Introduction to Erlang

Mark Greenstreet

CpSc 418 – January 6, 2016

Outline:

Objectives

- Learn/review key concepts of functional programming:
  - Referential transparency.
  - Structuring code with functions.
- Introduction to Erlang
  - Basic data types and operations.
  - Program design by structural decomposition.
  - Writing and compiling an Erlang module.

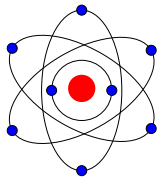
Erlang Expressions

- The basic pieces of Erlang expressions: numerical constants, atoms, lists, and tuples.
- Operations: arithmetic, comparison, boolean.
- Variables and matching.

Numbers

- Integers
  - just write the decimal representation: `0`, `1`, `2`, `-17`, `42`, and so on.
  - for other bases, write **B#V**, where **B** is the base (written in decimal); $1 \leq \mathbf{B} \leq 36$; and **V** is the value. For example,
    - ★ `8#42` is the same as `34`.
    - ★ `16#2a` is the same as `42`.
    - ★ `36#4p` is the same as `169` – but you'd better have a good excuse for using base 36!
  - Integers can be arbitrarily large.
- Floating point constants – just like C and Java (well, almost). Examples:
  - `3.0`, `0.5`, `2.99792458e8`, `-1.602e-19`.
  - Unlike C and Java, Erlang requires that at least one digit on each side of the decimal point. Thus, `2.` and `.5` are not valid constants in Erlang.
  - Erlang floats are (implementation dependent but you can assume that they are) IEEE double precision.
- See also: the *LYSE* explanation.

Atoms



- Erlang has a primitive type called an atom.
  - An atom is any non-empty sequence of
    - ★ letters, `a`...`z` and `A`...`Z`,
    - ★ digits, `0`...`9`, and
    - ★ underscores, _,
    - ★ where the first character is a lower-case letter, `a`...`z`.
  - Or, any sequence of characters enclosed by single quotes, `'`.
  - Examples: `atom`, `r2D2`, `'3r14|\|6 r00lz'`.
- Each atom is distinct.
  - Handy for "keys" for pattern matching and flags to functions.
  - Erlang uses several standard atoms including: `true`, `false`, `ok`.
  - Module and function names are atoms.
- See also: the *LYSE* explanation.

Arithmetic

- `+`, `-`, and `*` do what you'd expect.
- `/` is division and always produces a floating point result.
- `div` is integer division. The operands of `div` must be integers, and the result is an integer.
- `rem` is integer remainder.
- Examples:

```
1> 12 + 3.
15
2> 12 + 3.0.
15.0
3> 12 / 3.
4.0
4> 12 div 3.
4
5> 12 div 3.0.
** exception error:  an error occurred when
     evaluating an arithmetic expression...
```

- See also, the *Learn You Some Erlang* description of numbers.

Comparisons

- `<`, `>`, and `>=` do what you'd expect.
- "less-than-or-equal" is written `=<` (just to be weird).
- There are two ways to say "equal"
  - `=:=` strict comparison: `5 =:= 5.0` evaluates to `false`.
  - `==` numerical comparison: `5 == 5.0` evaluates to `true`.
  - For non-numerical values, the two are equivalent.
- Likewise,
  - `=/=` is the strict not-equals, `5 =/= 5.0` evaluates to `true`;
  - `/=` is the numerical version, `5 /= 5.0` evaluates to `false`.
- See also: the *LYSE* explanation.

Boolean operations

- The atoms `true` and `false` are the two boolean constants.
  - Erlang does not treat non-zero values as true or anything like that. For example,

```
6> not true.
false
7> not 0.
** exception error:  bad argument ...
8>
```

- `not` is boolean negation (see the example above).
- `and` is conjunction. Note that `expr1 and expr2` evaluates **both** `expr1` **and** `expr2`, even if `expr1` evaluates to false. If you want or need short-circuit evaluation, use the operator `andalso`.
- `or` is conjunction; it evaluates both of its operands. `orelse` is the short-circuit version (it's not a threat).
- `xor` is exclusive-or.
- See also: the *LYSE* explanation.

Lists

- We described lists in the Jan. 4 lecture. They are so central to Erlang, I'll say more here.
- How to make lists:
  - Just write the elements inside square brackets: `[A, B+C, 23]`.
    - This is like the `list` function in Racket (or Scheme, or Lisp).
    - Lists can be empty: `[]`.
    - Lists can be nested (to make trees):
      `[X, [[Y, Z], 2, [A, B+C, [], 23]], 14, [[[8]]]]`.
  - Prepend a new head to an existing list: `[NewElement | ExistingList]`.
    - This is like the `cons` function in Racket
    - Example: `[A | [1, 2, 3]` is equivalent to `[A, 1, 2, 3]`.
  - Concatenate two lists: `List1 ++ List2`.
    - This is like the `append` function in Racket
    - Example: `[1, 2, 3] ++ [A, B, C]` is equivalent to `[1, 2, 3, A, B, C]`.

More Lists

- How to take a list apart.
  - `hd(L)` is the head of list `L`.
    - ★ `hd([1, 2, 3])` evaluates to `1`.
    - ★ `hd([[1, 2], 3])` evaluates to `[1, 2]`.
    - ★ `hd([])` and `hd(dog)` throw bad argument exceptions.
  - `tl(L)` is the tail of list `L`.
    - ★ `tl([1, 2, 3])` evaluates to `[2, 3]`.
    - ★ `tl([[1, 2], 3])` evaluates to `[3]`.
    - ★ `tl([])` and `hd(dog)` throw bad argument exceptions.
- Deleting elements from a list.
  - `L1 -- L2` deletes the first occurrence of each element of `L2` from `L1`.
  - `[1,2,3,4,5] -- [2,4,6]` evaluates to [1,3,5].
  - `[1,4,6,4,1] -- [2,4,6]` evaluates to [1,4,1].

More^2 Lists

- `length(L)` returns the number of elements in list `L`.
  - `length([1, 2, 3])` evaluates to `3`.
  - `length([[1, 2], 3])` evaluates to `2`.
  - `length([])` evaluates to `0`.
- Patterns are a great way to take lists apart:
  - `[Head | Tail] = L` binds `hd(L)` to `Head` and `tl(L)` to `Tail` (assuming `Head` and `Tail` were previously unbound).
  - Frequently, patterns are a **much** clearer way to access parts of a list than writing a bunch of calls to `hd` and `tl`.
  - We'll describe patterns in more detail in the Jan. 8 lecture.
- There are many more functions for lists in the lists module that is part of the standard Erlang API.
- List comprehensions are also quite handy. We'll cover them in the Jan. 8 lecture.
- See also: the *LYSE* explanation.

Tuples

- Tuples are the other main data-structure in Erlang.
- Some simple examples:

```
8> T1 = {cat, dog, potoroo}.
{cat,dog,potoroo}
9> L6 = [ {cat, 17}, {dog, 42}, {potoroo, 8}].
[{cat,17}, {dog,42}, {potoroo,8}]
10> element(2, T1).
dog
11> T2 = setelement(2, T1, banana).
{cat,banana,potoroo}
12> T1.
{cat,dog,potoroo}
13>
```

- Observe that `setelement` created a **new tuple** that matches `T1` in all positions except for position 2, which now has the value `'banana'`. The original tuple, `T1`, is unchanged.
- See also: the *LYSE* explanation.

Lists vs. tuples

Why have both lists and tuples?

- Tuples are typically used for a small number of values of heterogeneous “types”. The position in the tuple is significant.
- Lists are typically used for an arbitrary number of values of the same “type”. The position in the list is usually not-so-important (but we may have sorted lists, etc.).

Strings

What happened to strings?!

- Well, they're lists of integers.
- This can be annoying. For example,

```
13> [102, 111, 111, 32, 98, 97, 114].
"foo bar"
14>
```

- By default, Erlang prints lists of integers as strings if every integer in the list is the ASCII code for a "printable" character.
- *Learn You Some Erlang* discusses strings in the "Don't drink too much Kool-Aid" box for lists.

Variables

- An Erlang variable is any non-empty sequence of
  - letters, `a`...`z` and `A`...`Z`,
  - digits, `0`...`9`, and
  - underscores, _,
  - where the first character is an **upper-case** letter, `A`...`Z`.
- A value is bound to a variable by "matching" for example:

```
14> X = [0, 1, 4, 9, 16, 25].
[0, 1, 4, 9, 16, 25]
15> hd(X).
0
```

- If a variable is already bound to a value, then it's an error to try to bind a **different** value to it.

```
16> X = 42.
** exception error:  no match of right hand side value 42
17> X = [0, 1, 4, 9, 16, 25].
[0, 1, 4, 9, 16, 25]
```

- This has brought us to the next segment of this lecture: Erlang is functional.
- See also: the *LYSE* explanation.

Functional Programming

- Imperative programming (C, Java, Python, . . .) is a programming model that corresponds to the von Neumann computer:
  - A program is a sequence of statements.
    In other words, a program is a recipe that gives a step-by-step description of what to do to produce the desired result.
  - Typically, the operations of imperative languages correspond to common machine instructions.
  - Control-flow (`if`, `for`, `while`, function calls, etc.)
    Each control-flow construct can be implemented using branch, jump, and call instructions.
  - This correspondence program operations and machine instructions simplifies implementing a good compiler.
- Functional programming (Erlang, lisp, scheme, Haskell, ML, . . .) is a programming model that corresponds to mathematical definitions.
  - A program is a collection of definitions.
  - These include definitions of expressions.
  - Expressions can be evaluated to produce results.
- See also: the *LYSE* explanation.

Erlang Makes Parallel Programming Easier

- Erlang is functional
  - Each variable gets its value when it's declared – in **never** changes.
  - Erlang eliminates many kinds of races – another process **can't** change the value of a variable while you're using it, because the values of variables never change.
- Erlang uses message passing
  - Interactions between processes are under explicit control of the programmer.
  - Fewer races, synchronization errors, etc.
- Erlang has simple mechanisms for process creation and communication
  - The structure of the program is not buried in a large number of calls to a complicated API.

Big picture: Erlang makes the issues of parallelism in parallel programs more apparent and makes it easier to avoid many common pitfalls in parallel programming.

Referential Transparency

- This notion that a variable gets a value when it is declared and that the value of the variable never changes is called **referential transparency**.
  - You'll here me use the term many times in class – I thought it would be a good idea to let you know what it means. ☺
- We say that the value of the variable is **bound** to the variable.
- Variables in functional programming are much like those in mathematical formulas:
  - If a variable appears multiple places in a mathematical formula, we assume that it has the same value everywhere.
  - This is the same in a functional program.
  - This is not the case in an imperative program. We can declare `x` on line 17; assign it a value on line 20; and assign it another value on line 42.
  - The value of `x` when executing line 21 is different than when executing line 43.

Loops violate referential transparency

```
// vector dot-product
sum = 0.0;
for(i = 0; i < a.length; i++)
   sum = a[i] * b[i];
```

```
// merge, as in merge-sort
while(a != null && b != null) {
   if(a.key <= b.key) {
      last->next = a;
      last = a;
      a = a->next;
      last->next = null;
   } else {
      ...
   }
}
```

- Loops rely on changing the values of variables.
- Functional programs use recursion instead.
- See also the *LYSE* explanation.

Life without loops

Use recursive functions instead of loops.

```
dotProd([], []) -> 0;
dotProd([A | Atl], [B | Btl]) -> A*B + dotProd(Atl, Btl).
```

- Functional programs use recursion instead of iteration:

  ```
  dotProd([], []) -> 0;
  dotProd([A | Atl], [B | Btl]) -> A*B + dotProd(Atl, Btl).
  ```

- Anything you can do with iteration can be done with recursion.
  - But the converse is not true (without dynamically allocating data structures).
  - Example: tree traversal.

Example: Sorting a List

- The simple cases:
  - Sorting an empty list: `sort([]) ->` _____
  - Sorting a singleton list: `sort([A]) ->` _____
- How about a list with more than two elements?
  - Merge sort?
  - Quick sort?
  - Bubble sort (NO WAY! Bubble sort is DISGUSTING!!!).
- Let's figure it out.

Merge sort: Erlang code

- If a list has more than one element:
  - Divide the elements of the list into two lists of roughly equal length.
  - Sort each of the lists.
  - Merge the sorted list.
- In Erlang:

```
sort([]) -> [];
sort([A]) -> [A];
sort([A | Tail]) ->
   {L1, L2} = split([A | Tail]),
   L1_sorted = sort(L1), L2_sorted = sort(L2), merge(L1_sort
```

- Now, we just need to write `split`, and `merge`.

split(L)

Identify the cases and their return values according to the shape of L:

```
% If L is empty (recall that split returns a tuple of two lists):
split([]) -> {                    ,                    }

% If L

split(                ) ->

% If L



% If L
```

`merge(L1, L2)`

- Precondition: We assume `L1` and `L2` are each in non-decreasing order.
- Return value: a list that consists of the elements of `L1` and `L2` and the elements of the return-list are in non-decreasing order.
- Identify the cases and their return values.
  - What if `L1` is empty?
  - What if `L2` is empty?
  - What if both are empty?
  - What if neither are empty?
  - Are there other cases?
    Do any of these cases need to be broken down further?
    Are any of these case redundant?

`merge(L1, L2)`

Let's write the code:

Modules

- To compile our code, we need to put it into a module.
- A module is a file (with the extension `.erl`) that contains
  - Attributes: declarations of the module itself and the functions it exports.
    - The module declaration is a line of the form:

      ```
      -module(moduleName).
      ```

      where `moduleName` is the name of the module.
    - Function exports are written as:

      ```
      -export([functionName1/arity1,
      functionName2/arity2, ...]).
      ```

      The list of functions may span multiple lines and there may be more than one `-export` attribute.

      `arity` is the number of arguments that the function has. For example, if we define

      ```
      foo(A, B) -> A*A + B.
      ```

      Then we could export `foo` with

      ```
      -export([..., foo/2, ...]).
      ```
    - There are many other attributes that a module can have. We'll skip the details. If you really want to know, it's all described here.
  - Function declarations (and other stuff) – see the next slide

A module for `sort`

```
-module(sort).
-export([sort/1]).
% The next -export is for debugging. We'll comment it out later
-export([split/1, merge/2]).
sort([]) -> [];
...
```

Let's try it!

```
18> c(sort).
{ok,sort}
19> R20 = count3s:rlist(20, 100).   % test case: a random list
[45,73,95,51,32,60,92,67,48,60,15,21,70,16,56,22,46,43,1,57]
20> S20 = sort:sort(R20).   % sort it
[1,15,16,21,22,32,43,45,46,48,51,56,57,60,60,67,70,73,92,95]
21> R20 -- S20.   % empty if S20 is a permutation of R20
[]
22> S20 -- R20.   % empty if S20 is a permutation of R20
[]
```

Yay – it works!!! (for one test case)

Summary

- Why Erlang?
  - Functional – avoid complications of side-effects when dealing with concurrency.
  - But, we can't use imperative control flow constructions (e.g. loops).
    - Design by declaration: look at the structure of the data.
    - More techniques coming in upcoming lectures.
- Sequential Erlang
  - Lists, tuple, atoms, expressions
  - Using structural design to write functions: example sorting.

Preview

January 8: More FUN with Erlang FUNctions

Reading:	*Learn You Some Erlang*, the next four sections – Syntax in Functions through Higher Order Functions
Mini-Assignment:	Mini-Assignment 1 due **10:00am**
Homework:	**Homework 1 goes out** – simple programming with Erlang

January 11: Processes and Messages

Reading:	*Learn You Some Erlang*, The Hitchhiker's Guide . . . through More on Multprocessing
Mini-Assignment:	Mini-Assignment 2 due **10:00am**

January 13: Reduce and Scan (simple)

Reading:	Lin & Snyder, chapter 5, pp. 112–125

January 15: Reduce and Scan (generalize)

Homework:	Homework 1 deadline for early-bird bonus (11:59pm)
	Homework 2 goes out – parallel programming with Erlang

January 18: Architecture Review

Reading:	Pacheco, Chapter 2, Sections 2.1 and 2.1.
Homework:	Homework 1 due **11:59pm**

January 20: Shared-Memory Machines

Reading:	Pacheco, Chapter 2, Section 2.2

Review Questions

- What is the difference between `==` and `=:=`?
- What is an atom?
- Draw the tree corresponding to the nested list

  `[X, [[Y, Z], 2, [A, B+C, [], 23]], 14, [[[8]]]].`
- What is referential transparency?
- Why don't functional languages have loops?

Supplementary Material

The remaining slides are some handy material that we won't cover in lecture, but you can refer to if you find it helpful.

- Common mistakes with lists and how to avoid them.
- Suppressing verbose output when using the Erlang shell.
- Forgetting variable bindings (only in the Erlang shell).
- Table of Contents.

Remarks about Constructing Lists

It's easy to confuse `[A, B]` and `[A | B]`.

- This often shows up as code ends up with crazy, nested lists; or code that crashes; or code that crashes due to crazy, nested lists; ....
- Example: let's say I want to write a function `divisible_drop(N, L)` that removes all elements from list `L` that are divisible by `N`:

```
divisible_drop(_N, []) -> []; % the usual base case
divisible_drop(N, [A | Tail]) ->
   if A rem N == 0 -> divisible_filter(N, Tail);
      A rem N /= 0 -> [A | divisible_filter(N, Tail)]
   end.
```

  It works. For example, I included the code above in a module called `examples`.

```
1> examples:divisible_drop(3, [0, 1, 4, 17, 42, 100]).
[1,4,17,100]
```

Miscontructing Lists

Working with `divisible_drop` from the previous slide...

- Now, change the second alternative in the `if` to
  `A rem N /= 0 -> [A, divisible_filter(N, Tail)]`
  Trying the previous test case:
  ```
  2> examples:divisible_drop(3, [0, 1, 4, 17, 42, 100]).
  [1,[4,[17,[100,[]]]]]
  ```
  Moral: If you see a list that is nesting way too much, check to see if you wrote a comma where you should have used a `|`.
- Restore the code and then change the second alternative for `divisible_drop` to `divisible_drop(N, [A, Tail]) ->` Trying our previous test:
  ```
  3> examples:divisible_drop(3, [0, 1, 4, 17, 42, 100]).
  ** exception error:  no function clause matching...
  ```

Avoiding Verbose Output

- Sometimes, when using Erlang interactively, we want to declare a variable where Erlang would spew enormous amounts of "uninteresting" output were it to print the variable's value.
  - We can use a comma (i.e. a block expression) to suppress such verbose output.
  - Example

```
4> L1_to_5 = lists:seq(1, 5).
[1, 2, 3, 4, 5].
5> L1_to_5M = lists:seq(1, 5000000), ok.
ok
6> length(L1_to_5M).
5000000
7>
```

Forgetting Bindings

- Referential transparency means that bindings are forever.
  - This can be nuisance when using the Erlang shell.
  - Sometimes we assign a value to a variable for debugging purposes.
  - We'd like to overwrite that value later so we don't have to keep coming up with more name.s
- In the Erlang shell, `f(Variable).` makes the shell "forget" the binding for the variable.

```
7> X = 2+3.
5.
8> X = 2*3.
** exception error:  no match of right hand side valu
9> f(X).
ok
10> X = 2*3.
6
11>
```

Table of Contents