# Sequential Erlang

Mark Greenstreet

CpSc 418 – Sept. 10, 2013

Outline:


- Why functional programming?
- Sequential Erlang
- A bit of processes and communication

# Objectives

- Learn/review key concepts of functional programming:
  - Referential transparency.
  - Higher-order functions can encode common programming patterns.
- Introduction to Erlang (mostly sequential)
  - Program design by structural decomposition.
  - Pattern matching: works naturally with structural decomposition.

# 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 thinking 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.

# How Erlang Helps

- Erlang uses message passing
  - Interactions between processes are under explicit control of the programmer.
  - Fewer races, synchronization errors, etc.
- Erlang is functional
  - Programming is declarative: it's more like math
    Imperative programs (e.g. Java, C, Python, etc.) are more like recipes.
  - A comment I've often heard from programmers in industry (Google, Intel, Microsoft, Oracle, . . . ): *"With sequential programming, assertional reasoning (invariants, pre- and post-conditions) are optional for most code. With parallel code, you must think assertionally."*
- 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.

# 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, 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.

# 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.
- Side effects are even more insidious in parallel programs.

# 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?
- Side effects are even more insidious in parallel programs.
  - You can see that a process 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.
  - Worse yet, you might have gotten there because of what some other process did,
  - And it might not happen the next 100 times you try because it depends on timing details.
  - Heisenbugs!!! ☹

# 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.

# Life without 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).
  ```

- Common programming patterns are provided by higher-order functions:

  ```
  -import(lists, [map/2, sum/1, zip/2]).
  dotProd(Atl, Btl) ->
     sum(map(fun({X, Y}) -> X*Y end, zip(Atl, Btl))).
  ```

  - `{X, Y}` is an Erlang tuple.
  - More about tuples coming up on Slide 17.

# Definitions Instead of Recipes

- Functional code tends to describe what the result is rather than what the code does.
- Example 1:

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

  - The first line says that the dot-product of two, zero-element vectors is 0.
  - The second line says that the dot-product of two, $N$-element vectors is the product of the products of their first elements plus the dot product of the $(N-1)$-element vectors that correspond to the rest of each argument.

- Example 2: `dotProd(A, B) ->`

```
  sum(map(fun({X, Y}) -> X*Y end, zip(A, B))).
```

The dot product of two vectors is the sum of the pairwise products of their elements.

Source code at

http://www.ugrad.cs.ubc.ca/~cs418/2013-1/lecture/09.10/ex.erl.

# 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.

# Sorting: Erlang code

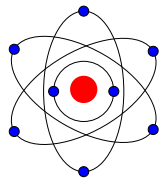
To be worked out in class.

# A bit more Erlang

The next few slides cover some frequently used Erlang features that we haven't covered yet:

- atoms
- tuples
- pattern matching

# 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.

# Tuples

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

```
1> T1 = {cat, dog, potoroo}.
{cat,dog,potoroo}
2> L6 = [ {cat, 17}, {dog, 42}, {potoroo, 8}].
[{cat,17}, {dog,42}, {potoroo,8}]
3> element(2, T1).
dog
4> T2 = setelement(2, T1, banana).
{cat,banana,potoroo}
5> T1.
{cat,dog,potoroo}
```

- On Slide 11 we used the function `lists:zip(A, B)`. This creates a list of two-element tuples from two lists of the same length.

```
6> lists:zip([cat, dog, potoroo], [1, 2, 3]).
[{cat, 1}, {dog, 2}, {potoroo, 3}].
```

# 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.).

# 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:

```
7> [Head | Tail] = [1,5,34].
[1,5,34]
8> Head.
1
9> Tail.
[5,34]
```

  - `Head` and `Tail` were unbound before executing command 7.
  - 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:

```
10> [1 | X1] = L1.
[1,5,34]
11> X1.
[5,34]
12> [A1, B1, 2*17] = L1. % The compiler replaces 17*2 with 34.
[1,5,34]
13> [1, 5, 2*C1] = L1.
* 1: illegal pattern % But it's not a general equation solver!
14> [_, B2, _] = L1.
[1,5,34]
15> B2.
5
```

# An example

Problem Statement: write a function `balanced(L)` that returns true iff:

- `L` is string consisting of parentheses (`(` and `)`), square brackets (`[` and `]`), and whitespace.
- Every `(` is paired with a subsequent ).
- Every `[` is paired with a subsequent ].
- Proper nesting.
- Examples:
  - `balanced("([][()[]])") -> true.`
  - `balanced(")(") -> false.`
  - `balanced("([)]") -> false.`
  - `balanced("()[]") -> true.`

# `balanced`: Design

Consider: `balanced(L)`

- What if `L` is a list?
  - What if `L` is the empty list?
  - What if `L` has the form `[Hd | Tl]`?
    - What if `Hd` is `$(` or `$]`?
    - What are `Hd` is a white space character?
    - What are `Hd` is a white space character?
- What if `L` is not a list?
- Hint: it may be simpler to write a helper function that can return values other than `true` and `false`, and then check the final result.

# `balanced`: Erlang code

To be worked out in class (time permitting).

# 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

```
16> MyPid = self().
<0.152.0>
17> spawn(fun() -> MyPid ! "hello world" end).
<0.164.0>
18> receive Msg1 -> Msg1 end.
"hello, world"
```

# Balanced parenthesis, with processes

- The plan:
  - The main process will open a file and send one character at a time to a child process.
  - The child will check for the balanced-parenthesis condition for the stream of characters that it receives.
  - When the child receives an ¡tt¿eof¡/tt¿ it sends ¡tt¿true¡/tt¿ or ¡tt¿false¿ back to the main process.
- Why?
  - An example of using processes.
  - This is “more efficient” that the first approach because we don’t have to hold the entire string in memory. This could matter if we had a really big input file.
  - It leads to a cool problem for the Sept. 12 mini-assignment.
- Source code at

  http://www.ugrad.cs.ubc.ca/~cs418/2013-1/lecture/09.10/ex.erl.

# Balanced parenthesis: the main process

```
% bFile(FileName) -> boolean
% Raises an error if FileName cannot be opened for reading.
% Read the contents of FileName and return true if it's a string
% with balanced and properly nested ()s and []s.
bFile(FileName) ->
    File = case file:open(FileName, read) of   % open the file
        {ok, F} -> F;
        {error, Reason} -> error(Reason)
    end,
    MyPid = self(),    % heed Socrates
    Child = spawn(fun() ->    % create the child process
        MyPid ! {balanced, FileName, (bproc() == true)} end),
    bScan(File, Child),    % read the file, send contents to Child.
    Ans = receive    % get the result from Child.
        {balanced, FileName, IsBalanced} -> IsBalanced
        after 5000 -> error("time out")
    end,
    file:close(File),    % clean up
    Ans.    % return the result
```

# Balanced parenthesis: reading the file

```
% bScan(File, Pid) -> ok
% Raises an error if read(File, 1) fails.
% Read characters from File one at a time and send them to Pid.
% On end-of-file, send the eof atom to Pid, and return.
bScan(File, Pid) ->
    X = file:read(File, 1),          % read a character
    case X of                        % what did we get?
        {ok, [Char]} ->              % got a character
            Pid ! Char,              % send it to Pid
            bScan(File, Pid);        % continue
        eof ->                       % end of file
            Pid ! eof;               % send it to Pid, and we're done
        {error, Reason} ->           % the read failed
            error(Reason)            % raise an error
    end,
    ok.
```

# Balanced parenthesis: checking for balance

```
% proc() -> true | false | NextChar
% Receive input characters. If we reach the end-of-input and all ()s and []s are
% balanced, return true. If we reach something that doesn't match, we return it as
% NextChar – it may be a right parenthesis or bracket for an enclosing expression.
bproc() ->
   receive
      $( -> bproc($));
      $[ -> bproc($]);
      eof -> true;
      X ->
         IsSpace = is_whitespace(X),
         if IsSpace -> bproc(); % ignore whitespace and continue
            true -> X
         end
   end.
% Look for a Right to match a left parenthesis or bracket.
bproc(Right) ->
   case bproc() of % skip balanced substrings and get next character
      Right -> bproc(); % It's what we wanted. Check the remaining input.
      _Else -> false % mismatch – give up and return false.
end.
```

# 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.
    - ★ Higher-order functions (e.g. `map`, and `foldl`) encode common software patterns.
- Sequential Erlang
  - Lists, tuple, atoms
  - Pattern matching
  - Using structural design for sorting, balanced parentheses.
- Quick intro to processes.
  - More on Thursday (Sept. 12).

# Preview

## September 12: Parallel programming with Erlang

Mini-Assignment: Mini-Assignment due before class

## September 17: Performance Loss

Reading: Lin & Snyder, chapter 3, pp. 61–68

## September 17: Performance Measurement

Homework: Homework 2 goes out – parallel programming with Erlang
Reading: Lin & Snyder, chapter 3, pp. 68–77
Homework: Homework 1 deadline for early-bird bonus

## September 24: Matrix Multiplication

Reading: Lin & Snyder, chapter 3, pp. 77–85
Homework: Homework 1 due

## September 26: Superscalars and compilers

Reading: The MIPS R10000 Superscalar Microprocessor (Yeager)

## October 1: Shared Memory Multiprocessors

Reading: Lin & Snyder, chapter 2, pp. 30–43.
Homework: Homework 3 goes out

## October 3: Message Passing Multiprocessors

# Review Questions

- What is referential transparency?
- Why don't functional languages have loops?
- Describe the `map` function.
- Describe the `foldl` function.
- How would you write a `pow(X, Y)` so that
  - If `X` and `Y` are both numbers (integers or floats), then `pow(X, Y)` returns `X` raised to the $Y^{th}$ power.
  - If `X` and `Y` are lists of the same length, then each element of `X` is raised to the power given by the corresponding element of `Y`.
  - If `X` is a list and `Y` is a number, each element of `X` is raised to the $Y^{th}$ power.
  - If `X` is a number and `Y` is a list, then the result is a list with `X` raised to each power given by `Y`.
- We'll cover processes more on Thursday.

# Extra 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.

- Message ordering in Erlang: what's guaranteed and what's not.
- Message and pattern matching.
- Using time-outs with messages.
- Using time-outs for debugging: how to avoid hanging the Erlang shell.
- Erlang's rules for punctuation: making sense of when to write comma, semicolon, end, and period.
- Suppressing verbose output when using the Erlang shell.
- Forgetting variable bindings (only in the Erlang shell).

# 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 5 lecture (slide 23)):

```
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:

```
19> count3s_p1:time_it(1000).
% hangs "forever"
^G
User switch command
 --> i
 --> c
20>
```

- 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:

```
20> count3s_p1:time_it(1000).
time out for receive
cuont3s, 14 % bug found!
cuont3s, 14 % 'cuont3s' is misspelled
...
{'time out for receive',3.5063929999999996}
21>
```

# 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`

# 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

```
21> L1_to_5 = lists:seq(1, 5).
[1, 2, 3, 4, 5].
22> L1_to_5M = lists:seq(1, 5000000), ok.
ok
23> length(L1_to_5M).
5000000
24>
```

# 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.

```
24> X = 2+3.  5.
25> X = 2*3.
** exception error:  no match of right hand side valu
26> f(X).
ok
27> X = 2*3.
6
28>
```