CS11 – Erlang

Winter 2012-2013

Lecture 1

Welcome!

- Aim for 8 lectures, 8 labs
- Slides posted on CS11 website
  - http://courses.cms.caltech.edu/cs11
- A CS cluster account is required
- Submit all assignments through csman
  - csman uses CS cluster account for authentication
  - http://csman.cs.caltech.edu
- Can also use the Erlang installation on the CS lab machines, if you wish

Assignments and Grading

- Assignments posted on Erlang track page
  - Usually available around lecture time
  - Due one week later, on Tuesday at 12:00 noon
- Assignments will receive a score in range 0..3
  - Required fixes will be noted in graded work
- Late assignments will receive a 0.5 point/day deduction
- Must receive ~3/4 of the available points to pass the CS11 Erlang track
  - e.g. for 8 assignments: 18/24 points
  - e.g. for 7 assignments: 15.5/21 points

Erlang/OTP Platform

- Current version of Erlang/OTP is R15B02
  - I will grade with some variant of R15B
- Can download and install a local copy
  - URL: http://www.erlang.org/download.html
  - Windows binary is available
  - For other platforms, check Erlang Solutions website:
    - https://www.erlang-solutions.com/downloads/download-erlang-otp
    - Prebuilt install packages for many platforms
  - Or, download the source code and build it yourself
    - (That's what I do.)

Erlang Books!

- Several very useful Erlang books!
  - Not required for the course, but get them if you want to continue learning the language
- Programming Erlang by Joe Armstrong
  - Basic intro to Erlang syntax and programming
  - Very light coverage of Erlang libraries (the OTP)
- Erlang Programming
  - Cesarini and Thompson (O'Reilly book)
- Erlang and OTP in Action
  - Logan, Merritt, Carlsson
  - First book focusing primarily on the OTP

Erlang is…

- Concurrency-oriented programming language
  - Focus on distributed computing and concurrency
  - Supports <u>many</u> lightweight processes
    - e.g. thousands, tens of thousands, or more!
  - Processes communicate only using messages
    - No locks, no shared memory
- Focuses on fault-tolerance and robustness
  - Processes can monitor each other for failure conditions
  - When a process dies, it automatically sends signals to all listening processes

Erlang is… (2)

- A functional programming language
  - No in-place mutation of state!
  - Supports higher-order functions
    - (but not all functions are higher-order functions…)
  - Has no explicit looping statements (e.g. for, while)
    - Must implement looping with recursive calls
    - Supports tail-call optimization for efficient recursion
- A virtual machine-based language
  - Source is compiled into bytecodes and executed in an Erlang emulator
  - Also allows for hot-swapping of code in running system, for upgrades/bug-fixes without downtime

Erlang/OTP

- Erlang also includes the OTP
  - OTP = Open Telecom Platform
- OTP is a set of tools and libraries for building large-scale, fault-tolerant distributed apps
  - Apps that basically never go down
    - Process crashes are handled automatically
    - System upgrades can be performed on running system
  - Apps that can provide soft-realtime guarantees
    - Processes can be added/removed to scale with load
    - Software can be run on a cluster of machines
- OTP includes many useful libraries
  - HTTP server, XML parsing, distributed database, …

Erlang/OTP (2)

- Erlang and OTP were developed at Ericsson
  - Swedish telecommunications company
- Originally developed in 1986
- Open-sourced in 1998
- Name “Erlang” is dual:
  - Agner Krarup Erlang (1878–1929)
    - Danish mathematician who invented fields of traffic engineering and queuing theory
  - A nice coincidence: also a contraction of “Ericsson Language”

Hello World, Erlang Style

- Traditional “hello world” program in Erlang:

```
-module(world).
-export([hello/0]).

% Tell the world hello!
hello() -> io:format("Hello world!~n").
```

- All Erlang code is structured into modules
  - This module’s name is “**`world`**”
  - Module’s source must be stored in file “**`world.erl`**”
    - Module name and filename must match

Compiling and Running

- Compile and run our Erlang program:

```
erlc world.erl
erl
1> world:hello().
Hello world!
ok
2>
```

- **`erlc`** is the Erlang compiler
  - Compiles **`world.erl`** into **`world.beam`**
- **`erl`** is the Erlang shell
  - Interactive console for running and interacting with Erlang programs

Compiling from `erl`

- Can also compile/run entirely within **`erl`**

```
erl
1> c(world).
{ok,world}
2> world:hello().
...
```

- **`c(module)`** command compiles **`module.erl`** and then loads it
  - **`c(module.erl)`** or **`c("module.erl")`** also works

Erlang Statements, Comments

- Our “hello world” program:

```
-module(world).
-export([hello/0]).

% Tell the world hello!
hello() -> io:format("Hello world!~n").
```

- Statements are terminated with a period
  - Erlang syntax generally follows English punctuation usage
- Comments start with % and extend to end of line
  - No block-comments in Erlang

Module Attributes

- Our “hello world” program:

```
-module(world).
-export([hello/0]).

% Tell the world hello!
hello() -> io:format("Hello world!~n").
```

- Statements starting with “–” specify module attributes
  - **-module(*name*)** specifies the module’s name
  - **-export([*functions*])** specifies list of functions callable from outside this module
  - Many other attributes, as well as user-defined ones!

Erlang Data Type Overview

- Erlang has a relatively small set of data types
- Integers: arbitrary-size whole numbers
  - 1, -65, 36893488147419103232 (= 2^{65})
- Floats: double-precision floating point numbers
  - 3.14159, 6.022e23
- Atoms: named symbolic constants
  - e.g. **`ok`**, **`world`**, **`red`**, **`title`**
  - First character must be a lowercase letter
    - If doesn't start with lowercase character, must be enclosed with single-quotes, e.g. **`'Monday'`**
    - Subsequent characters are alphanumeric, underscore "_" or at-sign "@"

Erlang Data Type Overview (2)

- Booleans are represented by **`true`** and **`false`** atoms
  - No actual Boolean *data type* in Erlang
  - Various logical operators that act on these atoms
- Lists of values enclosed with []
  - Elements separated with commas
  - Any number and type of elements
  - e.g. **`[1, true, 3.14, red]`**
- Tuples are a compound data type with fixed number of terms
  - Enclosed with {}
  - e.g. **`{ok,world}`**, or **`{point, 5.1, 2.3}`**

Floating-Point Arithmetic

- Floating point numbers have decimal point and one or more digits to right of decimal point
  - If no digits to right of decimal point, Erlang thinks the period ends the statement…
- For addition, subtraction, multiplication:
  - If any operand is a floating-point number, the result is a floating-point number
- Division operator "/" <u>always</u> produces a float!

```
1> 4 / 2.
2.00000
```

Integer Division and Remainder

- For integer division, use **div** and **rem**
- Examples:

```
1> 7 div 3.
2
2> 7 rem 3.
1
3> 7 / 3.
2.33333
```

Erlang Function Details

- Functions are uniquely defined by module name, function name, and arity
  - Arity = number of arguments
  - Argument and return types are not specified
- From our example:

```
-export([hello/0]).
hello() -> io:format("Hello world!~n").
```

- **`-export`** specifies list of functions to export
  - Each element is of form "**`name/arity`**"
  - Can list multiple functions, separating with commas

```
-export([hello/0, hello/1, goodbye/0]).
```

  - Can also specify multiple **`-export`** statements

Erlang Function Details (2)

- To call a function in another module, specify **module:function(args)**
- From our example:

```
hello() -> io:format("Hello world!~n").
```

- Can also use **-import()** module-attribute to import functions into a module
  - Can call imported functions as if they were local
  - Syntax: **-import(*module*, [*function*/*arity*, ...]).**
- Example:

```
-export([hello/0]).
-import(io, [format/1]).
hello() -> format("Hello world!~n").
```

Erlang Function Declarations

- Functions are defined with the syntax:

```
name(Arg1, Arg2, ...) -> body.
```

- Function name is an atom
  - Use underscores to separate words, e.g. **`is_even()`**
  - Functions that return Boolean values named **`is_xxxx()`**
- Previous example:

```
hello() -> io:format("Hello world!~n").
```

- Can specify multiple statements by separating with commas, ending with period.

```
print_square(X) ->
    SqX = X * X, io:format("X^2 = ~w~n", [SqX]).
```

  - **`io:format()`** is like C's **`printf()`** function
    - Takes a format specification and a list of values to plug in

Erlang Variables

- Erlang variables must start with a capital letter or an underscore “_”
  - Subsequent letters may be alphanumeric, at-sign “@”, or underscore “_”
- Erlang variables are single-assignment
  - Once variable is bound to a value, it cannot change!
- Example:

```
1> Mass = 45.
45
2> Mass = 15.
=ERROR REPORT====  etc.
```

Matching and Binding

- Variables may be bound to a value, or they may be unbound
  - Once variable is bound to a value, it cannot change
- **=** is a pattern-matching operator
  - Not simple assignment! Not equality comparison!
- Form: ***`pattern = term`***
  - A term is any valid Erlang expression, but all variables *must be* bound.
  - A pattern is like a term, but may also contain unbound variables.
  - If the pattern matches the term, unbound variables in pattern are bound to corresponding values in the term

Matching and Binding (2)

- Previous example:

  ```
  1> Mass = 45.
  45
  ```

  - Matches pattern **`Mass`** with term **`45`**
    - They match since **`Mass`** is unbound; **`45`** is bound to **`Mass`**

  ```
  2> Mass = 15.
  =ERROR REPORT====  ...
  ** exited: {{badmatch,15}, ... } **
  ```

  - Tries to match pattern **`Mass`** with term **`15`**
    - **`Mass`** is already bound to **`45`**, so match fails!
- This is a *very* simple example of pattern matching, but it is a very powerful feature

Lists in Erlang

- Erlang supports lists, enclosed with `[]`
  - Elements are separated by commas
  - Elements may be of different types
- Already saw several lists:

```
-export([hello/0]).
-import(io, [format/1]).
io:format("X^2 = ~d~n", [SqX]).
```

- Can easily specify lists of values in your code

```
Colors = [blue, red, green, yellow].
```

  - Can also include bound variables, to store their values into the list, e.g. `[SqX]` above
- Empty list is just `[]`

Lists in Erlang (2)

- Lists are divided into `[Head|Tail]` components
  - Head = first element of list
  - Tail = another list, containing rest of the elements
- A single-element list `[a]` is actually `[a|[]]`
  - Head of list is the atom `a`
  - Tail of list is the empty list `[]`
- The `|` operator lets you break apart a list this way
- Examples:

```
Values = [3, 4, 5].
[X | Y] = Values.
```

- X = 3, Y = [4, 5]

```
MoreValues = [1, 2 | Values].
```

- MoreValues = [1, 2, 3, 4, 5]

Improper Lists

- Definitely possible to construct improper lists in Erlang

  ```
  Improper = [a|b].
  ```

  - Both **`a`** and **`b`** are atoms. Tail of list is not another list.
  - Displays as **`[a|b]`** in Erlang shell
- Using **`|`** to split apart improper list gives back both atoms

  ```
  [V1 | V2] = Improper.
  ```

  - **`V1`** = **`a`**, **`V2`** = **`b`**

List Concatenation, Subtraction

- Can use **++** operator to concatenate lists

  ```
  Part1 = [1, 2, 3].
  Part2 = [4, 5, 6].
  Complete = Part1 ++ Part2.    % [1, 2, 3, 4, 5, 6]
  ```

- The **--** operator performs list subtraction
  - `List1 -- List2`
  - For each element in second list, the first matching element in the first list is removed
- Examples:

  ```
  [a, a, b, b, c, c] -- [a, b, c].
  ```

  - Evaluates to `[a, b, c]`

  ```
  [a, a, b, b, c, c] -- [a, b, c, b].
  ```

  - Evaluates to `[a, c]`

Multiple Function Clauses

- Can specify multiple clauses for functions:

```
name(Pattern11, Pattern12, ...) -> body1;
name(Pattern21, Pattern22, ...) -> body2;
...
name(PatternN1, PatternN2, ...) -> bodyN.
```

  - A "function clause" is the name/arguments/body combination
- Example: factorial function

```
factorial(1) -> 1;
factorial(N) -> N * factorial(N - 1).
```

  - First clause handles base case; second clause handles recursive case.
- **Note:** first matching clause is chosen!
  - If `factorial(N)` clause came first, this wouldn't work.

Function When-Guards

- Can also specify a when-guard for any function clause
- Factorial function, take two:

```
factorial(N) when N > 1 ->
    N * factorial(N - 1);
factorial(1) -> 1.
```

  - Now, first clause only matches when N > 1
- When-guards can only contain simple tests!
  - Simple arithmetic and comparisons, or combinations of these tests
  - e.g. can’t call your own Boolean function
  - See Erlang reference documentation for details

Lists and Matching

- Can use list constructs in pattern-matching expressions too

```
% This function sums up a list of numbers.
sum([Value|Rest]) -> Value + sum(Rest);
sum([]) -> 0.
```

- **Pop quiz:** What is the arity of `sum`?
  - `sum/1` – the list is a single argument
- Can even construct more clever list-matching expressions
- What does this pattern match: `[X|[X|Rest]]`
  - A list with at least two identical elements at start
  - Can also write: `[X, X | Rest]`

Recursion in Erlang

- Not all recursion is equal!
  - When a function recursively calls itself, is there still more work to do on previous invocation?
- Example: factorial function

```
factorial(1) -> 1;
factorial(N) -> N * factorial(N – 1).
```

- When factorial(N) calls factorial(N – 1):
  - Recursive call for N – 1 must complete before factorial(N) can complete its computation
  - Produces a series of deferred operations which must be evaluated *after* full recursion completes

Recursion in Erlang (2)

- If the recursive call is the very last operation in a function, this is called tail recursion
  - Since no more operations are required for current iteration, no extra resources are consumed
- Tail-recursive factorial function:

```
factorial(N) -> factorial_helper(N, 1).
factorial_helper(1, Result) -> Result;
factorial_helper(N, Result) ->
    factorial_helper(N - 1, Result * N).
```

  - All arguments are evaluated before a call is made
  - When **`factorial_helper`** calls itself, no more work to do for current invocation.

Recursion in Erlang (3)

- Tail-recursive factorial function:

```
factorial(N) -> factorial_helper(N, 1).
factorial_helper(1, Result) -> Result;
factorial_helper(N, Result) ->
    factorial_helper(N - 1, Result * N).
```

  - Erlang optimizes tail-recursive calls so that they use no extra stack space.
- Important note!
  - Only `factorial/1` should be exported!
  - `factorial_helper/2` should be kept private
    - It is an internal implementation detail for the module

Flow-Control in Erlang

- Erlang includes only very simple flow-control constructs
- All looping must be implemented via recursive calls
  - Tail-recursive calls are strongly encouraged for efficiency and performance reasons!
- Two main flow-control constructs:
  - `if` statements
  - `case` statements
- This time: `if`
- Next time: `case`

Erlang `if` Statements

- General form:

```
if
     cond1 -> body1;
     cond2 -> body2;
     ...
end
```

  - Use a condition of **`true`** for an else-clause
  - First clause with condition that evaluates to true is used
    - Corresponding body is evaluated, and body's result is the result of entire if statement
  - A "body" is either a single statement, or multiple statements separated by commas (as before)
  - If no clause matches, a runtime error is generated
    - The if statement must evaluate to *some* value…

Erlang `if` Statements (2)

- Style suggestions:
- If body is a single short statement, put on same line

```
if
    cond1 -> body1;
    cond2 -> body2;
    ...
end
```

- If body includes multiple statements, put on next line

```
if
    cond1 ->
        body1;
    cond2 ->
        body2;
    ...
end
```

Factorial Function with `if`

- Factorial function, take 3:

```
factorial(N) ->
    if
        N == 1 -> 1;
        true   -> N * factorial(N – 1)
    end.
```

- Definitely not tail–recursive…

Erlang Comparison Operators

- Comparison operators in Erlang:
  - **==** equal to
  - **/=** not equal to
  - **<** less than
  - **=<** less than or equal to
  - **>=** greater than or equal to
  - **>** greater than
- These operators can compare different types
  - e.g. integer and float
  - Values are coerced into a common type, then compared
- These comparison operators *do not* coerce:
  - **=:=** exactly equal to
  - **=/=** exactly not equal to

Erlang Logical Operators

- Several logical operators for working with Boolean values (**`true`** and **`false`**)

```
Val1 and Val2
Val1 or Val2
Val1 xor Val2
not Val
```

  - **Important note:** NONE of these operators short-circuit!
    - They always evaluate both arguments
- Short-circuiting logical operators:

```
Val1 andalso Val2
Val1 orelse Val2
```

    - Second argument is only evaluated when necessary

Final Notes for This Week

- Very useful Erlang/OTP documentation online
- We are using version R15B… for this track
  - URL for R15B02 Documentation:
  - http://www.erlang.org/doc/
- Useful links:
  - “Modules” link at top of lefthand frame for list of all standard modules, e.g. **`io`**, **`lists`**, **`erlang`**, etc.
  - Erlang Programming section in lefthand frame:
    - “Getting Started” for some basic tutorials
    - “Erlang Reference” for more formal language details

This Week's Assignment

- Write some simple modules and functions in Erlang
- Practice writing recursive functions, and especially tail-recursive functions
  - Will become important very quickly!
- Compile your modules and run them from the Erlang shell

Next Week!

- Jump straight into Erlang concurrency!
  - Writing simple server processes
  - Starting processes
  - Passing messages to processes
- More details about Erlang language
  - Tuples
  - More pattern-matching details