and accented characters from various European languages.

(d) `6.02214e23`
Ans. This is an Erlang floating point constant. It satisfies the type recognizers is_float and is_number.

(e) `[ { ok, 42 } ]`
Ans. This is an Erlang tuple. It has two elements. The first element is the atom `ok` and the second is the number number `42`. `42` satisfies the type recognizers is_integer and is_number.

(f) `false` Ans. This is an Erlang boolean. The boolean constants `true` and `false` are really special atoms. They satisfy the type recognizer is_atom. As noted in *Learn You Some Erlang*, the operations such as `and`, `or`, `if`, ..., all work with `true` and `false` just the way they should; so, you can think of `true` and `false` as being the two, boolean constants and not worry about the technical detail that they also happen to be atoms. Of course if you write

f(A) when is_atom(A) -> atom;
f(_) -> molecule. Then note that `f(1 < 2)` evaluates to `atom`.

(g) `_haberdashery` Ans. This is an Erlang variable because it starts with an underscore. Starting a variable with an underscore suppresses compiler warnings about the value not being used. A couple of remarks.

- If you have a variable such as `_haberdashery` as a parameter to a function or bind it with `_haberdashery = ...`, you can use the value of that is bound to `_haberdashery`. DON'T. Even though the compiler accepts it, starting a variable name with `_` is telling the reader that you are ignoring the variable. If you say you're ignoring the variable, then you should ignore it. We will feel free to take off "style points" on solutions that use the value of variables whose names begin with `_`.
- Stub functions in the templates that we provide for homework solutions usually have parameters with names that begin with `_`. This is so the template file will compile without warnings (that's good). When you write your own code and use the values of those parameters, you should remove the `_` from the name. For example, if a stub has a parameter called `_ListOfExoPlanets`, your code should rename this to `ListOfExoPlanets` in any pattern where the value is used.
- The name `_` is syntactically a variable, but it is never bound to a value. For example, if you write:

  f1(_,_) -> 42.
  f2(_A, _A) -> 42.

  then `f1(1,1)`, `f1(1,2)`, and `f2(1,1)` will all evaluate to `42`. However, `f2(1,2).` fails with the error "no function clause matching ... `f2(1,2)`". That's because `_A` cannot be bound to both `1` and `2`. Because `_` is never bound to a value, it can "match" both `1` and `2`.

(h) `{ [ 1 | [ 2 | [ 3 ]]] }` % 1 element tuple (containing a 3 element list). Ans. This is an Erlang tuple of one element. The element is the list of three integers, `[1, 2, 3]`.

3. **Write You Some Erlang.** Write a function `sublist(List, Start, End)` where `List` is a list, `Start` is a positive integer and `End` is an integer greater than or equal to start. The function returns the sublist of `List` containing elements `Start` (inclusive) through `End` (exclusive). Remember that Erlang likes to start numbering elements from 1.

   Ans. Here's the solution that we worked out in class:

```
-module(sublist).
-export([sublist/3, testlist/0]).

sublist([], _Start, _End) -> [];
sublist(_List, 1, 1) -> [];
sublist([ H | T ], 1, End) -> [ H | sublist(T, 1, End-1) ];
sublist([ _H | T ], Start, End) -> sublist(T, Start-1, End-1).
```

   It's a good, in-class solution. However, we should add some guards. For example, with the solution above:

   `sublist([1,2,3], 2, 1000).`

   returns `[2,3]` – it should probably fail with some kind of error. As another example

   `sublist([1,2,3,4,5], 2, 0).`

   returns `[2,3,4,5]`; again, a failure would be better. The same proble occurs if `End` is a non-integer, floating point number. Let's add some guards. We'll start with:

   `sublist([], _Start, _End) -> [];`

   What should the guard be? Well, we have only "finished" the job if `_Start` and `_End` are both `1`. We could write:

   `sublist([], _Start, _End) when _Start =:= 1, _End =:= 1 -> [];`

   but I'll take of points for using the value of a variable whose name starts with `_`. Removing the underscores gives us:

   `sublist([], Start, End) when Start =:= 1, End =:= 1 -> [];`

   That's better. Now we see this is just a special case of the next pattern:

   `sublist(_List, 1, 1) -> [];`

   So, we decide to delete the first pattern entirely!

   Now, let's look at the second pattern:

   `sublist(_List, 1, 1) -> [];`

   Does it need a guard? Well, consider, `sublist(monkey, 1, 1)`. This matches the pattern and produces the value `[]`. We probably intend that the first argument should be a list. The pattern with the guard is

   `sublist(List, 1, 1) when is_list(List) -> [];`

   Do we need guards for `1` and `1`. Nope – any call to `sublist` that matches this pattern must have a value of the integer `1` for the second and third arguments; there's nothing more we need to check.

   The next pattern is

   `sublist([ H | T ], 1, End) -> [ H | sublist(T, 1, End-1) ];`

   The pattern enforces that the first argument is a list (good), and that the second argument is a positive integer (good). We're not checking that `End` is an integer, or that `End >= Start`.

We could be lazy and just let the code recurse until something goes wrong and generates an error. OTOH, failing early usually gives us better error messages and makes our code easier to debug and maintain. The only downside is that the extra guard checking adds a bit of execution overhead. Here, the best practice is to write code that is easy to debug and maintain. If **and only if** it turns out that the code is performance critical, then we can reevaluate that choice. So, we'll add a guards for `End`.

```
sublist([ H | T ], 1, End) when is_integer(End), 1 < End -> [ H | sublist(T, 1, End-1) ];
```

One remark about syntax. If we have a list of guards separated by commas, that means that all of them must hold. The comma acts like a `andalso` – if one of the earlier guard clauses evaluates to false, then the others are note evaluated. We can also use semicolons that are kind of like `orelse`. See the description of guard seqeunces in the Erlang reference manual if you want the gory details. For the most part, in this class, we'll find it convenient to write a list of guards separated by commas to indicate that they all must hold. If you want something more complicated, just use `andalso` and `orelse` and you can make it clear what you're trying to say.

The final pattern is

```
sublist(sublist([ _H | T ], Start, End) -> sublist(T, Start-1, End-1).
```

Here, we'll just add our guards that `Start` and `End` must be positive integers with `Start =< End`:

```
sublist(sublist([ _H | T ], Start, End)
      when is_integerStart, is_integer(End), 1 =< Start, Start =< End ->
   sublist(T, Start-1, End-1).
```

Putting it all together, we get

```
sublist(List, 1, 1) when is_list(List) -> [];
sublist([ H | T ], 1, End) when is_integer(End), 1 < End -> [H | sublist(T, 1, End-1)];
sublist([ _H | T ], Start, End)
      when is_integer(Start), is_integer(End), 1 =< Start, Start =< End ->
   sublist(T, Start-1, End-1).
```

That's it!

4. **Referential Transparency: Not Just for Functional Languages!** Write a function `dotProd(A, B)` in your favorite *imperative* language (eg: Java, Python, C, C++, etc.) which returns the dot product of the one-dimensional vectors of numbers `A` and `B`. You may assume that `A` and `B` are stored in arrays, lists, or whatever data type is convenient for your language choice. However, *your code must display referential transparency*; in other words, you may not change the value of a variable once it is set.

   Not an answer: We didn't cover this one in class. I'll leave it unanswered. However, if there's a discussion on piazza, or you bring the question to office hours or tutorial, we'll be happy to discuss it.