Erlang

Language Introduction

Ericsson **Lang**uage

Agner Krarup **Erlang**

What is Erlang?

- Language developed at Ericsson
- Core language is a simple dynamically-typed functional programming language
- Concurrent (light-weight processes belong to language, not OS)
- "Share nothing" process semantics
- Pure asynchronous message passing
- Transparent distribution of processes across machines
- Mechanisms for in-service code upgrade
- Large set of libraries (OTP)

from: http://portal.acm.org/citation.cfm?id=1238844.1238850

History of Erlang

- (mid-80s) Ericsson (Swedish telecom company) set out to find the best language for building the next generation of telecom systems

- Requirements
  - Handling a very large number of concurrent activities
  - Systems distributed over several computers
  - Actions to be performed at a certain point of time or within certain time
  - Very large software systems
  - Complex functionality such as feature interaction
  - Continuous operation over several years
    - Software maintenance (reconfiguration, etc.) without stopping the system
    - Stringent quality and reliability requirements
    - Fault tolerance both to hardware failures and software errors

from: http://portal.acm.org/citation.cfm?id=1238844.1238850

History of Erlang cont'd

- Initially **Smalltalk** was considered. Method calls in Smalltalk are "message sends." *Message passing figures prominently in Erlang's design.*

- Soon after, they discovered that the rules for how a telecom system should work could be elegantly expressed in the logic language **Prolog**. *The first Erlang interpreters were written in Prolog and the syntax retains the flavor of Prolog.*

History of Erlang cont'd

- Prolog is a **logic language** - a program consists of facts and queries over those facts are evaluated

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)      :- parent_child(Z, X),
                      parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).

?- sibling(sally, erica).
Yes
```

Facts

Query

http://en.wikipedia.org/wiki/Prolog

History of Erlang cont'd

- Prolog is a **logic language** - a program consists of facts and queries over those facts are evaluated

- It turns out that the logic features of Prolog were deemed unnecessary for telecom systems so they were removed from Erlang (leaving a functional language)

Query

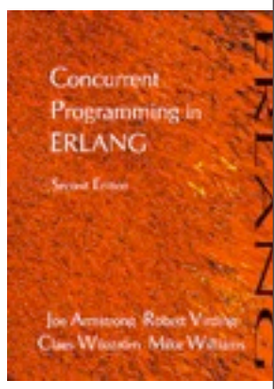
```
parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).

?- sibling(sally, erica).
Yes
```

http://en.wikipedia.org/wiki/Prolog

History of Erlang cont'd

- Other modifications from Prolog were the addition of concurrency and message passing. The syntax diverged over time.
- Erlang continued to evolve "organically", growing from 2 to "hundreds" of developers between 1989 and 1997.
- First Erlang book published in 1993, Ericsson attempted to commercialize Erlang.



- Banned within some groups at Ericsson in 1998, released as open source.

Real-World Erlang

- CouchDB - document store "NoSQL" database
- Facebook Chat
- RabbitMQ - owned by VMWare, message queue server
- ejabberd - XMPP (jabber chat) server
- Amazon SimpleDB
- GitHub
- Wings3d - open source 3d modeling tool
- ... and more: http://stackoverflow.com/questions/690875/real-world-applications-of-erlang

Why Erlang?

- Functional (like OCaml, Scheme, Haskell)
  - Program by evaluating functions to produce results, not by mutation of program state
  - A good match for concurrent programming since concurrent access to shared state (and the required mutual exclusion) are the source of many bugs (races, atomicity violations, deadlocks)

Why Erlang? cont'd

- Transparent distribution of processes across machines
- Efficient implementation of processes, message passing
- Robust error handling support
- Runtime updates

Installing Erlang

- Available on linuxlab.cs.umd.edu (R14B) - access using class account (see grades server for login info)
- Windows: http://erlang.org/download.html
- Mac: need to build from source
- Web REPL: http://www.tryerlang.org/

Erlang Resources

- Getting Started with Erlang (http://www.erlang.org/doc/getting_started/users_guide.html)
- Erlang Documentation (http://www.erlang.org/doc/)
- "learn you some Erlang for a great good" (http://learnyousomeerlang.com)
- Erlang (CACM) (http://cacm.acm.org/magazines/2010/9/98014-erlang/fulltext)
- Erlang for Concurrent Programming (ACM Queue) (http://queue.acm.org/detail.cfm?id=1454463)
- Books:
  - Programming Erlang: Software for a Concurrent World (Armstrong)
  - Erlang Programming (Cesarino, Thompson)

The Erlang Language

Erlang REPL

- Read-Eval-Print-Loop - allows immediate execution of Erlang code
- execute "`erl`" from the shell
- type "`q().`" to exit the REPL

```
> io:format("hello "),
  io:format("world\n").
```

Comma (,) separates terms within an expression

Period (.) ends an expression

Erlang Numerics

```
1> 42.
42
2> $A.
65
3> $\n.
10
4> 2#101.
5
5> 16#1f.
31
6> 2.3.
2.3
7> 2.3e3.
2.3e3
8> 2.3e-3.
0.0023
```

ASCII code for character

101 interpreted as binary

1F interpreted in base-16

scientific notation

http://www.erlang.org/doc/reference_manual/data_types.html#id63103

Erlang Arithmetic

```
1> +2.3.
2.3
2> -2.3.
-2.3
3> 12+2.3.
14.3
4> 12-4*3.
0
5> 10/3.
3.3333333333333335
6> 10 div 3.
3
7> 10 rem 3.
1
8> 2#11001 band 2#10011 == 2#10000 bor 2#00001.
true
```

unary operators

integer division/modulus

bitwise operators

http://www.erlang.org/doc/reference_manual/expressions.html#id72847

Atoms

```
> hello.
> phone_number.
> 'Monday'.
> 'phone number'.
> true.
> false.
> ok.
```

- Atoms are extremely useful - uses include places where you might use enumerations (in C) or internal string constants (e.g. hash table keys in Java)

http://www.erlang.org/doc/reference_manual/data_types.html#id61072

Lists

```
[1, two, 3]
```

Lists

```
[1, two, 3]
```

Heterogeneous

Lists

```
[1, two, 3]
```

```
[1 | [two, 3] ]
```

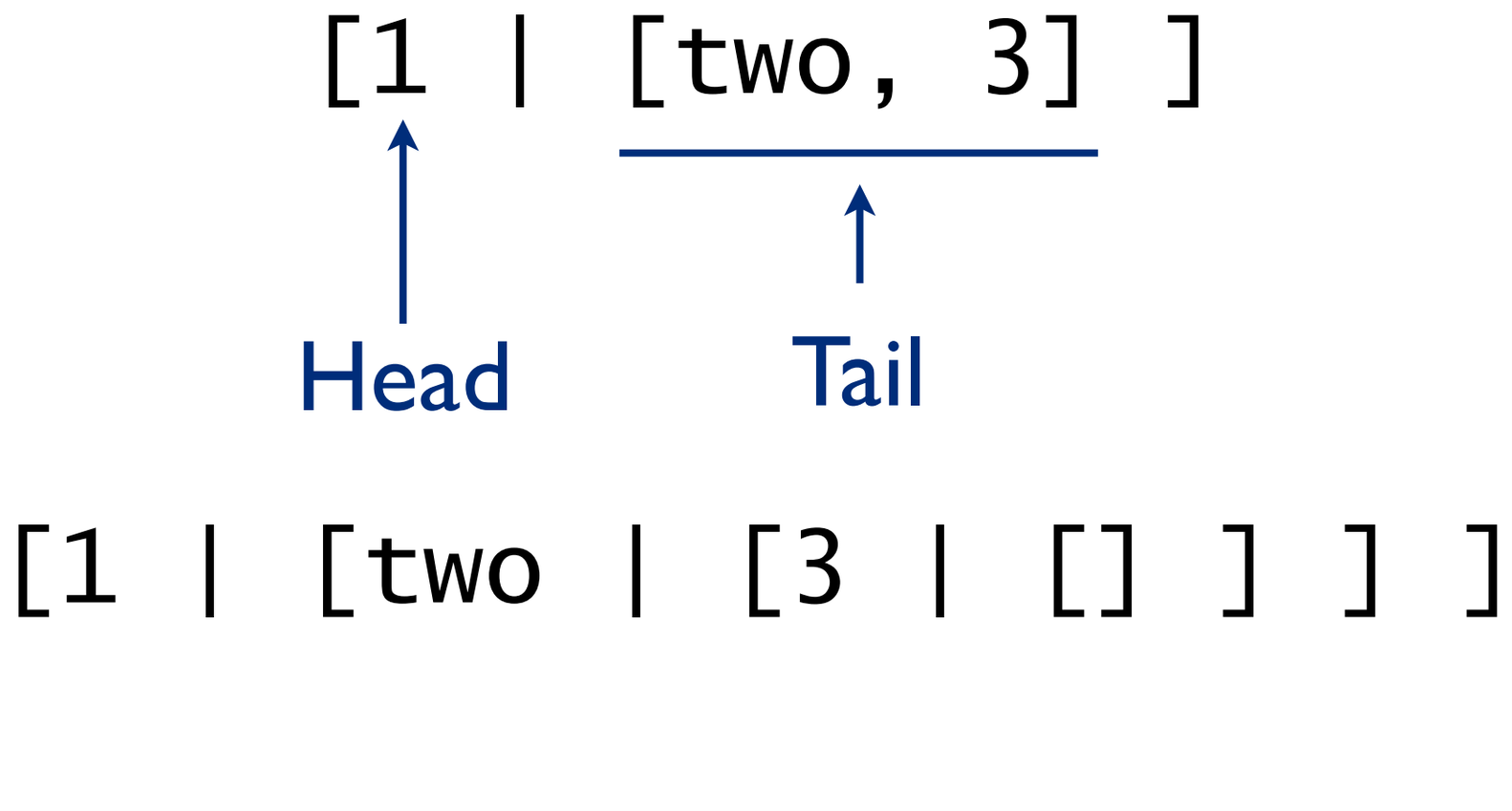
Lists

```
[1, two, 3]
```

```
[1 | [two, 3] ]
```

Lists

```
[1, two, 3]
```



Strings

```
> [72,101,108,108,111,32,87,111,114,108,100,10].
"Hello World\n"

> "XYZ" ++ "ABC".
"XYZABC"

> "FOODFOOD" -- "OO".
"FDFOOD"

> lists:prefix("Abra", "Abracadabra").
true

> string:tokens("we the people", " ").
["we","the","people"]
```

strings are lists of numbers

list operations apply to strings

can apply tools from **lists** and **string** modules

http://www.erlang.org/doc/man/string.html
http://www.erlang.org/doc/man/lists.html

Variable Binding / Pattern Matching

```
> X = 10.
10
> Y = hello.
hello
```

X and Y are initially *unbound*

```
> X.
10
> Y.
hello
```

now they are bound

```
> X = 20.
** exception error: no match of right hand side value 20
> X = 10.
10
> 10 = 10.
10
> X = X + 1.
** exception error: no match of right hand side value 11
```

Erlang does **not** provide mutable variables!

Pattern Matching

```
> [First | Rest] = [1, two, 3].
[1, two, 3]
> First.
1
> Rest.
[two, 3]
```

Underscore means *match anything*, don't create a binding

```
> [A | _] = [1, two, 3].
[1, two, 3]
> [First, two, Q, _] = [1, two, 3, hello].
[1, two, 3, hello]
> Q.
3
```

Patterns can contain a mix of bound variables, unbound variables, literals, and underscores.

Tuples

- use lists for cases where length may vary and/or it makes sense to iterate through the data
- use tuples for data where each position has a distinct meaning

```
>  {course, 433, enrollment, 30}.
{course, 433, enrollment, 30}.

> R = {course, 433, enrollment, 30}.
> R.
{course, 433, enrollment, 30}.

> {_, CourseNum, _, _} = {course, 433, enrollment, 30}.
> CourseNum.
433
```

Modules

- Erlang modules are placed in .erl files.

same as file name (minus .erl)

```
-module(foo).
```

exported functions (name/arity)

```
-export([somefun/0, otherfun/1]).
```

source module

```
-import(io, [format/1]).
```

imported functions (name/arity)

```
somefun() -> format("testing...\n").
```

no need to write "io:" due to import

```
otherfun(X) -> X.
```

Compilation

```
shell$ erlc file.erl

-or-

(erlang shell)
> c(file).
{ok,file}
```

produces **file.beam**
for the erlang virtual machine

run compiled erlang code from commandline

```
erl file.beam -noshell -s file entry_func -s init stop
```

compiled file module function to call then stop

Functions

fun name

args

```
area_circle(Radius) ->
        Pi = 3.14,
        RSquared = Radius * Radius,
        Pi * RSquared.
```

function body

pattern matching on arguments

```
area({circle, Radius}) -> 3.14*Radius*Radius;
area({square, Side})   -> Side * Side;
area({rect, Length, Width}) -> Length * Width.
```

Recursive Functions

```
list_sum([]) -> 0;

list_sum([Head | Rest]) ->

    Head + list_sum(Rest).
```

```
my_map(F, []) -> [];

my_map(F, [Head | Rest]) ->

    [F(Head) | my_map(F, Rest)].
```

Anonymous Functions

```
> fun(X) -> X*X end.
#Fun<erl_eval.6.13229925>

> lists:map(fun(X) -> X*X end, [1,2,3,4,5]).
[1,4,9,16,25]

> lists:filter(fun(X) -> X rem 2 == 1 end, [1,2,3,4,5]).
[1,3,5]
```

Conditions

```
if
     X > 5 -> "greater";
     X < 5 -> "less”;
     true -> “other”
end.
```

```
case X of
     king -> 13;
     queen -> 12;
     _ -> unknown
end.
```

List Comprehensions

```
[X * X || X <- [1,2,3,4]]

[{X,Y} || X <- [1,2,3,4],
          Y <- [1,2,3,4]]

[{X,Y} || X <- [1,2,3,4],
          Y <- [1,2,3,4],
          X>Y ]
```

Error Handling

- "**Fail-fast**" philosophy - don't handle every error, let the process fail (this will make more sense once we discuss multi-process servers)

```
> throw(some_error_value).
** exception throw: some_error_value
> erlang:error(bad_arith).
** exception error: bad_arith
> erlang:exit(failure).
** exception exit: failure
```

Error Handling cont'd

```
> catch throw(some_error_value).
some_error_value
> catch 25.
25
try Expr()
catch
    Throw -> {caught_throw, Throw};
    exit:Exit -> {caught_exit, Exit};
    error:Error -> {caught_error, Error}
end
```