Erlang notes

Installation and references

Download prebuilt packages for the virtual machine and libraries (OTP)
https://www.erlang-solutions.com/resources/download.html

Book: Learn You Some Erlang For Great Good, by Fred Hebert
http://learnyousomeerlang.com/

Language overview

Library reference (e.g. you can look up the math or io module, etc.)
http://erlang.org/doc/apps/stdlib/index.html

Joe Armstrong dissertation (see Ch 2 for philosophy and objectives of Erlang)

Interesting comments on efficiency of list operations
http://www.erlang.org/doc/efficiency_guide/users_guide.html

IDEs
- I suggest *avoiding* the Eclipse plugin, there seem to be serious design flaws and poor support.
- There is a plugin for IntelliJ IDEA that seems to work well, and you can use it with the free version of IntelliJ IDEA.
- I have been using the emacs profile for Erlang, supposedly there is one for vim also that I have not tried.

The stuff I covered in the introduction to Erlang in class, is covered in the following chapters of the book:

- Introduction
- Starting Out For Real (ignore the bit syntax and list comprehensions for now)
- Modules
- Syntax in Functions
  [you can ignore the Types section for now]
- Recursion - see the first few sections at least

Code examples discussed here

http://web.cs.iastate.edu/~smkautz/cs430s17/examples/erlang/
Using the REPL

(type erl at command prompt to start)
q(). quit
Ctrl-g job control, h for help
b(). display variable bindings
f(X). unbind variable X in the REPL ("forget")
f(). unbind all variables
Ctrl-a, Ctrl-e cursor to beginning/end of line
up/down arrows history
cd(path) change working directory (path is string in quotes)
c(modulename) compile (filename should be modulename + ".erl")
Note: An expression to evaluate must be followed by a period

Arithmetic

Arithmetic as you’d expect, but use div and rem for integer division and remainder
Integers are infinite precision
Relational operators >, >=, <, <= (note that wacky one)
=:, /= exact equality/inequality of value and type
==, /= equality/inequality of value (useful for comparing ints with floats)
Boolean operators not, and, or, xor, these do not short-circuit
(For short-circuiting use andalso, orelse)

Variables and atoms

Variables start with upper case letter, can be bound to value only once
Atoms start with lower case letter, or can be any text in single quotes
(An atom is just a symbolic literal, not associated with a value.)
true and false are predefined atoms
Use = for assignment (however this is really a special case of pattern matching)

Tuples

A sequence of values in curly braces, fixed in size, e.g. P = {2, 3}.
Can assign a tuple, using pattern matching, e.g.
{A, B} = P. (now A is 2 and B is 3)
{_, C} = P. (now C is 3, underscore is like a "don’t care")
Built in functions:
element(index, tuple) returns element at index, which is 1-based
setelement(index, tuple, value) returns new tuple with value at index
A tuple {atom, value} is called a "tagged tuple", can be used as a dictionary-like data
type, e.g. {{width, 10}, {height, 20}}
Lists

Immutable sequence of values in square brackets, a nonempty list is composed of a value ("head") followed by a list ("tail"), which may be the empty list []
E.g. [1, 2, {foo, bar}, banana] is a list
Can append a list M onto another list L using L ++ M
Note that L ++ M causes L to be copied
Can prepend value A at head of list L using [A | L] (the "cons" operator)
[A | L] does not require any copying, L is still a valid list
There is also an operator -- that removes elements, e.g.
   > L = [2, 4, 6, 4, 8].
   [2,4,6,4,8]
   > L -- [8, 4].
   [2,6,4]

There are built-in functions (BIFs) length(L), hd(L), tl(L) for length, head, tail
Or, pattern match: [H|T] = L to get head H and tail T

Strings

Are just lists of integers, if all integers are in range 0-255, will display as text. This is sometimes confusing.

11> S = "hello".
"hello" 12> S.
"hello"
13> hd(S).
104
14> tl(S).
"ello"

Functions and modules

A module is a file containing function definitions
Generally to call a function we have to specify modulename:functionname(args),
except for the module erlang, which is automatically imported
E.g. lists:seq(1, 10)

A module contains metadata, which must start with -module
Will usually have an -export as well, e.g.,
-module (modulename) .
-export (list of functions to export, in brackets)

Function in the export list is identified by name and number of arguments, e.g.
helloworld/0 or area/1
Filename must be modulename + ".erl"
**Function syntax**

A function "clause" is of the form `name(args) -> body`

One function may have multiple clauses, separated by semicolons, and the last clause ends with a period.

Function body may contain multiple expressions, separated by commas. Last expression is function value.

```erlang
helloworld() ->
  io:format("Hello, World\n"),
  io:format("Erlang is wacky!\n").
```

(The `\n` is a format specifier that works like `'\n'`, can also use format strings followed by a list of values, e.g.

```erlang
>io:format("~5.2f\n", [3.14159]).
3.14
```

The most common specifier is `~p`, for printing any Erlang value in the "usual" way)

Note that `io:format` returns the atom 'ok'.

When a function has multiple clauses, pattern matching is performed on the arguments, in the order the clauses are listed. Note use of a "tagged tuple" as the function argument.

```erlang
area({rect, Width, Height}) ->
  Width * Height;
area({circle, Radius}) ->
  math:pi() * Radius * Radius;
area(_) ->
  0.
```

**Case expressions, guards, and if-expressions**

The pattern matching for the `area` function could have been done with a case expression:

```erlang
area2(Shape) ->
  case Shape of
    {rect, Width, Height} ->
      Width * Height;
    {circle, Radius} ->
      math:pi() * Radius * Radius;
    _ -> 0
  end.
```

Pattern matching can be combined with a "guard" or conditional that must be true for the match to proceed. Multiple guards can be combined with a comma ("andalso") or semicolon ("orelse"):
grade(N) when N >= 90 ->
   "A";
grade(N) when N >= 80 ->
   "B";
grade(_) ->
   "F".

An if-expression can be used when you want guards, but not the pattern matching. This works for the grade/1 function since the pattern matching is trivial:

```prolog
grade(N) ->
   if N >= 90 ->
      "A";
   N >= 80 ->
      "B";
   true -> "F"
end.
```

Notice there is no else keyword.

**No loops**

Repetition happens via recursion. Tail recursion is optimized, so "infinite" recursion is possible.

```prolog
%% Factorial, not tail recursive
fac(1) -> 1;
fac(N) -> N * fac(N - 1).
```

```prolog
%% Uses tail recursive helper function tailFac/2
tailFac(N) ->
   tailFac(N, 1).
tailFac(1, Acc) ->
   Acc;
tailFac(N, Acc) ->
   tailFac(N - 1, N * Acc).
```

**List comprehensions**

Use || to mean "such that":

```prolog
> L = [1, 2, 3, 4, 5].
L = [1, 2, 3, 4, 5]
> [X * X || X <- L].
L = [1, 2, 3, 4, 5]
```

With a guard:

```prolog
> [X || X <- L, X rem 2 == 0].
L = [1, 2, 3, 4, 5]
```
Can also use a pattern match in the "generator", e.g., \( \{X, \text{fog}\} \leftarrow \text{Weather} \)

Other list operations in lists module: `map`, `filter`, `foldl`, `foldr`, `all`, `any`, 
`foreach`, `takewhile`, `dropwhile`

Can also define anonymous functions, \( \text{fun}(\text{args}) \Rightarrow \text{result} \) \text{end.}