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 Basics

- Numbers:
  - Numerical Constants: `1`, `8#31`, `1.5`, `1.5e3`,
    but not: `1.` or `.5`
  - Arithmetic: `+`, `-`, `*`, `/`, `div`, `band`, `bor`, `bnot`, `bsl`, `bsr`, `bxor`
- Booleans:
  - Comparisons: `=:=`, `=/=`, `==`, `/=`, `<`, `=<`, `>`, `>=`
  - Boolean operations (strict): `and`, `or`, `not`, `xor`
  - Boolean operations (short-circuit): `andalso`, `orelse`
- Atoms:
  - Constants: `x`, `'big DOG-2'`
  - Operations: tests for equality and inequality. Therefore pattern matching.

Lists and Tuples

- Lists:
  - Construction: `[1, 2, 3]`, `[Element1, Element2, ..., Element_N | Tail]`
  - Operations: `hd`, `tl`, `length`, `++`, `--`
  - Erlang's list library, http://erlang.org/doc/man/lists.html: `all`, `any`, `filter`, `foldl`, `foldr`, `map`, `nth`, `nthtail`, `seq`, `sort`, `split`, `zipwith`, and **many** more.
- tuples:
  - Construction: `{1, dog, "called Rover"}`
  - Operations: `element`, `setelement`, `tuple_size`.
  - Lists vs. Tuples:
    - **Lists** are typically used for an **arbitrary** number of elements of the same "type" – like arrays in C, Java, ...
    - **Tuples** are typically used for an **fixed** number of elements of the varying "types" – likes a `struct` in C or an object in Java.

Strings

What happened to strings?!

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

```
1> [102, 111, 111, 32, 98, 97, 114].
"foo bar"
2>
```

- 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.

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 between 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 – it **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_sorted, L2_sorted).
```

- 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
```

Finishing merge sort

- An exercise for the reader – see slide 29.
- Sketch:
  - Write `merge(List1, List2) -> List12` – see slide 30
  - Write an Erlang module with the `sort`, `split`, and `merge` functions – see slide 31
  - Run the code – see slide 33

Fun with functions

- Programming with patterns
  - often, the code just matches the shape of the data
  - like CPSC 110, but pattern matching makes it obvious
  - see slide 16
- Fun expressions
  - in-line function definitions
  - see slide 17
- Higher-order functions
  - encode common control-flow patterns
  - see slide 18
- List comprehensions
  - common operations on lists
  - see slide 19
- Tail call elimination
  - makes recursion as fast as iteration (in simple cases)
  - see slide 20

Programming with Patterns

```
% leafCount: count the number of leaves of a tree represented by a nested list
leafCount([]) -> 0; % base case – an empty list/tree has no leaves

leafCount([Head | Tail]) -> % recursive case
   leafCount(Head) + leafCount(Tail);
leafCount(_Leaf) -> 1; % the other base case – _Leaf is not a list
```

- Let's try it

  ```
  2> examples:leafCount([1, 2, [3, 4, []], [5, [6, banana]]]).
  7
  ```

- Notice how we used **patterns** to show how the recursive structure of `leafCount` follows the shape of the tree.
- See Pattern Matching in *Learn You Some Erlang* for more explanation and examples.
- Style guideline: if you're writing code with lots of `if`'s `hd`'s, and `tl`'s, you should think about it and see if using patterns will make your code simpler and clearer.

Anonymous Functions

```
3> fun(X, Y) -> X*X + Y*Y end.  % fun ... end creates an "anonymous function"
#Fun<erl_eval.12.52032458> % ok, I guess, but what can I do with it?!
4> F = fun(X, Y) -> X*X + Y*Y end.
#Fun<erl_eval.12.52032458>
5> F(3, 4).
25
6> Factorial = % We can even write recursive fun expressions!
     fun Fact(0) -> 1;
         Fact(N) when is_integer(N), N > 0 -> N*Fact(N-1)
     end.
7> Factorial(3).
6
8> Fact(3).
* 1: variable 'Fact' is unbound
9> Factorial(-2).
** exception error: no function clause matching
      erl_eval:'-inside-an-interpreted-fun-'(-2)
10> Factorial(frog).
** exception error: no function clause matching
      erl_eval:'-inside-an-interpreted-fun-'(frog)
```

See Anonymous Functions in *Learn You Some Erlang* for more explanation and examples.

Higher-Order Functions

- `lists:map(Fun, List)` apply `Fun` to each element of `List` and return the resulting list.

```
11> lists:map(fun(X) -> 2*X+1 end, [1, 2, 3]).
[3, 5, 7]
```

- `lists:fold(Fun, Acc0, List)` use `Fun` to combine all of the elements of `List` in left-to-right order, starting with `Acc0`.

```
12> lists:foldl(fun(X, Y) -> X+Y end, 100, [1, 2, 3]).
106
```

- For more explanation and examples:
  - See Higher Order Functions in *Learn You Some Erlang*.
  - See the lists module in the Erlang standard library. Examples include
    - `all(Pred, List)`: true iff `Pred` evaluates to true for **every** element of `List`.
    - `any(Pred, List)`: true iff `Pred` evaluates to true for **any** element of `List`.
    - `foldr(Fun, Acc0, List)`: like `foldl` but combines elements in right-to-left order.

List Comprehensions

- Map and filter are such common operations, that Erlang has a simple syntax for such operations.
- It's called a **List Comprehension**:
  - `[`*Expr* `||` *Var* `<-` *List*, *Cond*, `...]`.
  - *Expr* is evaluated with *Var* set to each element of *List* that satisfies *Cond*.
  - Example:

```
13>R = count3s:rlist(5, 1000).
[444,724,946,502,312].
14>[X*X || X <- R, X rem 3 == 0].
[197136,97344].
```

- See also List Comprehensions in *LYSE*.

Head vs. Tail Recursion

- I wrote two versions of computing the sum of the first N natural numbers:

```
sum_h(0) -> 0; % "head recursive"
sum_h(N) -> N + sum_h(N-1).

sum_t(N) -> sum_t(N, 0).
sum_t(0, Acc) -> Acc; % "tail recursive"
sum_t(N, Acc) -> sum_t(N-1, N+Acc).
```

- Here are some run times that I measured:

N	t_{head}	t_{tail}	N	t_{head}	t_{tail}
1K	21μs	13μs	1M	21ms	11ms
10K	178μs	114μs	10M	1.7s	115ms
100K	1.7ms	1.1ms	100M	28s	1.16s
			1G	$>$ 8 min	11.6s

Head vs. Tail Recursion – Comparison

- Both grow linearly for $N \leq 10^6$.
  - The tail recursive version has runtimes about 2/3 of the head-recursive version.
- For $N > 10^6$,
  - The tail recursive version continues to have run-time linear in N.
  - The head recursive version becomes much slower than the tail recursive version.
- The Erlang compiler optimizes tail calls
  - When the last operation of a function is to call another function, the compiler just revises the current stack frame and jumps to the entry point of the callee.
  - The compiler has turned the recursive function into a while-loop.
  - Conclusion: **When people tell you that recursion is slower than iteration – don't believe them.**
- The head recursive version creates a new stack frame for each recursive call.
  - I was hoping to run my laptop out of memory and crash the Erlang runtime – makes a fun, in-class demo.
  - But, OSX does memory compression. All of those repeated stack frames are very compressible. The code doesn't crash, but it's very slow.

Tail Call Elimination – a few more notes

- I doubt we'll have time for this in lecture. I've included it here for completeness.
- Can you count on your compiler doing tail call elimination:
  - In Erlang, the compiler is **required** to perform tail-call elimination. We'll see why on Monday.
  - In Java, the compiler is forbidden from performing tail-call elimination. This is because the Java security model involves looking back up the call stack.
  - `gcc` performs tail-call elimination when the `-o` flag is used.
- Is it OK to write head recursive functions?
  - Yes! Often, the head-recursive version is much simpler and easier to read. If you are confident that it won't have to recurse for millions of calls, then write the clearer code.
  - Yes! Not all recursive functions can be converted to tail-recursion.
    - Example: tree traversal.
    - Computations that can be written as "loops" in other languages have tail-recursive equivalents.
    - But, recursion is more expressive than iteration.

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.
  - Functions: patterns, higher-order functions, head vs. tail recursion.

Preview

January 9: Processes and Messages

Reading:	*Learn You Some Erlang*, Higher Order Functions and The Hitchhiker's Guide. . . through More on Multprocessing
Homework:	**Homework 1 goes out (due Jan. 18)** – Erlang programming
Mini-Assignment:	Mini-Assignment 1 due **10:00am**
	Mini-Assignment 2 goes out (due Jan. 13)

January 11: Reduce

Reading:	*Learn You Some Erlang*, Errors and Exceptions through A Short Visit to Common Data Structures

January 13: Scan

Reading:	Lin & Snyder, chapter 5, pp. 112–125
Mini-Assignment:	Mini-Assignment 2 due **10:00am**

January 16: Generalized Reduce and Scan

Homework:	Homework 1 deadline for early-bird bonus (11:59pm)
	Homework 2 goes out (due Feb. 1) – Reduce and Scan

January 18: Reduce and Scan Examples

Homework:	Homework 1 due **11:59pm**

January 20–27: Parallel Architecture

January 29–February 6: Parallel Performance

February 8–17: Parallel Sorting

Review Questions

- What is the difference between `==` and `=:=` ?
- What is an atom?
- Which of the following are valid Erlang variables, atoms, both, or neither?

  `Foo`, `foo`, 25, `'25'`, `'Foo foo'`,
  `"4 score and 7 years ago"`, X2,
  `'4 score and 7 years ago'`.
- 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?
- Use an anonymous function and `lists:filter` to implement the body of `GetEven` below:

```
% GetEven(List) -> Evens, where Evens is a list consisting of all
%       elements of List that are integers and divisible by two.
%       Example: GetEven([1, 2, frog, 1000]) -> [2, 1000]
GetEven(List) ->
    you write this part.
```

A Few More Review Questions

- Use a list comprehension to implement to body of `Double` below:

```
% Double(List) -> List2, where List is a list of numbers, and
%     List2 is the list where each of these are doubled.
%     Example: Double([1, 2, 3.14159, 1000]) ->
%         [2, 4, 6.28318, 2000]
Double(List) ->
   you write this part.
```

- Use a list comprehension to write the body of `Evens` as described on the previous slide.
- What is a tail-recursive function?
- In general, which is more efficient, a head-recursive or a tail-recursive implementation of a function? Why?

Supplementary Material

The remaining material is included in the web-version of these slides:

http://www.ugrad.cs.ubc.ca/~cs418/2016-2/lecture/01-06/slides.pdf

I'm omitting it from the printed handout to save a few trees.

- Erlang resources.
- Finishing the merge sort example.
- Common mistakes with lists and how to avoid them.
- A few remarks about atoms.
- Suppressing verbose output when using the Erlang shell.
- Forgetting variable bindings (only in the Erlang shell).
- Table of Contents.

Erlang Resources

- *LYSE* – you should be reading this already!
- Install Erlang on your computer
  - Erlang solutions provides packages for Windows, OSX, and the most common linux distros
    https://www.erlang-solutions.com/resources/download.html
  - Note: some linux distros come with Erlang pre-installed, but it might be an old version. You should probably install from the link above.
- http://www.erlang.org
  - Searchable documentation
    http://erlang.org/doc/search/
  - Language reference
    http://erlang.org/doc/reference_manual/users_guide.html
  - Documentation for the standard Erlang library
    http://erlang.org/doc/man_index.html
- The CPSC 418 Erlang Library
  - Documentation
    http://www.ugrad.cs.ubc.ca/~cs418/resources/erl/doc/index.html
  - `.tgz` (source, and pre-compiled `.beam`)
    http://www.ugrad.cs.ubc.ca/~cs418/resources/erl/erl.tgz

Finishing the merge sort example

- Write `merge(List1, List2) -> List12` – see slide 30
- Write an Erlang modle with the `sort`, `split`, and `merge` functions – see slide 31
- Run the code – see slide 33

`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?
- Now, try writing the code (an exercise for the reader).

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!

```
1> c(sort).
{ok,sort}
2> 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]
3> 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]
4> R20 -- S20.   % empty if each element in R20 is in S20
[]
5> S20 -- R20.   % empty if each element in S20 is in R20
[]
```

- Yay – it works!!! (for one test case)
- The code is available at
  http://www.ugrad.cs.ubc.ca/~cs418/2016-2/lecture/01-06/src/sort.erl

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`.

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

Misconstructing 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:

  ```
  7> 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:

  ```
  8> examples:divisible_drop(3, [0, 1, 4, 17, 42, 100]).
  ** exception error: no function clause matching...
  ```

Punctuation

- Erlang has lots of punctuation: commas, semicolons, periods, and `end`.
- It's easy to get syntax errors or non-working code by using the wrong punctuation somewhere.
- Rules of Erlang punctuation:
  - Erlang declarations end with a period: `.`
  - A declaration can consist of several alternatives.
    - Alternatives are separated by a semicolon: `;`
    - Note that many Erlang constructions such as `case`, `fun`, `if`, and `receive` can have multiple alternatives as well.
  - A declaration or alternative can be a block expression
    - Expressions in a block are separated by a comma: `,`
    - The value of a block expression is the last expression of the block.
  - Expressions that begin with a keyword end with `end`
    - `case` *Alternatives* `end`
    - `fun` *Alternatives* `end`
    - `if` *Alternatives* `end`
    - `receive` *Alternatives* `end`

Remarks about Atoms

- An atom is a special constant.
  - Atoms can be compared for equality.
  - Actually, any two Erlang can be compared for equality, and any two terms are ordered.
  - Each atom is unique.
- Syntax of atoms
  - Anything that looks like an identifier and starts with a lower-case letter, e.g. `x`.
  - Anything that is enclosed between a pair of single quotes, e.g. `'47 BIG apples'`.
  - Some languages (e.g. Matlab or Python) use single quotes to enclose string constants, some (e.g. C or Java) use single quotes to enclose character constants.
    - ★ But not Erlang.
    - ★ The atom `'47 big apples'` is not a string or a list, or a character constant.
    - ★ It's just its own, unique value.
  - **Atom constants can be written with single quotes, but they are not strings.**

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

```
9> L1_to_5 = lists:seq(1, 5).
[1, 2, 3, 4, 5].
10> L1_to_5M = lists:seq(1, 5000000), ok.
ok
11> length(L1_to_5M).
5000000
12>
```

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.

```
12> X = 2+3.
5.
13> X = 2*3.
** exception error:  no match of right hand side value 6.
14> f(X).
ok
15> X = 2*3.
6
16>
```

Table of Contents