Introduction to Erlang

Mark Greenstreet

CpSc 418 – Sept. 11 & 13, 2012

Outline:

- Why Erlang?
- Erlang by Example

Why Parallel Programming is Hard

- Programming is hard.
- Parallel programming adds more complexity:
  - Finding parallelism.
  - Coordination: avoiding races and deadlocks.
  - Keeping overhead under control.
- We need to simplify something to make cognitive room for parallelism:
  - Example: Google's map-reduce paradigm.
    Everything is divide-and-conquer (also Hadoop).
  - Example: nVidia's data parallelism – CUDA.
    Everything is a big, homogeneous array.
  - Example: Parallel functional programming: Erlang
    Everything is side-effect free.

Functional Programming and Erlang

- Programming without state.
- Referential transparency.
- Life without loops.
- Definitions vs. recipes.
- Thanks: this section was adopted from slides that Kurt Eiselt prepared for CPSC 312.

"*A language that doesn't affect the way you think about programming is not worth knowing.*" (Alan Perlis)

What is Functional Programming?

- Imperative programming (C, C++, Java, Ruby, Fortran, perl, . . .) is a programming model that corresponds to the von Neumann computer:
  - A program is a sequence of statements.
    Each statement can be translated into a sequence of machine instructions.
  - Control-flow (`if`, `for`, `while`, function calls, etc.)
    Each control-flow construct can be implemented using branch, jump, and call instructions.
- 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.

Programming and State

- In an imperative program, statements modify the values of variables. For example,
  - `x = y+3;` sets the value of `x` to the sum of the value of `y` and `3`.
  - The old value of `x` is overwritten (i.e. destroyed).
  - Note that this is what make debugging hard:
    - ★ You can see that your program computed an incorrect value or reached a point in the control-flow where it shouldn't be.
    - ★ BUT you can't see how it got there, because intermediate results that led to this point are now gone.
- In a functional language, declarations associate values with variables.
  - A variable gets a value when it is declared.
  - This value is never changed.

Referential Transparency

- In a functional program, every function call with the same parameters returns the same result. Every time. This is a result of a mathematical and functional programming principle called referential transparency.
- Thus, `cos(`π`/4) = `$\sqrt{2}/2$ every time you call `cos`. You don't get different values for the cosine of the same argument with different calls.
- Isn't this obvious?
  - Apparently not. In imperative languages (such as C or Java) a function can have side effects; it can change the value of global state:

    ```
    int countCalls(args ...)  {
        static ncalls = 0;
        return(++ncalls);
    }
    ```

    Successive calls to `countCalls` return different values.
  - We rely on this: I/O functions, memory allocation, object construction, and much, much more.

Side Effects

- As noted above, imperative languages rely on having functions with side effects.
- But, if a function (e.g. `cos`) has side-effects and returns different values on different calls with the same argument, most of us will get confused.
- How do we know when a function has side-effects?
  - "It should be 'obvious' when you think about what the function does."
  - BUT, if you think about the function differently than I do, and we are working on the same project, life can get very confusing very quickly.
  - So, we need to document all of the side-effects, and pay attention to the documentation. Of course, this doesn't really happen in the real world.
- Functional programming solves these problems by excluding side-effects.
  - Of course, that means we'll have to think about things like I/O, memory allocation, loops and other constructs in a different way.
  - This is what makes functional programming both much easier and much harder than imperative programming.

Back to Referential Transparency

- It's not difficult to see that referentially-transparent programs are easier to work with (e.g., make correct, debug, prove correct) than those that are referentially opaque. Processing referentially opaque programs also requires more complex compilers or interpreters.
- The bad news is that to make referentially-transparent programs, you have to learn how to write programs that don't rely on side effects. And that means you'll have to give up your beloved assignment statements.
- In ten words or less, that's functional programming: you give up side effects to gain referential transparency.

Getting Erlang

- You can run erlang by giving the command `erl` on any departmental machine. For example:
  - Linux: bowen, lin01, . . . , lin25, . . . ,
  - Solaris: galiano, gambier

  all machines above are .ugrad.cs.ubc.ca, e.g. bowen.ugrad.cs.ubc.ca, etc.
- Or, download it for your own computer.
  - See `http://www.erlang.org/download.html`
  - I followed the instructions at

    `http://sacharya.com/erlang-on-mac-osx/`

    to install Erlang on my laptop (OSX snow-leopard).

Starting Erlang

- Start the erlang interpretter.

```
gambier % erl
Erlang R14B (erts-5.8.1) [source]
  [smp:64:64][rq:64][async-threads:0]
  [kernel-poll:false]
Eshell V5.8.1 (abort with ^G)
1> 2+3.
5
2>
```

- The erlang interpreter evaluates expressions that you type.
- Expressions end with a "`.`" (period).

Factorial in Erlang

- `m1.erl`:

```
-module(m1).    % This module is named m1
-export([fac/1]).    % m1 exports one function, named fac
fac(0) -> 1;             % Base case;
fac(N) -> N*fac(N-1).    % recursive case.
```

- Let's try it:

```
2> c(m1).
{ok,m1}
3> m1:fac(1).    % invoke a function with ModuleName:FunctionName(Args).
1
4> m1:fac(3).
6
5> m1:fac(100).
933262154439441526816992388562667004907159682643816214
685929638952175999932299156089414639761565182862536979
208272237582511852109168640000000000000000000000000
6> m1:fac(0).
1
```

Factorial Execution

```
fac(3)                      % matches fac(N) -> N*fac(N-1)
   ->3*fac(2)               % matches fac(N) -> N*fac(N-1)
   ->3*(2*fac(1))           % matches fac(N) -> N*fac(N-1)
   ->3*(2*(1*fac(0)))       % matches fac(0) -> 1
   ->3*(2*(1*1))            % now do the arrithmetic
   ->3*(2*1)                % keep going
   ->3*2                    % we're almost there.
   ->6                      % Done: 3! = 6, just like it should.
```

Integers and Floats

- Integer constants are pretty much what you would expect.
  - Integers can be arbitrarily large. (See slide 11 for an example.)
  - `$c` is the ASCII value of the character `c`.
  - `base#value` is an integer constant represented in base `base`. `base` must be an integer in 2. . . 36.
- Floating point constants are pretty much what you would expect.
  - Erlang requires at least one digit on each side of the decimal point.
  - `1e3` is not a valid erlang floating point constant.
  - `1.0e3` is ok.
- Examples:

```
7> $e.
101 % ASCII for e
8> 16#DE1.
3553 % Hexadecimal
9> 16#2a.
42 % Upper or lower case ok
```

```
10> 2e3.
* 1: syntax error before: e3
11> .2e4.
* 1: syntax error before: 2
12> 2.0e3.
2.0e3
```

Variables

- A variable name 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`.
- Examples: `X`, `R2D2`, `A_Erlang_1_1_1878`.
- When a variable is declared, it must be bound to a value. For example,

```
X = 12.
```

- Once the value of a variable is bound, it cannot be changed.
- See also: atoms, patterns.

Variables – Examples

```
13> X = 12.
12
14> X = 99.    % try to change the value of X.
** exception error:  no match of right hand side value 99
15> X = 12.
12 % The value of X is unchanged.
16> X = 12.0.
** exception error:  no match of right hand side value 12.0
17> f(X).
ok % The Erlang shell lets you forget, f(), a variable.
   % You can't use f() in .erl files.
18> X = 42.17.
42.17
19> A = B.
* 1: variable 'B' is unbound
20>
```

Arithmetic Operations

- Binary operations:

  $\star$, /, `div`, `rem` $\succ$ `band` $\succ$ +, − $\succ$ `bor`, `bxor`, `bsl`, `bsr`
- Unary operations: +, −, `bnot`
- Precedence and associativity:
  - `rem` $\succ$ `band` indicates that `rem` had higher precedence than `band`.
  - Operators separated commas in the list above have the same precedence.
  - The three unary operators have higher precedence than any of the binary operators.
  - All Erlang binary arithmetic operators are left-associative.
- Bitwise binary operations:

`band`: and	`bnot`: not	`bor`: or	`bxor`: exclusive-or
`bsl`: arithmetic shift left		`bsr`: arithmetic shift right	

Arithmetic and Types

- `div` and `rem` are integer division and remainder respectively. Their operands must be integers, and they produce integer results.
- The operands for bit-wise boolean operators must be integers, and they produce integer results.
- `/` is floating-point division:
  - It's operands can be any mix of integers or floats.
  - The result is always a float.
- `+`, `-`, and `*` are addition, subtraction, and multiplication as expected:
  - Their operands can be any mix of integers or floats.
  - If both operands are integers, then the result is an integer.
  - If one or both operands are floats, then the result is a float.
  - If one operand is an integer, and the other is a float.
    - ★ The integer value is "promoted" to a float.
    - ★ If this causes an overflow, and an error occurs.

Arithmetic – Examples

```
20> 17 + 5.
22
21> 17 - 5.
12
22> 17 * 5.
85
23> 17 / 5.
3.4
24> 17 div 5.
3
25> 17 rem 5.
2
26> bnot 17.
-18
27> 17 band 5.
1
28> 17 bor 5.
21
```

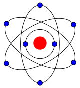
```
29> 17 bxor 5.
20
30> 17 bsl 5.
544
31> 42 bsr 3.
5
32> bnot 17 bsl 5.
-576
33> bnot (17 bsl 5).
-545
5
34> 17 * -5.0.
-85.0
35> 17 div 5.0.
** exception error: bad argument in
     an arithmetic expression
       in operator div/2
         called as 17 div 5.0
```

Arithmetic – more examples

```
36> A = m1:fac(200).
7886578673647905035523632139321850622951359776871732632947425332443594499634033429203042
8401198462390417721213891963883025764279024263710506192662495282993111346285727076331723
7396988943922445621451664240254033291864131227428294853277524242407573903240321257405579
5686602260319041703240623517008587961789222227896237038973747200000000000000000000000000
00000000000000000000000
37> A + 1
7886578673647905035523632139321850622951359776871732632947425332443594499634033429203042
8401198462390417721213891963883025764279024263710506192662495282993111346285727076331723
7396988943922445621451664240254033291864131227428294853277524242407573903240321257405579
5686602260319041703240623517008587961789222227896237038973747200000000000000000000000000
00000000000000000000001
38> A + 1.0
** exception error: bad argument in an arithmetic
   expression
     in operator +/2
        called as 788657867364790503552363213932185062295135977687173263294
        742533244359449963403342920304284011984623904177212138919638830257642790
        242637105061926624952829931113462857270763317237396988943922445621451664
        240254033291864131227428294853277524242407573903240321257405579568660226
        031904170324062351700858796178922222789623703897374720000000000000000000
        000000000000000000000000000000 + 1.0

39>
```

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: patterns. variables,

Comparisons

- Erlang has a the usual set of comparison operators:
  - `<`, `=<`, `==`, `/=`, `>=`, `>`
  - Most of these are like their C/C++/Java equivalents.
  - `=<` is "less-than-or-equal-to" and `/=` is "not-equal-to".
- Erlang has two special operators for comparing integers and floating point numbers:
  - `=:=` "exactly equal to" – `X =:= Y` iff `X == Y` and if `X` is an integer, `Y` is an integer too; and if `X` is a float, then `Y` is a float too.
  - `=/=` "exactly not-equal to" – the logical negation of `=:=`.
- Examples:

```
39> 2 < 3.           | 42> 2 =:= 2.0.      | 45> tom == mary.
true                 | false               | false
40> 2 == 2.0.        | 43> 2 =/= 2.0.      | 46> tom == 2.
true                 | true                | false
41> 2 < 2.0.         | 44> 2.0 =:= 2.0.    | 47> tom == tom.
false                | true                | true
```

Boolean Expressions

- Erlang represents boolean values with the atoms `true` and `false`.
- Erlang has the unary boolean operator `not`.
- Erlang has the binary boolean operators: `and` ≻ `or`, `xor`.
  - The binary operators always evaluate both operands, even if the result is determined by the left-operand.
  - Boolean operators have higher precedence than comparisons – use parentheses.

```
48> (2 < 3) or (0 == 5).
true
49> 2 < 3 or false.   % or has higher precedence than <
** exception error: bad argument
     in operator or/2
        called as 3 or false
50> (2 < 3) or (17 div 5 < 0).
true
51> (2 < 3) and (17 div 5 < 0).
false
```

Short-Circuit Booleans

- `andalso` only evaluates its second operand if its first operand evaluates to `true`.
- `orelse` only evaluates its second operand if its first operand evaluates to `false`.

```
52> (2 < 3) or (17.0 div 5 < 0).
** exception error: bad argument in
        an arithmetic expression
     in operator div/2
        called as 17 div 5.0
53> (2 < 3) orelse (17.0 div 5 < 0).
true
```

Lists

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

```
54> L1 = [1, 2+3, 4+5*6].        % declare a list by stating its elements
[1,5,34]
55> L2 = [0 | L1].               % prepend an element to a list
[0,1,5,34]
56> L3 = [L1, L2, foo].          % lists can be nested
[[1,5,34],[0,1,5,34],foo]
57> L4 = [L1 | L2].              % Prepend L1 as a single element to L2.
[[1,5,34],0,1,5,34]
58> L5 = L1 ++ L2.               % ++ denotes list concatenation.
[1,5,34,0,1,5,34]
59> L6 = [L1 | L2 | L3].
* 1: syntax error before: '|'
```

- See also: list operations, tuples.

Pattern Matching

- Erlang makes extensive use of pattern matching.
  - The examples on this slide are very simple because of the small Erlang fragment that we have so far.
  - More extensive examples will occur on subsequent slides.
- Simple example:

```
60> [Head | Tail] = L1.   % L1 declared on slide 24.
[1,5,34]
61> Head.
1
62> Tail.
[5,34]
```

  - `Head` and `Tail` were unbound before executing command 60.
  - The Erlang run-time finds if there is a way to choose values for `Head` and `Tail` such that the left side of the `=` operator, `[Head, Tail]`, matches the right side, `L1`.
  - The Erlang run-time finds such a choice of values and sets `Head` and `Tail` accordingly.
  - If there's no way to make a match, then an error is reported.

More Matching

- The general form for matching is: *LeftSide* = *RightSide*.
- *LeftSide* can be an expression of erlang values and unbound variables combined using lists and tuples.
- *RightSide* can be an arbitrary expression.
- Examples:

```
63> [1 | X1] = L1.
[1,5,34]
64> X1.
[5,34]
65> [A1, B1, 2*17] = L1. % The compiler replaces 17*2 with 34.
[1,5,34]
66> [1, 5, 2*C1] = L1.
* 1: illegal pattern % But it's not a general equation solver!
67> [_, B2, _] = L1.
[1,5,34]
68> B2.
5
```

Modules

- Erlang code is arranged in modules.
  - The code for module `foo` should be in a file called `foo.erl`.
  - Erlang supports organizing groups of related source files into packages.
    - We won't use packages here, but you can learn about them at: `http://www.erlang.se/publications/packages.html`
- An erlang module is a list of attributes followed by a list of function declarations.
- Syntax for attributes: `-`*Name*`(`*Value*`).`
  - `-module(`*ModuleName*`).`
    ModuleName must match the file name without the `.erl` extension. *ModuleName* must be an atom.
  - `-export([`*fun1*`/`*arity1*`,` *fun2*`/`*arity2*`, ...]).`
    This module exports a function named *fun1* that has *arity1* arguments. *fun1* must be an atom, and *arity1* must be a non-negative integer.
- Syntax for functions: see next slide.

Function Declarations

- Syntax:

  FunctionName(*ArgList1*) -> *Expr1*;
  FunctionName(*ArgList2*) -> *Expr2*;
  ...
  FunctionName(*ArgListN*) -> *ExprN*.

- *FunctionName* is an atom, the name of the function.
- *ArgList1* is a list of arguments: *Arg1*, *Arg2*, ..., *ArgK*, where `K` is the arity of the function.
  - If *FunctionName* is invoked with parameters that match the pattern of *ArgList1*,
  - then the expression for *Expr1* is evaluated to produce the return value for the function.
  - otherwise the other patterns are tried, in order until
    - ★ A match is found, or
    - ★ The last alternative is tried, and fails to match. In this case, an error is reported.

Function Declarations (continued)

- *FunctionName* must be the same atom for all alternatives.
- Each of the *ArgList*'s should have be a different pattern, but they must each have the same number of arguments.
- Alternatives are separated by semicolons; the final alternative is terminated with a period.
- Example: factorial (again)

```
-module(m1).
-export([fac/1]).

fac(0) -> 1;
fac(N) -> N*fac(N-1).
```

Crashing factorial

- Consider the factorial function from slide 29.
- What happens if I give the command:

```
69> m1:fac(-1).
% well, I waited a few minutes and then
beam.smp(5555,0xb0250000) malloc: ***
     mmap(size=1140850688) failed...
```

  and about 15 more lines of error message as the Erlang interpretter crashes.
- Maybe I shouldn't do that. ☺
- The problem was that `-1` matched the pattern `fac(N)`.

Using a “when” clause

- A pattern may be qualified by a when clause.
- Example:

```
fac2(0) -> 1;
fac2(N) when N > 0 -> N*fac2(N-1).
```

- Let’s try it:

```
70> c(m1).
{ok,m1}
71> m1:fac2(0).
1
72> m1:fac2(3).
6
73> m1:fac2(-1).
** exception error: no function clause matching
     m1:fac2(-1)
```

When clauses

- Syntax: `when` *guard*
- Simple version: *guard* is a boolean-valued expression
  - The guard can consist of constants, variables, arithmetic and boolean operations, and comparisons.
  - Erlang is restrictive about what functions you can use.
    - ★ built-in functions that have no side-effects.
    - ★ some handy ones: `length(List)`, `element(N, Tuple)`, `is_integer(X)`, `is_list(X)`, `is_tuple(X)`, ...
- More elaborate guards can be written. See Erlang Language Reference – Expressions→Guard Sequences

Tuples

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

```
74> T1 = {cat, dog, potoroo}.
{cat,dog,potoroo}
75> L6 = [ {cat, 17}, {dog, 42}, {potoroo, 8}].
[{cat,17}, {dog,42}, {potoroo,8}]
76> element(2, T1).
dog
77> T2 = setelement(2, T1, banana).
{cat,banana,potoroo}
78> T1.
{cat,dog,potoroo}
```

- Lists vs. 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.).

Another example

Let's make a module for common operations on matrices:

- Functions:
  - `add`, `mult`, `transpose`
  - `lu` – LU decomposition
  - …
- Matrix representation, a list of lists:

```
A = [ [1, 2],
      [3, 4]
    ].
```

$$\equiv \quad A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

  - This representation is problematic for some empty matrices.
  - For example, `A = [[], [], []]` is a 3×0 empty matrix, but `transpose(A)`, a 0×3 empty matrix, has no representation.
  - We're just using this as a simple example $\rightarrow$
    We won't bother with empty matrices.

Matrix multiplication

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 7 & 5 & 11 \\ 9 & -2 & 6 \end{bmatrix} = \begin{bmatrix} 25 & 1 & 23 \\ 57 & 7 & 57 \end{bmatrix}$$

- The element in the *i^th^* row and *j^th^* column of the product is the sum of the element-wise products of
  - The elements of the i^{th} row of the left multiplicand and
  - the elements of the j^{th} column of the right multiplicand.
- This requires that the number of columns of the left multiplicand must be the same as the number of rows of the right multiplicand.

Matrix multiplication

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 7 & 5 & 11 \\ 9 & -2 & 6 \end{bmatrix} = \begin{bmatrix} 25 & 1 & 23 \\ 57 & 7 & 57 \end{bmatrix}$$

$$\begin{aligned} & 1 * 7 + 2 * 9 \\ = \; & 7 + 18 \\ = \; & 25 \end{aligned}$$

- The element in the i^{th} row and j^{th} column of the product is the sum of the element-wise products of
  - The elements of the i^{th} row of the left multiplicand and
  - the elements of the j^{th} column of the right multiplicand.
- This requires that the number of columns of the left multiplicand must be the same as the number of rows of the right multiplicand.

Matrix multiplication

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 7 & 5 & 11 \\ 9 & -2 & 6 \end{bmatrix} = \begin{bmatrix} 25 & 1 & 23 \\ 57 & 7 & 57 \end{bmatrix}$$

$$\begin{aligned} & 1 * 5 + 2 * (-2) \\ = \; & 5 + -4 \\ = \; & 1 \end{aligned}$$

- The element in the i^{th} row and j^{th} column of the product is the sum of the element-wise products of
  - The elements of the i^{th} row of the left multiplicand and
  - the elements of the j^{th} column of the right multiplicand.
- This requires that the number of columns of the left multiplicand must be the same as the number of rows of the right multiplicand.

Matrix multiplication

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 7 & 5 & 11 \\ 9 & -2 & 6 \end{bmatrix} = \begin{bmatrix} 25 & 1 & 23 \\ 57 & 7 & 57 \end{bmatrix}$$

$$\begin{aligned} & 1 * 11 + 2 * 6 \\ = \; & 11 + 12 \\ = \; & 23 \end{aligned}$$

- The element in the *i*th row and *j*th column of the product is the sum of the element-wise products of
  - The elements of the *i*th row of the left multiplicand and
  - the elements of the *j*th column of the right multiplicand.
- This requires that the number of columns of the left multiplicand must be the same as the number of rows of the right multiplicand.

Matrix multiplication

$$\begin{bmatrix} 1 & 2 \\ \boxed{3} & \boxed{4} \end{bmatrix} \cdot \begin{bmatrix} \boxed{7} & 5 & 11 \\ \boxed{9} & -2 & 6 \end{bmatrix} = \begin{bmatrix} 25 & 1 & 23 \\ 57 & 7 & 57 \end{bmatrix}$$

$$\begin{array}{rrcr} & 3*7 & + & 4*9 \\ = & 21 & + & 36 \\ = & 57 & & \end{array}$$

- The element in the *i*th row and *j*th column of the product is the sum of the element-wise products of
  - The elements of the *i*th row of the left multiplicand and
  - the elements of the *j*th column of the right multiplicand.
- This requires that the number of columns of the left multiplicand must be the same as the number of rows of the right multiplicand.

Matrix multiplication

$$\begin{bmatrix} 1 & 2 \\ \colorbox{#b0b0ff}{3 \quad 4} \end{bmatrix} \cdot \begin{bmatrix} 7 & \colorbox{#ffb0b0}{5} & 11 \\ 9 & \colorbox{#ffb0b0}{-2} & 6 \end{bmatrix} = \begin{bmatrix} 25 & 1 & 23 \\ 57 & \color{magenta}{7} & \color{gray}{57} \end{bmatrix}$$

$$\begin{array}{lcl} & 3 * 5 & + & 4 * (-2) \\ = & 15 & + & -8 \\ = & \color{magenta}{7} & & \end{array}$$

- The element in the *i*th row and *j*th column of the product is the sum of the element-wise products of
  - The elements of the *i*th row of the left multiplicand and
  - the elements of the *j*th column of the right multiplicand.
- This requires that the number of columns of the left multiplicand must be the same as the number of rows of the right multiplicand.

Matrix multiplication

$$\begin{bmatrix} 1 & 2 \\ \colorbox{#b3b3ff}{3 \quad 4} \end{bmatrix} \cdot \begin{bmatrix} 7 & 5 & \colorbox{#ffb3b3}{11} \\ 9 & -2 & \colorbox{#ffb3b3}{6} \end{bmatrix} = \begin{bmatrix} 25 & 1 & 23 \\ 57 & 7 & \color{magenta}{57} \end{bmatrix}$$

$$\begin{array}{llll} & 3 * 11 & + & 4 * 6 \\ = & 33 & + & 24 \\ = & \color{magenta}{57} & & \end{array}$$

- The element in the i^{th} row and j^{th} column of the product is the sum of the element-wise products of
  - The elements of the i^{th} row of the left multiplicand and
  - the elements of the j^{th} column of the right multiplicand.
- This requires that the number of columns of the left multiplicand must be the same as the number of rows of the right multiplicand.

Matrix multiplication

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 7 & 5 & 11 \\ 9 & -2 & 6 \end{bmatrix} = \begin{bmatrix} 25 & 1 & 23 \\ 57 & 7 & 57 \end{bmatrix}$$

- The element in the *i*th row and *j*th column of the product is the sum of the element-wise products of
  - The elements of the *i*th row of the left multiplicand and
  - the elements of the *j*th column of the right multiplicand.
- This requires that the number of columns of the left multiplicand must be the same as the number of rows of the right multiplicand.

Matrix Multiply in Erlang

- If $P = A * B$, then $P(i,j)$ is the inner-product of the i^{th} row of A with the j^{th} column of B.
- We represent a matrix as a list of rows (see Slide 34).
- Let $B^T = \mathrm{transpose(B)}$.
- $P(i,j)$ is the dot-product of the i^{th} row of A with the j^{th} row of B^T.
- Code sketch

```
for i in 1 ...NRows(A) {
    let r = row i of A,
    for j in 1 ...NRows(B^T) {
        let c = row j of B^T,
        let P(i,j) = dot_prod(r,c)
    }
}
```

Matrix Multiply in Erlang

```
mult(A, B) -> mult_rows(A, transpose(B)).
% mult_rows: for each row of A ...
mult_rows([], _BT) -> [];
mult_rows([HA | TA], BT) ->
   [mult_cols(HA, BT) | mult_rows(TA, BT)].
% mult_cols: for each row of B^T ...
mult_cols(_HA, []) -> [];
mult_cols(HA, [HBT | TBT]) ->
   [ dot_prod(HA, HBT) | mult_cols(HA, TBT)].
% dot_prod: compute the dot-product of two, equal-length lists.
dot_prod([], []) -> 0;
dot_prod([H1 | T1], [H2 | T2]) ->
   H1*H2 + dot_prod(T1, T2).
```

Matrix Multiply – Observations

- The erlang code on slide slide 37 seems way messier than the pseudo-code on slide 36.
- Can we do better?
  - Of course. ☺
  - We'll show how erlang has functions for common code patterns (hyperslideslides:list-ops.simple – slide 55).
  - Then, we'll write a "prettier" implementation of matrix-multiply on slide 56.
- First, I'll show some execution examples to show how the code from slide 37 works.

dot_prod – execution example

% dot_prod: compute the dot-product of two, equal-length lists.
```
[1]   dot_prod([], []) -> 0;
[2]   dot_prod([H1|T1], [H2|T2]) -> H1*H2 + dot_prod(T1, T2).
```

```
dot_prod([3, 4, 5], [2, 7, 11])          % matches [2]
                                         %    H1=3, T1=[4,5]
                                         %    H2=2, T2=[7,11]
  -> (3*2)+dot_prod([4, 5], [7, 11])     % matches [2]
  -> 6+((4*7)+dot_prod([5], [11]))       % matches [2]
  -> 6+(28+((5*11)+dot_prod([], [])))    % matches [1]
  -> 6+(28+(55+0))                       % now, do the arithmetic
  -> 89                        % Done: [3,4,5] · [2,7,11] = 89
```

mult_cols – execution example

```
[1]   mult_cols(_HA, []) -> [];
[2]   mult_cols(HA, [HBT | TBT]) ->
         [ dot_prod(HA, HBT) | mult_cols(HA, TBT)].
```

```
mult_cols([1,2], [[7,9], [5,-2], [11,6]])         % matches [2]
  ->[dot_prod([1,2], [7,9]) | mult_cols([1,2], [[5,-2], [11,6]])]
      % dot_prod([1,2], [7,9]) = 25
  ->[25 | mult_cols([1,2], [[5,-2], [11,6]])]      % matches [2]
  ->[25 | [dot_prod([1,2], [5,-2]) | mult_cols([1,2], [[11,6]])]]
      % dot_prod([1,2], [5,-2]) = 1
  ->[25 | [1 | mult_cols([1,2], [[11,6]])]]        % matches [2]
  ->[25 | [1 | [dot_prod([1,2], [11,6]) | mult_cols([1,2], [[]])]]]
      % dot_prod([1,2], [11,6]) = 23
  ->[25 | [1 | [23 | mult_cols([1,2], [[]])]]]     % matches [1]
  ->[25 | [1 | [23 | []]]]
```

mult_cols – execution example

```
[1]   mult_cols(_HA, []) -> [];
[2]   mult_cols(HA, [HBT | TBT]) ->
         [ dot_prod(HA, HBT) | mult_cols(HA, TBT)].
```

```
mult_cols([1,2], [[7,9], [5,-2], [11,6]])
      % shown above
  ->[25 | [1 | [23 | []]]]  % [23 | []] = [23]
  ->[25 | [1 | [23]]]       % [1 | [23]] = [1, 23]
  ->[25 | [1 , 23]]         % [25 | [1, 23]] = [25, 1, 23]
  ->[25, 1 , 23]            % Done
```

mult_rows – execution example

```
[1]  mult_rows([], _BT) -> [];
[2]  mult_rows([HA | TA], BT) ->
       [mult_cols(HA, BT) | mult_rows(TA, BT)].
```

Let `BT = [[7,9], [5,-2], [11,6]].`

```
mult_rows([[1,2], [3,4]], BT)                        % matches [2]
  ->[mult_cols([1,2],BT) | mult_rows([[3,4]],BT)]
     % mult_cols([1,2],BT) = [25,1,23]
  ->[[25,1,23]|mult_rows([[3,4]],BT)                 % matches [2]
  ->[[25,1,23]|[mult_cols([3,4],BT)|mult_rows([],BT)]]
     % mult_cols([3,4],BT) = [57,7,57]
  ->[[25,1,23]|[[57,7,57]|mult_rows([],BT)]]         % matches [1]
  ->[[25,1,23]|[[57,7,57]|[]]]   % [[57,7,57]|[]]=[[57,7,57]]
  ->[[25,1,23]|[[57,7,57]]]
  ->[[25,1,23],[57,7,57]]                            % Done
```

Useful list operations

From module lists:

- `lists:sum(L) ->` the sum of the elements of `L`.

  ```
  79> lists:sum([1,3,5,7]).
  16
  ```

- `lists:nth(N,L) ->` the `N`th element of `L`.

  ```
  80> lists:nth(3, [1,3,5,7]).
  5 % Erlang indices start at 1.
  ```

- `lists:zip(L1, L2) -> L12.`
  `nth(N, L12) = {nth(N, L1), nth(N, L2)}.`
  Make tuples from corresponding elements of `L1` and `L2`.
  `L1` and `L2` must be of the same length.

  ```
  81> lists:zip([1, 2, 3, 4, 5, 6], [1, 4, 9, 16, 25, 36]).
  [{1,1},{2,4},{3,9},{4,16},{5,25},{6,36}]
  ```

- `lists:unzip(L12) -> {L1, L2}.`
  The inverse of `zip` – converts a list of two-element tuples into a tuple of two lists.

  ```
  82> lists:unzip([{1,1},{2,4},{3, 9},{4,16},{5,25},{6,36}]).
  {[1, 2, 3, 4, 5, 6], [1, 4, 9, 16, 25, 36]}
  ```

Example, another implementation of `dot_prod`

- Strategy:
  - Apply `lists:zip` to lists `V1` an `V2`.
  - Write a helper function, `dp`, to compute the product of each pair in the zipped list.
  - Use `lists:sum` to compute the total.
- In Erlang:

```
dot_prod(V1, V2) -> lists:sum(dp(lists:zip(V1, V2))).
dp([]) -> [];
dp([{X,Y} | T]) -> [X*Y | dp(T)].
```

- Example, let `V1` = `[3,4,5]` and `V2` = `[2,7,11]`.

```
dot_prod(V1, V2)
   -> lists:sum(dp(lists:zip([3,4,5], [2,7,11])))
   -> lists:sum(dp([{3,2}, {4,7}, {5,11}]))
   -> lists:sum([(3*2) | dp([{4,7}, {5,11}])])
   -> lists:sum([6 | [4*7 | dp([{5,11}])]])
   -> lists:sum([6 | [28 | [(5*11) | dp([])]]])
   -> lists:sum([6 | [28 | [55 | []]]])
   -> lists:sum([6 | [28 | [55]]])
   -> lists:sum([6 | [28, 55]])
   -> lists:sum([6, 28, 55])
   -> 89
```

The map pattern

- Many of our examples have the pattern[1]

```
f([]) -> [];
f([Head | Tail]) -> [ g(Head) | f(Tail)].
```

- Can we encapsulate this pattern as a function?
- Yes!

```
map(G, []) -> [];
map(G, [Head | Tail]) -> [ G(Head) | map(G, Tail)].
```

  - Note that the parameter G to map is a function.
  - We need a way to write an expression whose value is a function.

[1] As in "Design Patterns" – see Gamma et al.'s book.

`fun` expressions

- Syntax:

  `fun(`*ArgList*`) ->` *Expression* `end`

  where
  - *ArgList* – the arguments to the function.
  - *Expression* – evaluating this function produces the value for the function.
- Example:

```
83> Add1 = fun(X) -> X+1 end.
#Fun<erl_eval.6.80247286>
84> Add1(2).
3
85>
```

- More elaborate forms are possible. See Erlang Language Reference – Expressions→Fun Expressions

fun with map

```
map(Add1, [1, 4, 9, 16, 25, 36])
    -> [Add1(1), Add1(4), Add1(9), Add1(16), Add1(25), Add1(36)]
    -> [1+1, 4+1, 9+1, 16+1, 25+1, 36+1]
    -> [2, 5, 10, 17, 26, 37]
```

dot_prod: version 3

```
dot_prod(V1, V2) ->
 lists:sum(lists:map(fun({X,Y}) -> X*Y end, lists:zip(V1, V2))).
```

- `lists:map` is the same as the `map` function described on slide 44.
- `map` is an example of a higher-order function:
  - It takes a function as an argument.
  - Higher order functions can also produce functions as a result.
  - Or both – functions as arguments and as the result.
- Higher-order functions allow us to encapsulate common patterns of computation.
  - This is a lot of what gives functional programming its expressiveness.
  - Of course, you can do the same things in C, C++, Java, or other languages.
  - But, the expression of higher-order functions is generally more direct and concise in functional languages.

dot_prod – execution example

```
dot_prod(V1, V2) ->
 lists:sum(lists:map(fun({X,Y}) -> X*Y end, lists:zip(V1, V2))).
```

```
dot_prod([3, 4, 5], [2, 7, 11])
  -> lists:sum(lists:map(fun({X,Y}) -> X*Y end,
                         lists:zip([3,4,5], [2,7,11])))
  -> lists:sum(lists:map(fun({X,Y}) -> X*Y end,
                         [{2,3}, {4,7}, {5,11}]))
  -> lists:sum([ (fun({X,Y}) -> X*Y end)({2,3}),
                 (fun({X,Y}) -> X*Y end)({4,7}),
                 (fun({X,Y}) -> X*Y end)({5,11})
               ])
  -> lists:sum([(2*3), (4*7), (5*11)])
  -> lists:sum([6, 28, 55])
  -> 89.
```

`foldl` – combine elements of a list

`lists:foldl(Fun, Acc0, List) -> Total`

“Accumulate” the values in `List`.

```
foldl(Fun, Acc0, [X1, X2, ..., XN]) ->
   Fun(XN, Fun(..., Fun(X2, Fun(X1, Acc0))...)).
```

Example:

```
lists:foldl(fun(N, Prod) -> N*Prod end, 1, [1, 2, 3, 4, 5])
  -> (fun(N, Prod) -> N*Prod end)(5,
       (fun(N, Prod) -> N*Prod end)(4,
         (fun(N, Prod) -> N*Prod end)(3,
           (fun(N, Prod) -> N*Prod end)(2,
             (fun(N, Prod) -> N*Prod end)(1,1)))))
  -> (fun(N, Prod) -> N*Prod end)(5,
       (fun(N, Prod) -> N*Prod end)(4,
         (fun(N, Prod) -> N*Prod end)(3,
           (fun(N, Prod) -> N*Prod end)(2,(1*1)))))
  -> (fun(N, Prod) -> N*Prod end)(5,
       (fun(N, Prod) -> N*Prod end)(4,
         (fun(N, Prod) -> N*Prod end)(3,2)))
```

foldl – combine elements of a list

Example:

```
lists:foldl(fun(N, Prod) -> N*Prod end, 1, [1,2,3,4,5])
  -> (fun(N, Prod) -> N*Prod end)(5,
       (fun(N, Prod) -> N*Prod end)(4,
         (fun(N, Prod) -> N*Prod end)(3,
           (fun(N, Prod) -> N*Prod end)(2,
             (fun(N, Prod) -> N*Prod end)(1,1)))))
  -> (fun(N, Prod) -> N*Prod end)(5,
       (fun(N, Prod) -> N*Prod end)(4,
         (fun(N, Prod) -> N*Prod end)(3,
           (fun(N, Prod) -> N*Prod end)(2,(1*1)))))
  -> (fun(N, Prod) -> N*Prod end)(5,
       (fun(N, Prod) -> N*Prod end)(4,
         (fun(N, Prod) -> N*Prod end)(3,2)))
  -> (fun(N, Prod) -> N*Prod end)(5,
       (fun(N, Prod) -> N*Prod end)(4,6))
  -> (fun(N, Prod) -> N*Prod end)(5,24)
  -> 120.
```

foldr, mapfoldl, and mapfoldr

- `lists:foldr(Fun, Acc0, List) -> ...`
  Like `foldl` but works from the last element of the list back to the first.
- `lists:mapfoldl(Fun, Acc0, List1) -> {List2, Total}`
  Combines the functionality of `Map` and `foldl`. `Fun` takes two arguments. `Fun(Elem, AccIn) -> {NewElem, AccOut}`, where `Elem` is an element of the list, and `AccIn` is the accumulated result so far. `NewElem` is the value for the result list corresponding to `Elem`, and `AccOut` is the accumulated result after processing `Elem`.
  Example:

  ```
  85> lists:mapfoldl(fun(X, Sum) -> {1/X, Sum + (1/X)} end,
                     0, [1, 2, 3, 4, 5])
  {[1.0, 0.5, 0.3333, 0.25, 0.2], 2.2833}
  ```
- `lists:mapfoldr(Fun, Acc0, List1) -> {List2, Total}` is like `lists:mapfoldl(Fun, Acc0, List1)` but works from the last element of the list back to the first.

`all` and `any`

- `lists:all(Pred, List) -> bool().`
  Returns `true` iff the `Pred` evaluates to `true` for every element of `List`.

```
86> lists:all(fun(X) -> X >   0 end, [1, 4, 9, 16, 25, 36]
true
87> lists:all(fun(X) -> X >  10 end, [1, 4, 9, 16, 25, 36]
false
88> lists:all(fun(X) -> X > 100 end, [1, 4, 9, 16, 25, 36]
false
```

- `lists:any(Pred, List) -> bool().`
  Returns `true` iff the `Pred` evaluates to `true` for at least one element of `List`.

```
89> lists:any(fun(X) -> X >   0 end, [1, 4, 9, 16, 25, 36]
true
90> lists:any(fun(X) -> X >  10 end, [1, 4, 9, 16, 25, 36]
true
91> lists:any(fun(X) -> X > 100 end, [1, 4, 9, 16, 25, 36]
false
```

seq and split

- `lists:seq(N1, N2) -> List.`
  Produces the list `[N1, N1+1, ..., N2]`. `N1` and `N2` must be integers, and `N2` must be greater than or equal to `N1-1`.

```
92> lists:seq(1, 10).
[1,2,3,4,5,6,7,8,9,10]
93> Fac = fun(N) ->
        lists:foldl(fun(M, Prod) -> M*Prod end,
                    lists:seq(1, N)
        ) end.
#Fun<erl_eval.6.80247286>
94> Fac(5).
120
```

- `lists:split(N, L) -> {L1, L2}.`
  `L1` is the first `N` elements of `L`, and `L2` is the rest.

```
95> lists:split(2, [1, 4, 9, 16, 25, 36]).
{[1, 4], [9, 16, 25, 36]}
```

List operations – examples

```
96> L1_10 = lists:seq(1, 10).
[1,2,3,4,5,6,7,8,9,10]
97> R10 = lists:map(
            fun(_) -> random:uniform() end, L1_10).
[0.0923,0.4436,0.7230,0.9458,0.5015,
 0.3113,0.5974,0.9157,0.6670,0.4771]
98> math:sqrt(lists:foldl(
      fun(X,Sum) -> Sum + X*X end, 0.0, R10)).
1.9593126739777107
99>
```

- `random:uniform()` returns a pseudo-random float.
- `math:sqrt(X)` is floating point square-root.
- To save space, I rounded the values printed by Erlang.

Block Expressions

- I'd like to show how I can use all these functions from `lists` to write a shorter, clearer version of matrix multiply.
- But, I need to use a comma.
- What's a comma?
  - Erlang has block expressions.
  - A block expression is a list of expressions separated by commas.
  - The expressions are evaluated in program-text order.
  - The value of the block expression is the value of the last expression in the block.
- Why use a comma?
  - In function bodies, it's often helpful to be able to declare variables for intermediate results.
  - These declarations are expressions.
  - Thus the function body needs more than one expression.
  - Block expressions let us write such function bodies.

Handy Hint

- Sometimes, when using Erlang interactively, we want to declare a variable where it 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

```
99> FAC5 = m1:fac(5).
120.
100> FAC1000 = m1:fac(1000), ok.
ok
101>
```

BTW, it took about 1.2ms to compute 1000!.

Matrix Multiply, version 2

```
mult(A, B) ->
   BT = transpose(B),
   lists:map(
      fun(RA) ->
         lists:map(
            fun(CB) -> dot_prod(RA, CB) end, BT)
      end, A).

dot_prod(V1, V2) ->
   lists:foldl(
      fun({X,Y},Sum) -> Sum + X*Y end,
      0, lists:zip(V1, V2)).
```

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.
  - An 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`

Processes – Overview

- The built-in function spawn creates a new process.
- Each process has a process-id, pid.
  - The built-in function self() returns the pid of the calling process.
  - `spawn` returns the pid of the process that it creates.
  - The simplest form is `spawn(Fun)`.
    - ★ A new process is created.
    - ★ The function `Fun` is invoked with no arguments in that process.
- Sending a message.
  - `Pid ! Message`
    sends `Message` to the process with pid `Pid`.
  - Message is any Erlang term (i.e. an arbitrary expression).
- Receiving messages:
  See next slide.

Receiving Messages (short version)

```
receive
    Pattern1 -> Expr1;
    Pattern2 -> Expr2;
    ...
    PatternN -> ExprN
end
```

- If there is a pending message for this process that matches one of the patterns,
  - The message is delivered, and the value of the `receive` expression is the value of the corresponding *Expr*.
  - Otherwise, the process blocks until such a message is received.

A simple example

```
101> MyPid = self().
<0.152.0>
102> spawn(fun() -> MyPid ! "hello world" end).
<0.164.0>
103> receive Msg1 -> Msg1 end.
"hello, world"
```

Message Ordering

- Let `Process1` and `Process2` be two processes.
- If `Process1` sends messages `Msg1` and `Msg2` to `Process2` in that order,
  - and `Process2` executes a `receive` with a pattern that matches both messages
  - and no other pattern of the receive matches either message,
  - then `Msg1` will be delivered before `Msg2`.
- No other ordering is guaranteed.
- In particular, the triangle inequality is not guaranteed:
  - `Process1` can send `Msg12` to `Process2` and then send `Msg13` to `Process3`.
  - `Process3` can receive `Msg13` from `Process1` and then send `Msg32` to `Process2`.
  - `Process2` can receive message `Msg32` before it receives message `Msg12`.
- Simple rule: messages can arrive in any order with the exception that two messages from the same sender to the same receiver will be delivered in order.

Messages and Pattern Matching

- Erlang makes extensive use of messages.
  - So, it's a good idea to use pattern matching to make sure that the message that you receive is the one that you wanted.
  - Example (based on the September 6 lecture (slide 21)):

```
count3s(L0, N0, NProcs, MyPid) -> % > 1 processor.
    ...
    spawn(fun() ->
       MyPid ! {count3s, count3s:count3s(L1)} end),
    C2 = count3s(L2, N2, NProcs-1, MyPid),
    receive {count3s, C1} -> C1 + C2 end.
```

    - ★ The message of the child gets delivered to us because it is sent to `MyPid`.
    - ★ The `receive` gets a message that is the number of 3's in a sublist because it is tagged with `count3s`.

Receive and Time Outs

- The final alternative of a `receive` can be a time-out (in milliseconds):

```
receive
    Pattern2 -> Expr2;
    ...
    PatternN -> ExprN
    after TimeOut -> ExprTimeOut
end
```

- There are two special values for `TimeOut`:
  - `0` – the time-out is taken immediately if there are no pending messages that match one of the patterns.
  - `infinity` – the time-out is never taken.
- Time-outs should be used carefully:
  - They don't work well with changes in processor or network technology.
- Time-outs are handy for debugging (see next slide).

Debugging with Time-Outs (part 1)

- Consider:

```
count3s(L0, N0, NProcs, MyPid) -> % > 1 processor.
   ...
   spawn(fun() -> MyPid !
      {cuont3s, count3s:count3s(L1)} end),
   C2 = count3s(L2, N2, NProcs-1, MyPid),
   receive {count3s, C1} -> C1 + C2 end.
```

- Now, try running it:

```
   104> count3s_p1:time_it(1000).
   % hangs "forever"
   ∧G
   User switch command
    --> i
    --> c
   105>
```

- What went wrong?
  - If we do some debugging, we'll find that the `receive` statement is hanging.

Debugging with Time-Outs (part2)

- Add a time-out

```
count3s(L0, N0, NProcs, MyPid) -> % > 1 processor.
   ...
   spawn(fun() -> MyPid !
      {cuont3s, count3s:count3s(L1)} end),
   C2 = count3s(L2, N2, NProcs-1, MyPid),
   receive
      {count3s, C1} -> C1 + C2
      after 500 -> msg_dump()
   end.
msg_dump() ->
   io:format("time-out on receive~n"),
   msg_dump2().
```

Debugging with Time-Outs (part3)

- The rest of the code

```
msg_dump2() ->
    receive
        X -> io:format("~w~n", [X]),
              msg_dump2()
        after 0 -> 'time out for receive'
    end.
```

- Now, try running it:

```
105> count3s_p1:time_it(1000).
time out for receive
cuont3s, 14 % bug found!
cuont3s, 14 % 'cuont3s' is misspelled
...
{'time out for receive',3.5063929999999996}
106>
```

Some features we missed

- `if` expressions
- `case` expressions
- List comprehensions – also see the examples at programming examples (from erlang.org).
- exception handling
- strings
- bit strings and binaries
- records
- edoc – documentation generator (similar to javadoc).