Lecture 8

Linda and Erlang

Part I

Linda

Overview of Message Passing

- One process sends a message
- Another process awaits for a message

- We will consider two dimensions of this approach:
  - What form of synchronisation is required
  - What form of process naming is involved in message passing

Communication Coupling

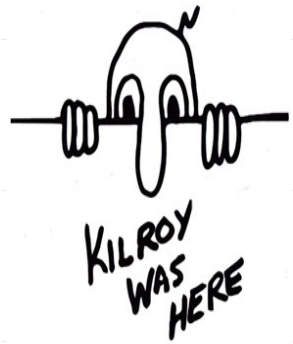
- Time
  - Synchronisation in message passing
    - Processes must usually exist simultaneously to communicate
      - Synchronous/rendezvous
      - Asynchronous
- Space
  - Naming
  - Global scope for the names
    - Processes, or
    - Channels/operations

Decoupling – Broadcast

- Communication by *broadcast* is decoupled
  - In space
    - Sender does not know who the receivers are, or
    - How many receivers there are, or
    - Whether there are any receivers at all
  - But not in time
    - Receivers must exist and act simultaneously with the sender

Decoupling – Blackboard

- Communication by *blackboard* is decoupled
  - In space
    - Writer does not know who the readers are, or
    - How many readers there are, or
    - Whether there are any readers at all
  - In time
    - Reader can even be created after the writer dies



The Linda Model

- Origins
  - 1983: David Gelernter
  - General idea
    - Parallel Programming =
    - Computation +
    - Coordination
- Based on the blackboard communication
  - Asynchronous communication
  - No naming – global “blackboard”
  - Content matching

The Linda Model

- Linda
  - Not a programming language, but
  - A collection of operations
    - Take your favorite (sequential) host language,
    - Add Linda operations
      - Library
      - Extend the language itself
    - You get a parallel/concurrent programming language

Blackboard – Tuple Space

- Linda shared memory
  - Blackboard – Tuple space
  - Data in the space: tuples
- Tuples
  - Typed sequences of data
    - Singletons, Pairs, Triples, Quadruples, …
  - Examples:
    - `(1,true)`, `(1,2)`, `(false,2)` differently typed
    - `("list",1,42)`

Patterns

- Patterns are used to find tuples
  - Tuples with variables
  - Examples
    - `("list", ?Index, ?Value)`
    - `(1, ?X)`
- Pattern matching examples
  - `(1, ?X)` matches `(1, true)` to give `X = true`
  - `(1, ?X)` fails to match `(2, true)`
  - `(1, ?X)` fails to match `(1, true, 3)`

Communication – Blocking

- `OUT(exp1,…,expn)`
  - Evaluate the expressions `exp1,…,expn` and produce a tuple T,
  - Atomically add the tuple T to the tuple space.
- `IN(P)`
  - Block until a tuple T in the tuple space matches the pattern P,
  - Atomically remove T from the tuple space,
  - Assign the variables in P to values in T according to the match.

Communication – Blocking

- `RD(P)`
  - Block until a tuple T in the tuple space matches the pattern P,
  - Assign the variables in P to values in T according to the match.
  - Like `IN(P)` except that the matched tuple T is left in the tuple space

Communication – Non-blocking

- Two non-blocking input/read variants
  - `INP(P)` and
  - `RDP(P)`
  - Predicates that return:
    - `true` and assign variables in `P` if there is a matching tuple
    - `false` otherwise

“Distributed” Data Structures

- To promote concurrency it is natural to use a “distributed” representation of data structures
- The tuple space is logically shared, but
- May be physically distributed.

- What do we mean by distributed representations of data-structures?

Example: List/Vector

- Create a three element list/vector

```
OUT("list",0     ,value0);
OUT("list",1     ,value1);
OUT("list",2     ,value2);
OUT("list","tail",3     );
```

- Add?

Example: List/Vector

- Create a three element list/vector

```
OUT("list",0     ,value0);
OUT("list",1     ,value1);
OUT("list",2     ,value2);
OUT("list","tail",3     );
```

- Add

```
void add(E element) {
   int Index;
   IN ("list","tail",?Index);
   OUT("list","tail",Index+1);
   OUT("list",Index ,element);
}
```

Example: Unbounded Buffer

- Create an empty buffer called “buff”

```
OUT("head", "buff", 0);
OUT("tail", "buff", 0);
```

- Buffer operation put

```
void put(String bufName, E element) {
   int Index;
   IN ("tail", bufName, ?Index);
   OUT("tail", bufName, Index+1);
   OUT("element", bufName, Index, element);
}
```

Example: Unbounded Buffer

- Buffer operation get

```
void E get(String bufName) {
   int Index;
   E Element;

   IN ("head", bufName, ?Index);
   OUT("head", bufName, Index+1);
   IN ("element", bufName, Index, ?Element);
   return Element;
}
```

Expressive Power

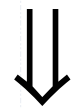
- Semaphores, monitors, message passing
  - Equally expressive
  - Any synchronisation with await statement
- Linda vs. XXX
  - Can we implement XXX?
  - Can XXX implement Linda?
    - Important theoretical question
    - An illustrative example, but not normal practice

Semaphores

Semaphore	Linda
`sem s = N`	`for(int x=0;x<N;x++)` `  OUT("s");`
`P(s);`	`IN("s");`
`V(s);`	`OUT("s");`

Expressive Power

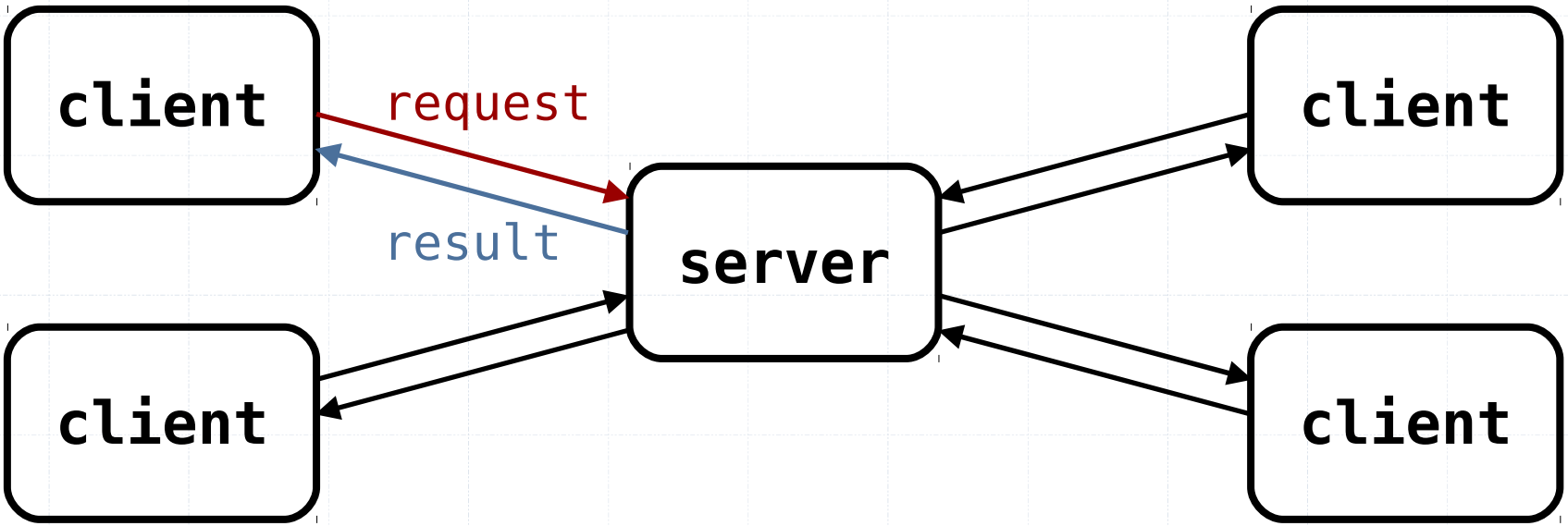
- Semaphores as Linda singletons
- Assignment 3 is for you to implement Linda tuple space using asynchronous message passing in Erlang



- The same expressive power
  - Can implement any await statement
  - Important theoretical result

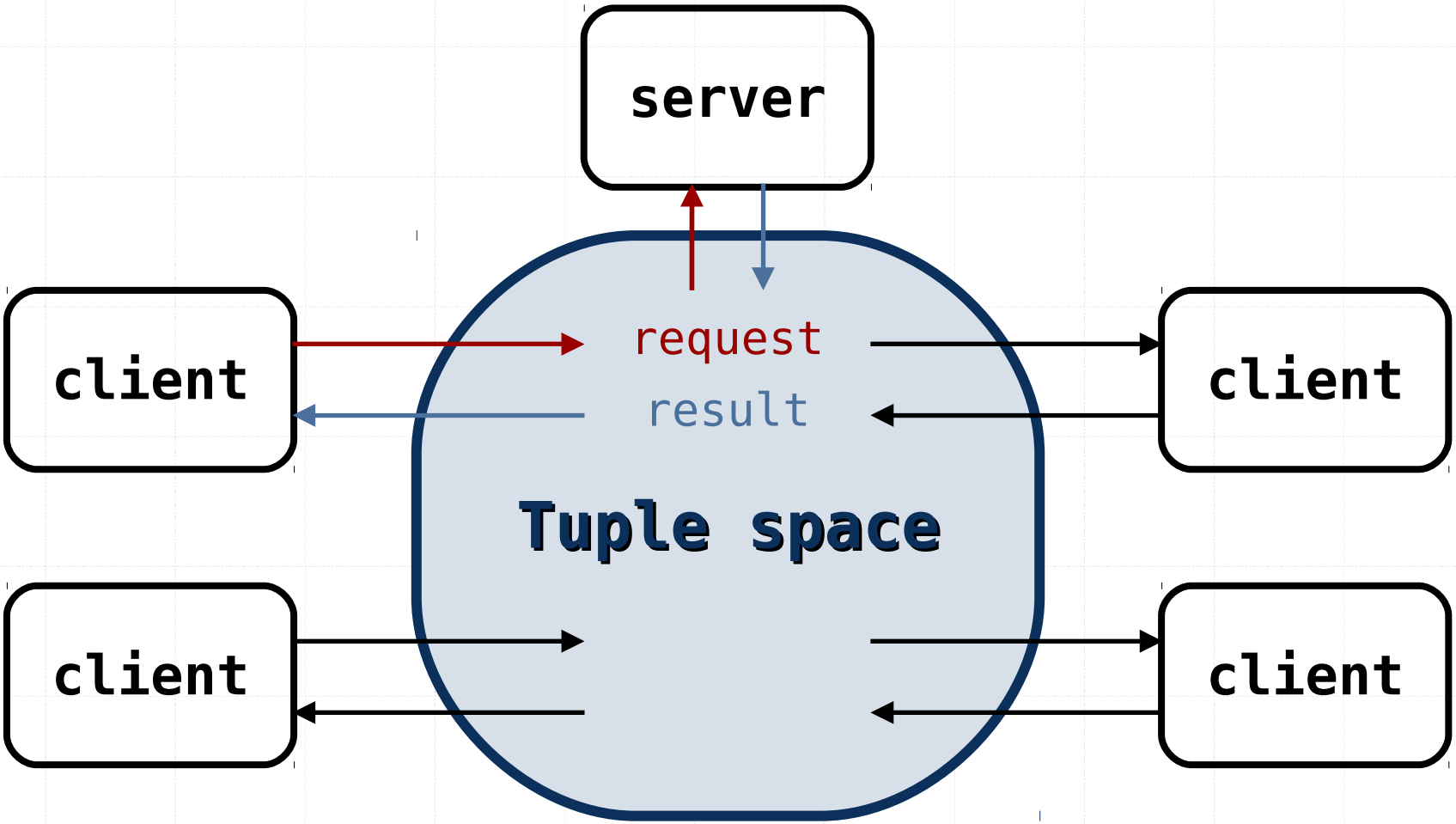
Client-Server Interaction

- Common asynchronous communication pattern
  - For example: a web server handles requests for web pages from clients (web browsers)



Linda's Client-Server

- Communication by "message content"



Replicated Worker Pattern

- A typical programming style used in the Linda model:
  - Break a problem into a collection of tasks
  - Start a team of (identical) Worker processes
  - Each Worker takes tasks and solves them
  - A Master process coordinates the activities

Example: NxN Matrix Multiplication

- Master process
  - Initialises the tuple space with rows of matrix A and columns of B,
  - Adds a “next job" counter, and
  - Collects the results.
- Worker processes
  - Grab a job number from 0 to N∗N-1
  - Get corresponding row + column to multiply
  - Place the resulting number in the tuple space

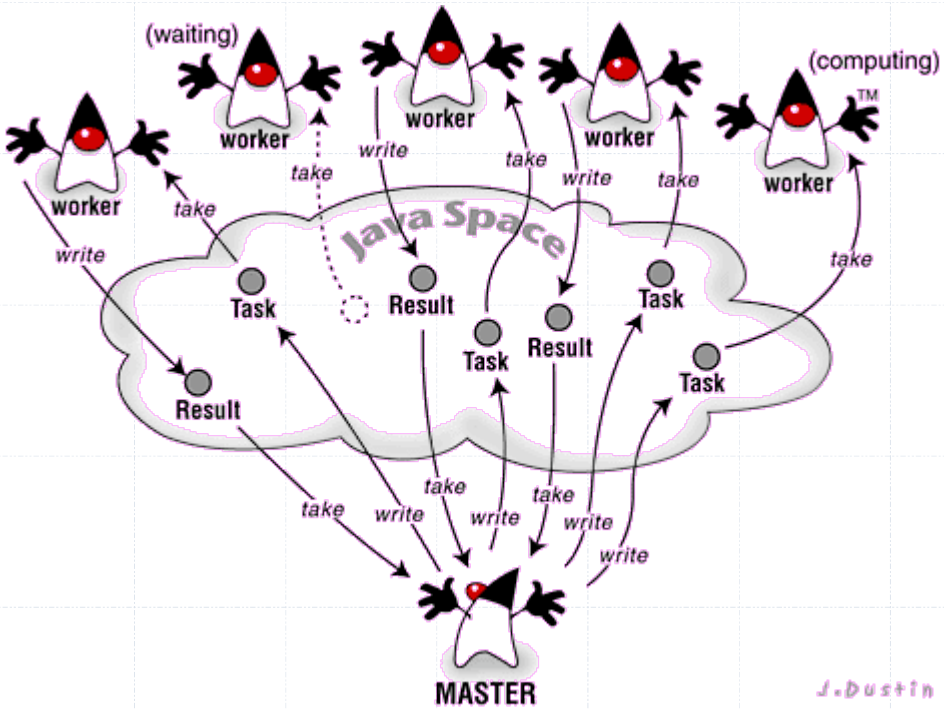
Master Multiplier

```
process master {
   final int N = 3;
   OUT("A", 0, {1,2,3});OUT("B", 0, {1,0,1});
   OUT("A", 1, {4,5,6});OUT("B", 1, {0,1,0});
   OUT("A", 2, {7,8,9});OUT("B", 2, {2,2,0});

   OUT("Next", 0);

   for(int i=0; i<N; i++)
      for(int j=0; j<N; j++)
         IN("C", i, j, ?C);
         //Do something with C(i,j)
      }
}
```

Worker Multiplier

```
process worker((int w=0; w<nWorkers; w++)) {
   int Element; int[] Vector1, Vector2;
   IN("Next", ?Element);
   OUT("Next", Element+1);
   while(Element < N*N) {
      int i = Element / N;
      int j = Element % N;
      RD("A", i, ?Vector1);
      RD("B", j, ?Vector2);
      int x = innerProduct(Vector1, Vector2);
      OUT("C", i, j, x);
      IN("Next", ?Element);
      OUT("Next", Element+1);
}}
```

JavaSpaces


(waiting)
worker
worker
worker
worker
worker
(computing)
TM
write
take
Java Space
Task
Result
Task
Result
Task
Task
Result
MASTER
J.Dustin

JavaSpaces

- Why tuple spaces in a Java context?
  - Simplified approach to distributed computation
  - Alternative middleware to the “standard” stuff
    - Remote Method Invocation (RMI)
    - Enterprise Java Beans (EJB)
    - Object Request Broker Architecture (CORBA)

Tuples = Entries

- The Tuple is replaced by the concept of an Entry in JavaSpaces
- An entry is an object which
  - Implements `net.jini.space.Entry`
  - Has objects as fields
- Pattern matching uses templates
- A template is a entry
  - Fields set to specified values, or
  - Wildcards (?X): fields set to `null`

Operations on a JavaSpace

- Standard operations
  - `write`: like `OUT`
  - `read`/`readIfExists`: like `RD`/`RDP`
  - `take`/`takeIfExists`: like `IN`/`INP`
- New operations
  - `notify`: requests a notification that an object matching a template is added to the space
    - It will cause the requester's notify method to be called when a matching tuple appears
- Transactions

Transactions

- A standard database concept
  - Collections of operations execute atomically,
  - Or not at all
- Example: adding to a list
  - Either complete the whole add or roll-back

```
void add(E element) {
   int Index;
   IN ("list","tail",?Index);
   OUT("list","tail",Index+1);
   OUT("list",Index ,element);
}
```

Conclusions – Linda

- Tuple spaces provide a very simple mechanism for parallel and distributed computation
  - Loose coupling
  - Flexibility
  - Extendibility
- Modern instances (examples)
  - JavaSpaces (Sun), and
  - TSpaces (IBM)

Conclusions – Linda

- Negatives?
  - Less structured communication
    - Data and structure intermixed
  - Implementation efficiency
    - Distributed space
    - Distributed pattern matching

Part II

Erlang

Erlang – Language

- Functional
- Concurrent
- Distributed
- “Soft” real-time
- OTP (fault-tolerance, hot code update…)
- Open

Erlang – Typical Applications

- Telecoms
  - Switches (POTS, ATM, IP, …)
  - GPRS
  - SMS applications
- Internet applications (in particular chat)
  - jabber
  - Twitter
  - Facebook
- Credit card clearing
- 3D modelling (Wings3D)

History Lesson

- 1981 – Ericsson CSLab formed
- 1986 – Prolog “games”
- 1987 – Erlang name appears
- 1989 – Prototypes show 9–22 fold increase in design efficiency
- 1995 – AXE-N fails after 8 years
- 1998 – AXD301 delivered
- 1998 – Erlang banned; goes open source
- 2007 – Ericsson uses Erlang for new products

Essence of Erlang

- A simple functional language
- Direct asynchronous message passing
- Open Telecom Platform libraries
  - Practically “proven” programming patterns
  - Utilities

Data types – Constant

- Numbers
  - Integers (arbitrarily big)
  - Floats
- Atoms
  - `start_with_a_lower_case_letter`
  - `'Anything_inside_quotes\n\09'`

Data types – Compound

- Tuples
  - `{}`
  - `{atom, another_atom, 'PPxT'}`
  - `{atom, {tup, 2}, {{tup}, element}}`
- Lists
  - `[]`
  - `[65,66,67,68]` = `"ABCD"`
  - `[1, true]` and `[a | b]`

Data types – Compound

- Tuples
  - `{}`
  - `{atom, another_atom, 'PPxT'}`
  - `{atom, {tup, 2}, {{tup}, element}}`
- Lists
  - `[]`
  - `[65,66,67,68] = "ABCD"`
  - `[1, true]` and `[a | b]`

Allowed but not a good programming practice!

Data types – Compound

- Records
  - `-record(person,`
    `{name = “”,`
    `phone = [],`
    `address}).`
  - `X = #person{name = “Joe”, phone = [1,1,2], address= “here”}`
  - `X#person.name`
  - `X#person{phone = [0,3,1,1,2,3,4]}`

Modelling Haskell data

- ```
  data Tree a =
      Leaf a
    | Node (Tree a) (Tree a)
  ```

- edoc style “types”:
- ```
  tree(A) =
          {leaf, A}
        | {node, tree(A), tree(A)}
  ```

Variables

- Identifires
  - `A_long_variable_name`
  - Must start with an Upper Case Letter
  - Can store values
  - Can be bound only once!
  - Bound variables cannot change values

Functions and Modules

- Basic compilation unit is a module
  - Module name = file name (.erl)
- Modules contain function definitions
  - Some functions can be exported – usable from outside of the module

```
-module(math_stuff).
-export([factorial/1]).

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

Evaluation

```
factorial(3) matches N = 3 in clause 2
== 3 * factorial(3 - 1)
== 3 * factorial(2)
                        matches N =2 in clause 2
== 3 * 2 * factorial(2 - 1)
== 3 * 2 * factorial(1)
                        matches N = 1 in clause 2
== 3 * 2 * 1 * factorial(1 - 1)
== 3 * 2 * 1  * factorial(0)
== 3 * 2 * 1  * 1 (clause 1)
== 6
```

Pattern Matching

- `area({square,Side}) -> Side*Side.`

a pattern

- `{square, Side}` *matches* `{square, 4}` and *binds* `4` to variable `Side`

More Pattern Matching

- `{B, C, D} = {10, foo, bar}`
  - Succeeds - binds `B` to `10`, `C` to `foo` and `D` to `bar`
- `{A, A, B} = {abc, abc, foo}`
  - Succeeds - binds `A` to `abc`, `B` to `foo`
- `{A, A, B} = {abc, def, 123}`
  - Fails
- `[A,B,C,D] = [1,2,3]`
  - Fails

Even More Pattern Matching

- `[H|T]= [1,2,3,4]`
  - Succeeds - binds `H` to `1`, `T` to `[2,3,4]`
- `[H|T] = [abc]`
  - Succeeds - binds `H` to `abc`, `T` to `[]`
- `[H|T] = []`
  - Fails
- `{A,_, [B|_],{B}} = {abc,23,[22,x],{22}}`
  - Succeeds - binds `A` = `abc`, `B` = `22`

List Examples

```
-module(list_stuff).
-export([average/1, average/2]).

average(X) -> sum(X) / len(X).
average(A, B) -> (A+B)/2.

sum([H|T]) -> H + sum(T);
sum([])    -> 0.

len([H|T]) -> 1 + len(T);
len([])    -> 0.
```

Loops – Tail Recursion

- Inefficient recursive definition

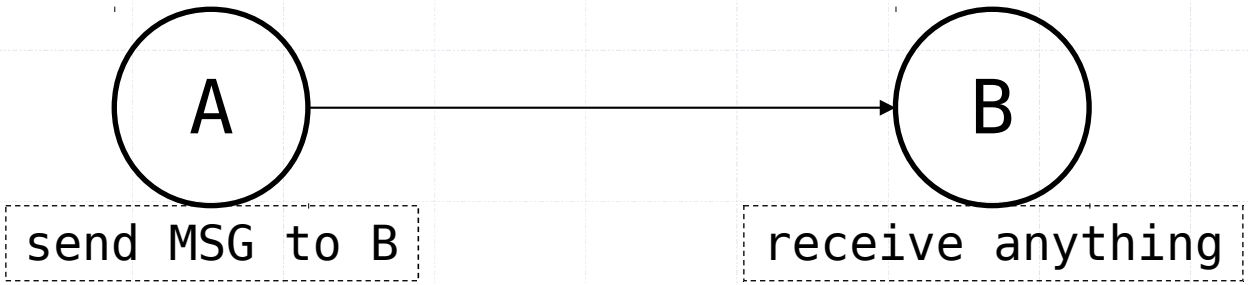
```
len([H|T]) -> 1 + len(T);
len([])    -> 0.
```

- Fast tail recursive version = while loop

```
len(List) -> len_a(List, 0).

len_a([_|T], Acc) -> len_a(T, Acc+1);
len_a([],    Acc) -> Acc.
```

Concurrent Programming

- Based on Message Passing:
  - Q. What form of synchronisation?
  - A. Asynchronous
  - Q. What form of process naming?
  - A. Direct, asymmetric



Processes

- Process = a function evaluated in a separate thread

```
-module(process).
-export([start/0]).

start() ->
    Pid = spawn(fun run/0),
    io:format("Spawned ~p~n",[Pid]).

run() -> io:format("Hello!~n",[]).
```

BIFs

- Built-in functions
  - `spawn`, `spawn_link`, `atom_to_list`
- Don’t need importing
- Mostly in module called `erlang`
- Some have guaranteed termination
  - These can be used in guards

Process Creation

```
-module(echo).
-export([start/0]).

start() ->
    Pid = spawn(fun() -> loop(42) end),
    Pid ! {self(), hello},
    io:format("Sent hello.~n",[]).

%%continues
```

Message Passing

```
%%continuation
loop(Number) ->
   receive
      {From, Msg} ->
         From ! {self(), Number},
         io:format(
            "Received ~p, N: ~p~n",
            [Msg, Number]),
         loop(Number+1);
      stop ->
         true
   end.
```

Conclusions – Erlang

- Functional
- Concurrent
  - Basics