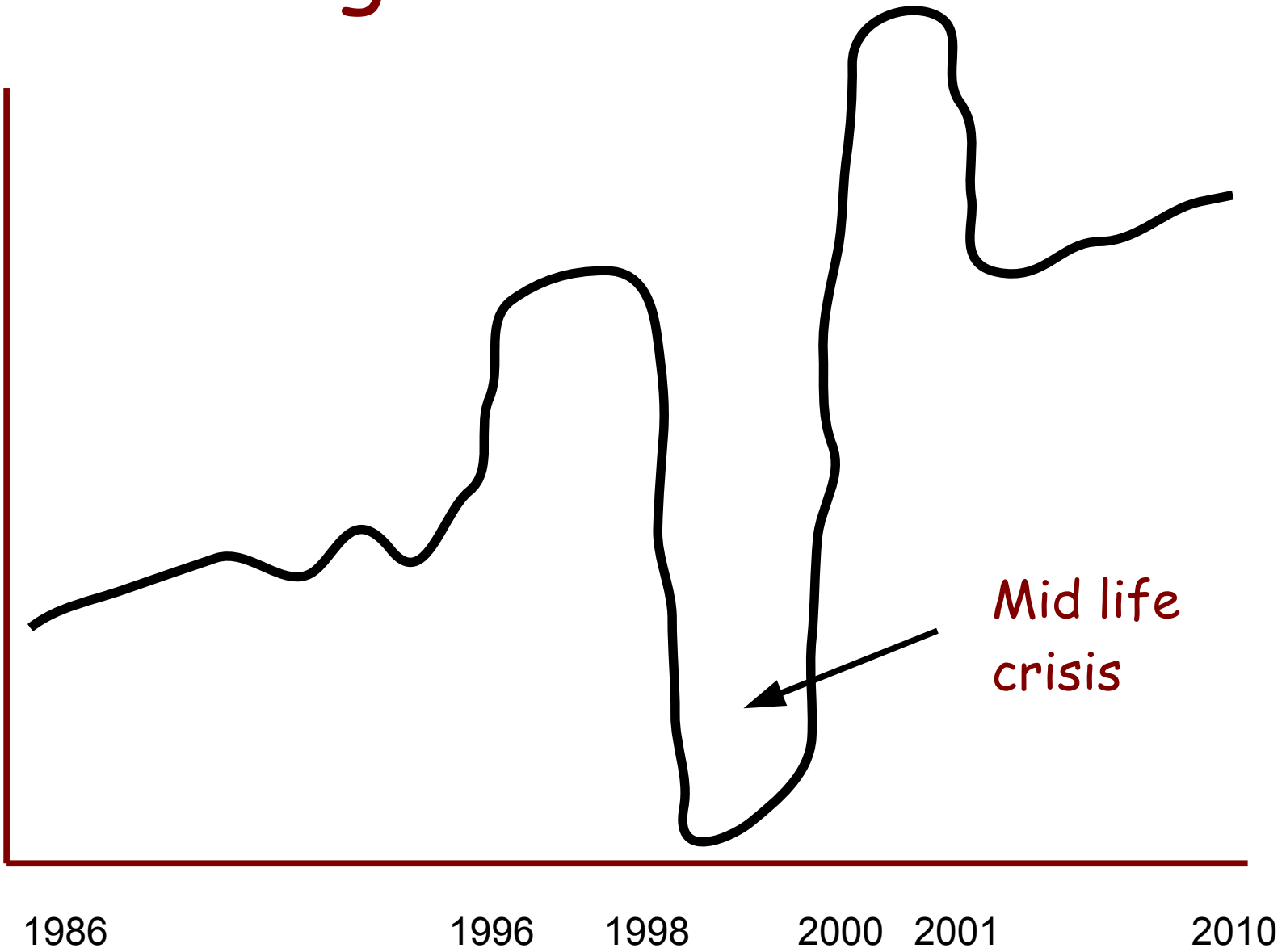
The Ideas in Erlang

Joe Armstrong

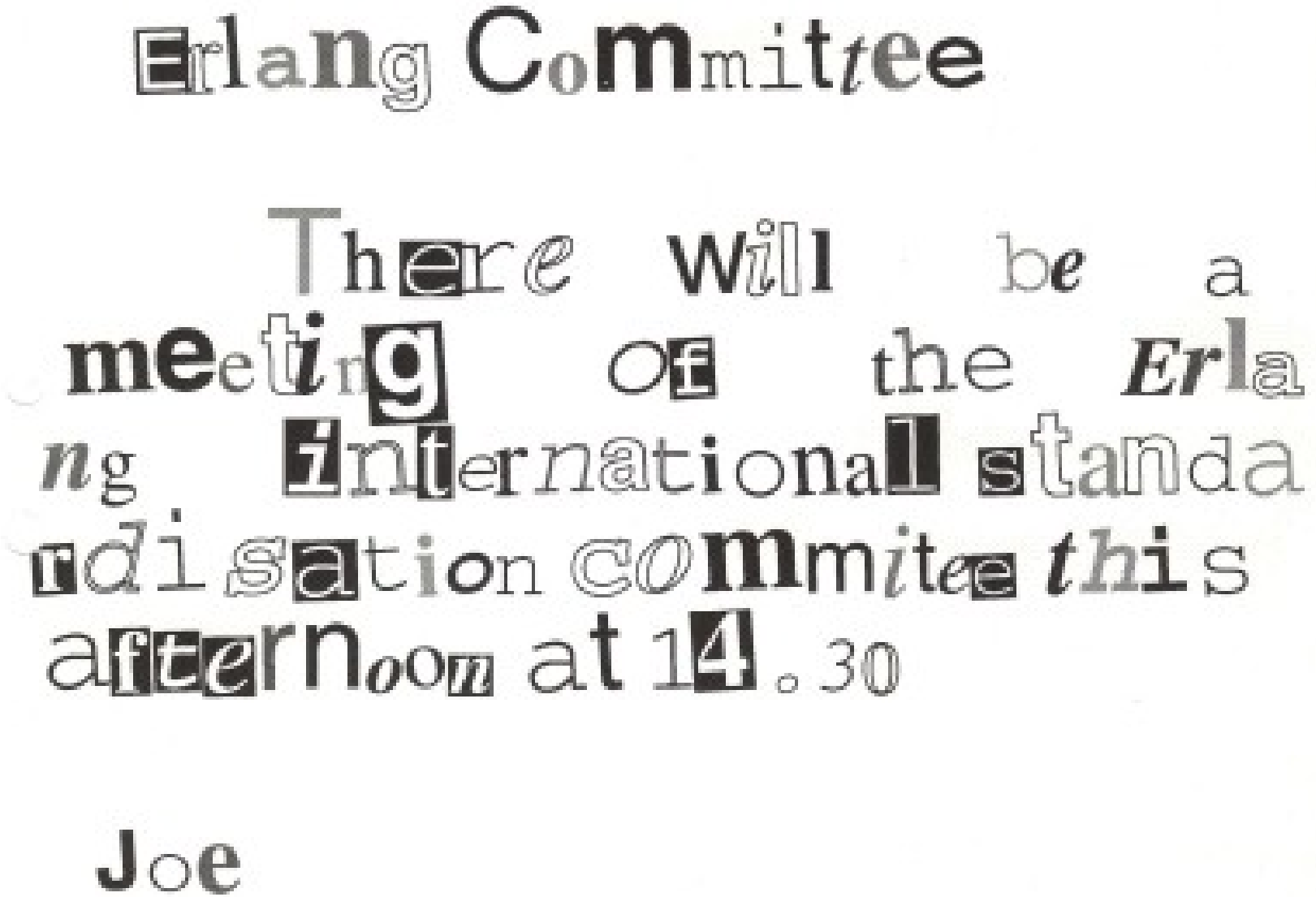
Plan

- A bit of history
- 3 things missing
- 1 big mistake
- 2 good ideas
- 3 great ideas

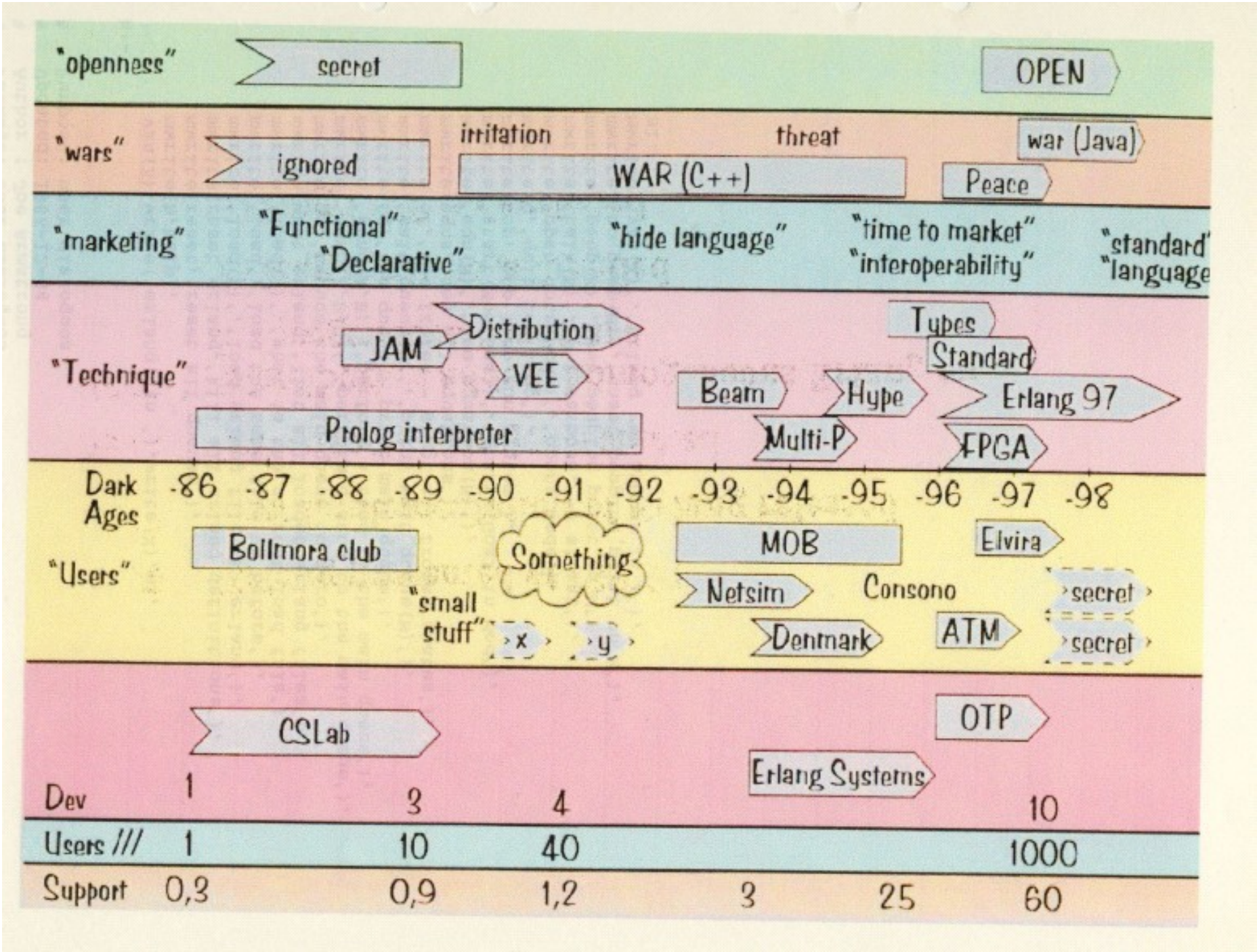
Erlang




ERICSSON


Erlang Committee
There will be a
meeting of the Erla
ng International standa
rdisation commitee this
afternoon at 14.30
Joe

1985 - 1998


"openness"
secret
OPEN
"wars"
ignored
irritation
WAR (C++)
threat
war (Java)
Peace
"marketing"
"Functional"
"Declarative"
"hide language"
"time to market"
"interoperability"
"standard"
"language
"Technique"
JAM
Distribution
VEE
Prolog interpreter
Beam
Hype
Multi-P
Types
Standard
Erlang 97
FPGA
Dark Ages
-86
-87
-88
-89
-90
-91
-92
-93
-94
-95
-96
-97
-98
"Users"
Bollmora club
"small stuff"
x
y
Something
MOB
Netsim
Denmark
Consono
ATM
Elvira
secret
secret
CSLab
OTP
Erlang Systems
Dev
1
3
4
10
Users ///
1
10
40
1000
Support
0,3
0,9
1,2
3
25
60

Timeline

- 1986-1989 - Productive.
- 1989 – 1996 – Wars
- 1996-1998 – Peace.
- 1998 – 2000 Mid life crisis
- 2001 – 2008 Continuous slow growth.
- 2008 – Upturn.

Early

1985 - 1989

Timeline

- Programming POTS/LOTS/DOTS (1885)
- A Smalltalk model of POTS
- A telephony algebra (math)
- A Prolog interpretor for the telephony algebra
- I add processes to prolog
- Prolog is too powerful (backtracking)
- Deterministic prolog with processes
- "Erlang" !!! (1986)
- ...
- Compiled to JAM code (1989)
-

Find a better way of programming

1985 - "find better ways of programming telephony"

SPOTS - LOTS

SPOTS = SPC for POTS

SPC = Stored Program Control ("computer controlled")

POTS = Plain Ordinary Telephone Service

Write telephony in many different languages

No "plan" to make another programming language

Pre history

AXE – programmed in PLEX

PLEX
- Programming language for exchanges)
- Proprietary
- blocks (processes) and signals
- in-service code upgrade

Eri Pascal

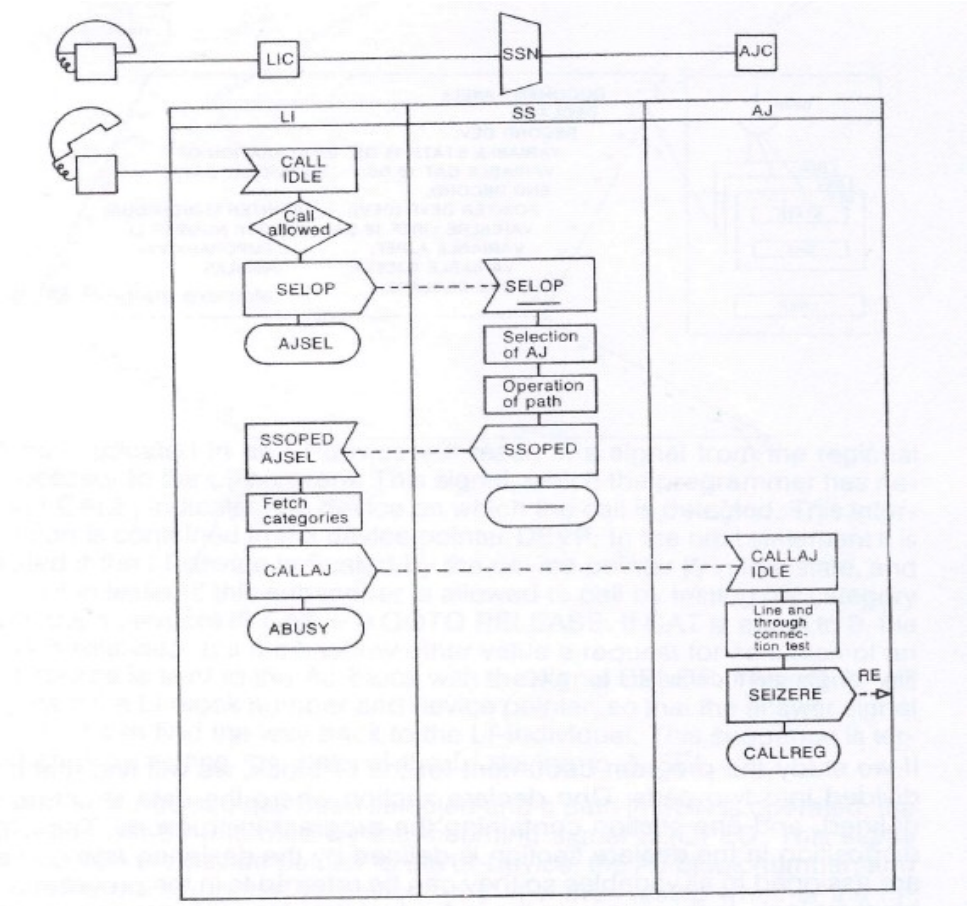
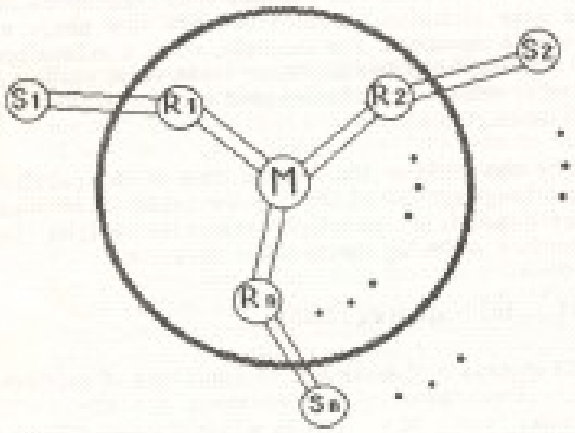


Fig. 11 AXE programming by PLEX

Phoning Philosopher's

7. A Telephone Exchange Model in PARLOG

Our exchange is modelled, in Parlog, as a set of communicating parallel logic processes, as illustrated in the figure below. Communication between logic processes takes place through unidirectional channels. A channel is represented by an infinite stream of messages.



The telephone sets are represented by external processes (Si's), each process (Si) communicates

State is an unbound variable which is bound to a value in the Manager process activation as follows:

```
manager([check_called(Rj,State)|To_M],
        From_M) :-
            get_state(Rj,State), ... .
```

in which the variable State gets a value to be bound in the caller_process communicating with the manager. This example is simplified a bit for illustration purposes. In the real program there are extra merging and forking processes to control communication to/from the manager.

An example of a time-dependent process is the hot-line service. The hot-line is a service provided by the exchange in which if a phone is picked up, and if no dialing has started within a given time, the system automatically dials a predefined number. This process is described in Parlog as follows:

```
resource_process(Ri, [off_hook|From_S],
                     From_M, To_S, To_M) :-
    idle(Ri) :
        start_call(Ri, From_S, From_M, Alarm,
                   Stop_cmd, To_S, To_M),
        timer(some_time, Stop_cmd, Alarm).
```

Conclusion – *Concurrent Logic programming with channel communication*

Armstrong, Elshiewy, Virding (1986)

The Telephony Algebra - (1985)

idle(N) means the subscriber N is idle
on(N) means subscribed N in on hook
...

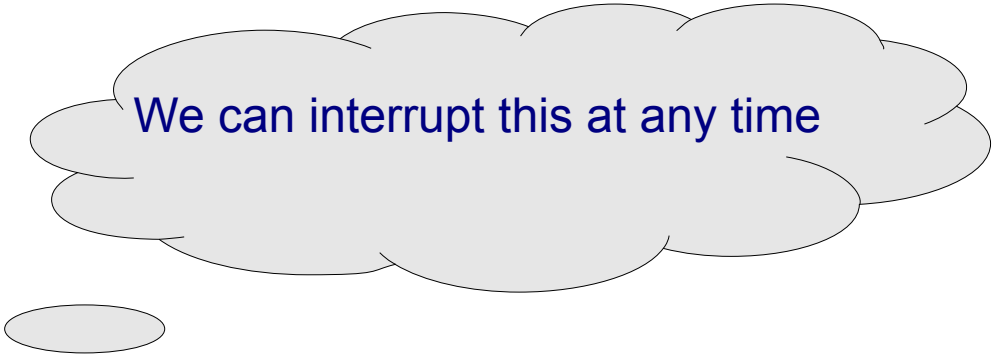
+t(A, dial_tone) means add a dial tone to A

process(A, f) :- on(A), idle(A), +t(A,dial-tone),
+d(A, []), -idle(A), +of(A)

Using this notation, POTS could be described using fifteen rules. There was just one major problem: the notation only described how one telephone call should proceed. How could we do this for thousands of simultaneous calls?

The reduction machine - (1985)

A -> B,C,D.
B -> x,D.
D -> y.
C -> z.



A
B,C,D
x,D,C,D
D,C,D
y,C,D
C,D
z,D
D
y
{}

A,B,C, D = nonterminals

x,y,z = terminals

To reduce X,...Y...
If X is a nonterminal replace it by it's definition
If X is a terminal execute it and then do ...Y...

Aside – term rewriting is tail recursive

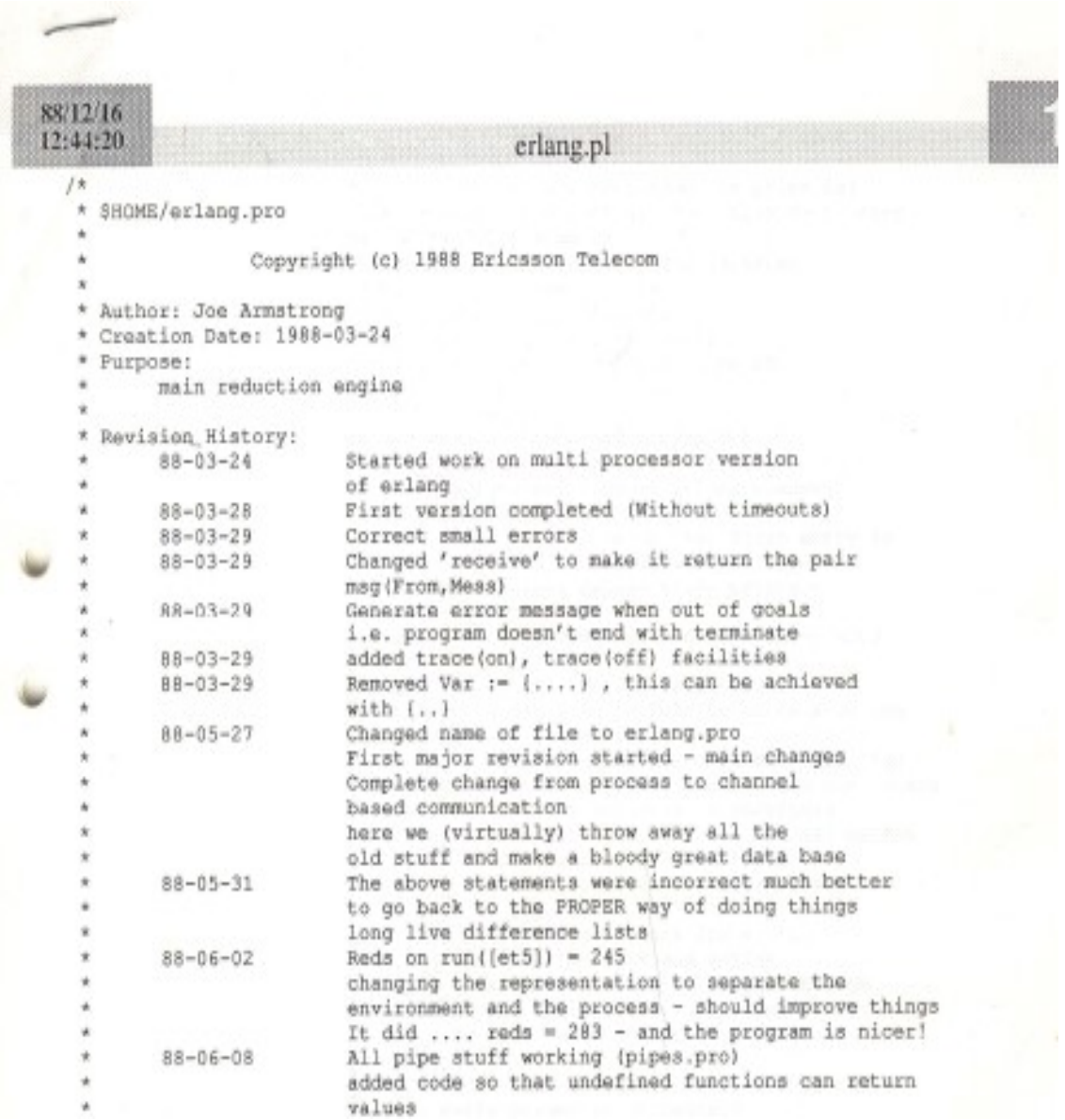
A -> x,y,A

A
x,y,A
y,A
A
x,y,A
y,A
A
...

```
loop(X) ->
    ...
    loop(X).
```

1988 – Interpreted Erlang

- 4 days for a complete re-write
- 245 reductions/sec
- semantics of language worked out
- Robert Virding joins the "team"



88/12/16 12:44:20 erlang.pl

```
/*
* $HOME/erlang.pro
*
*              Copyright (c) 1988 Ericsson Telecom
*
* Author: Joe Armstrong
* Creation Date: 1988-03-24
* Purpose:
*      main reduction engine
*
* Revision History:
*      88-03-24        Started work on multi processor version
*                      of erlang
*      88-03-28        First version completed (Without timeouts)
*      88-03-29        Correct small errors
*      88-03-29        Changed 'receive' to make it return the pair
*                      msg(From,Mess)
*      88-03-29        Generate error message when out of goals
*                      i.e. program doesn't end with terminate
*      88-03-29        added trace(on), trace(off) facilities
*      88-03-29        Removed Var := {....} , this can be achieved
*                      with {..}
*      88-05-27        Changed name of file to erlang.pro
*                      First major revision started - main changes
*                      Complete change from process to channel
*                      based communication
*                      here we (virtually) throw away all the
*                      old stuff and make a bloody great data base
*      88-05-31        The above statements were incorrect much better
*                      to go back to the PROPER way of doing things
*                      long live difference lists
*      88-06-02        Reds on run({et5}) = 245
*                      changing the representation to separate the
*                      environment and the process - should improve things
*                      It did .... reds = 283 - and the program is nicer!
*      88-06-08        All pipe stuff working (pipes.pro)
*                      added code so that undefined functions can return
*                      values
```

erlang vsn 1.05

h	help
reset	reset all queues
reset_erlang	kill all erlang definitions
load(F)	load erlang file <F>.erlang
load	load the same file as before
load(?)	what is the current load file
what_erlang	list all loaded erlang files
go	reduce the main queue to zero
send(A,B,C)	perform a send to the main queue
send(A,B)	perform a send to the main queue
cq	see queue - print main queue
wait_queue(N)	print wait_queue(N)
cf	see frozen - print all frozen states
eqns	see all equations
eqn(N)	see equation(N)
start(Mod,Goal)	starts Goal in Mod
top	top loop run system
q	quit top loop
open_dots(Node)	opens Node
talk(N)	N=1 verbose, =0 silent
peep(M)	set peeping point on M
no_peep(M)	unset peeping point on M
vsn(X)	erlang vsn number is X

The manual

1986 (or 85)

```
joe> cat  test.erlang                    listing of program
module(test).
1: start --> write('hello'),nl,go.
2: go --> start_proc(foo1,test,test),start_proc(foo2,test,test).
3: test --> wait.
4: wait,[X,1].
5: wait,[X,Y] --> write(received(Y)),nl,wait.
joe> erlang                              start erlang
erlang vsn 1.05
type h for help

yes
| ?- load(test).                         load  the program in test.erlang
translating the file:test.erlang
Module:test
12345                                    equantion numbers are displayed
compiling the file:test.obj
[/u/joe/logic/quintus/erlang/dots/test.obj compiled (1.950 sec 480 bytes)]
loading completed ...
```

Running a program

The Prolog interpreter (1986)

```
%   Package: make erlang
%   Author : Joseph Armstrong
%   Updated: 1986-12-18
%   Purpose: compiles and loads the erlang system

% this line MUST come first
:- ensure_loaded('/u/joe/logic/quintus/lib/set_library.pl').

%   vsn 1.03 lost in the mists of time
%   vsn 1.04 added modules and peeping (removed tracing)
%   vsn 1.05 mean version - fails in top loop to conserve space

%   vsn 1.06
%       added process constants
%               added commands
%               start_proc(Id,Module,Goal,Process_constants)
%                       is similar to start_proc/3 with added
%                       Process_constans
%                       Process_constants are a list of pairs of the form
%                               [(Key,Val),(Key1,Val1),...]
%               pconst(Key,Val)
%                       looks up the value of the process constant
%                       with key Key - Binds result to Value or makes
%                       error messages
%       added table driven number analyser
%               anal(Seq,Res)
%                       given a dialled sequence Seq binds Res
%                       to one of [invalid,get_more_digits,matched(Reason)]

vsn(1.06).


:- ensure_loaded(library(prims)).
:- ensure_loaded(library(findall)).

:- ensure_loaded('erlang1.04').
:- ensure_loaded(run).
:- ensure_loaded(queue).
:- ensure_loaded(reduce).
:- ensure_loaded(resume).
:- ensure_loaded(timeout).
```

version 1.06
dated
1986-12-18

1.03 "lost in the mists of time"

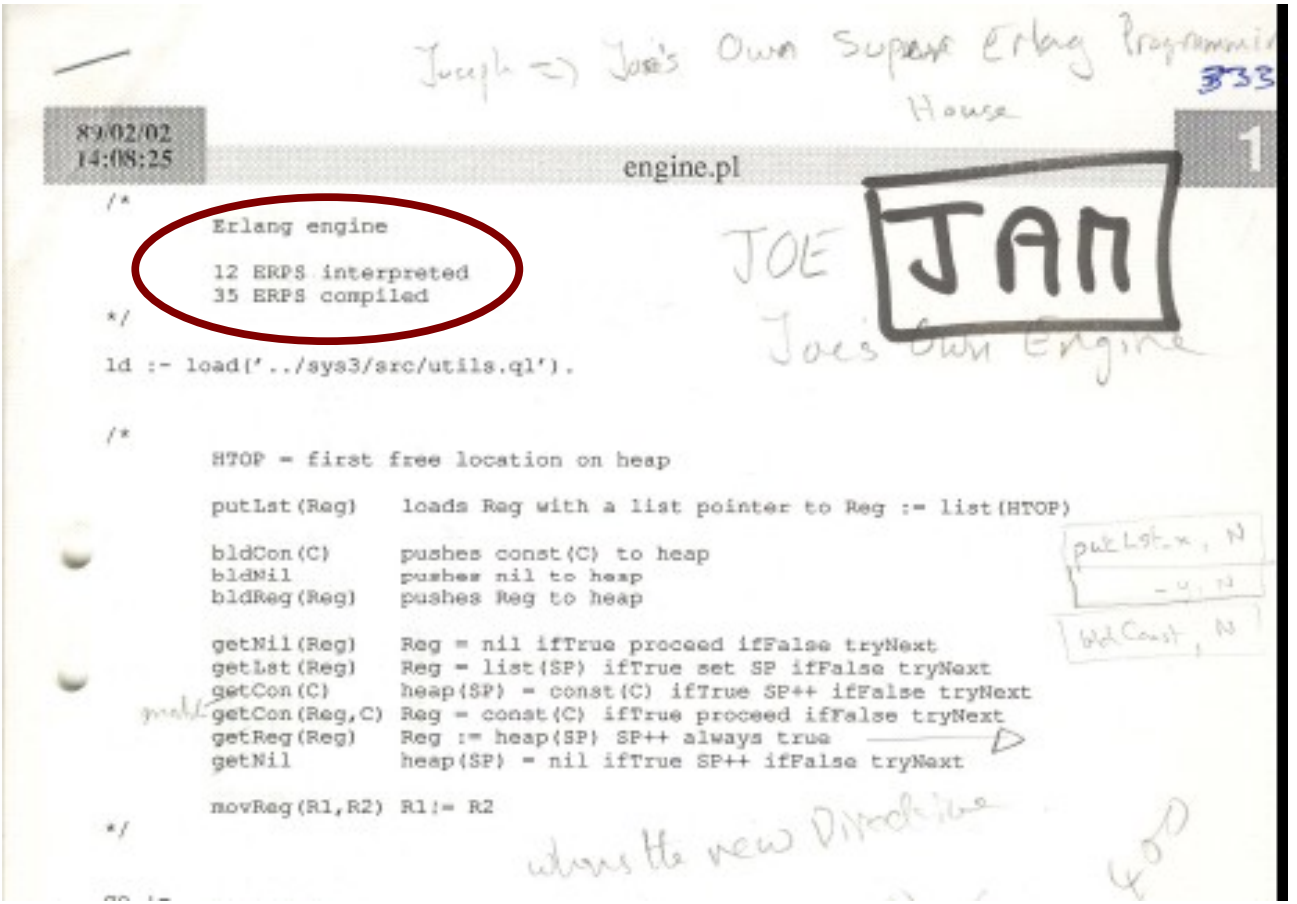
1989 – The need for speed

ACS- Dunder
- "we like the language but it's too slow" - must be 40 times faster

Mike Williams writes the emulator (in C)

Joe Armstrong writes the compiler

Robert Virding writes the libraries



An early JAM compiler (1989)

```
sys_sys.erl              18 dummy
sys_parse.erl           783 erlang parser
sys_ari_parser.erl      147 parse arithmetic expressions
sys_build.erl           272 build function call arguments
sys_match.erl           253 match function head arguments
sys_compile.erl         708 compiler main program
sys_lists.erl            85 list handling
sys_dictionary.erl       82 dictionary handler
sys_utils.erl            71 utilities
sys_asm.erl             419 assembler
sys_tokenise.erl        413 tokeniser
sys_parser_tools.erl     96 parser utilities
sys_load.erl            326 loader
sys_opcodes.erl         128 opcode definitions
sys_pp.erl              418 pretty printer
sys_scan.erl            252 scanner
sys_boot.erl             59 bootstrap
sys_kernel.erl            9 kernel calls
18 files                4544
```

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

{info, fac, 1}
 {try_me_else, label1}
        {arg, 0}
        {getInt, 0}
        {pushInt, 1}
        ret
 label1: try_me_else_fail
        {arg, 0}
        dup
        {pushInt, 1}
        minus
        {callLocal, fac, 1}
        times
        ret
```

Like the WAM with added primitives for spawning processes and message passing

factorial

```
rule(fac, 0)  ->
   [pop,{push,1}];
rule(fac, _)  ->
   [dup,{push,1},
    minus,
    {call,fac},
    times].
```

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

{info, fac, 1}
 {try_me_else, label1}
        {arg, 0}
        {getInt, 0}
        {pushInt, 1}
        ret
 label1: try_me_else_fail
        {arg, 0}
        dup
        {pushInt, 1}
        minus
        {callLocal, fac, 1}
        times
        ret
```

factorial

```
rule(fac, 0)  -> [pop,{push,1}];
rule(fac, _)  -> [dup,{push,1},minus,{call,fac},times].

run() -> reduce0([{call,fac}], [3]).

reduce0(Code, Stack) ->
    io:format("Stack:~p Code:~p~n",[Stack,Code]),
    reduce(Code, Stack).

reduce([],[X])                      -> X;
reduce([{push,N}|Code], T)    -> reduce0(Code, [N|T]);
reduce([pop|Code], T)           -> reduce0(Code, tl(T));
reduce([dup|Code], [H|T])       -> reduce0(Code, [H,H|T]);
reduce([minus|Code], [A,B|T]) -> reduce0(Code, [B-A|T]);
reduce([times|Code], [A,B|T]) -> reduce0(Code, [A*B|T]);
reduce([{call,Func}|Code], [H|_]=Stack) ->
    reduce0(rule(Func, H) ++ Code, Stack).
```

factorial

```
> fac:run().
Stack:[3] Code:[{call,fac}]
Stack:[3] Code:[dup,{push,1},minus,{call,fac},times]
Stack:[3,3] Code:[{push,1},minus,{call,fac},times]
Stack:[1,3,3] Code:[minus,{call,fac},times]
Stack:[2,3] Code:[{call,fac},times]
Stack:[2,3] Code:[dup,{push,1},minus,{call,fac},times,times]
Stack:[2,2,3] Code:[{push,1},minus,{call,fac},times,times]
Stack:[1,2,2,3] Code:[minus,{call,fac},times,times]
Stack:[1,2,3] Code:[{call,fac},times,times]
Stack:[1,2,3] Code:[dup,{push,1},minus,{call,fac},times,times,times]
Stack:[1,1,2,3] Code:[{push,1},minus,{call,fac},times,times,times]
Stack:[1,1,1,2,3] Code:[minus,{call,fac},times,times,times]
Stack:[0,1,2,3] Code:[{call,fac},times,times,times]
Stack:[0,1,2,3] Code:[pop,{push,1},times,times,times]
Stack:[1,2,3] Code:[{push,1},times,times,times]
Stack:[1,1,2,3] Code:[times,times,times]
Stack:[1,2,3] Code:[times,times]
Stack:[2,3] Code:[times]
Stack:[6] Code:[]
```

787 Kreds/

sec

Speedups

- Prolog Erlang Interpretor (1988) – 245 reds/sec
- Prolog JAM emulator – 35 reds/sec
- C Erlang JAM emulator (1989) – 30K reds/sec
- C Erlang BEAM emulator (2010) – 9 Mega reds/sec
- Erlang JAM emulator (2010) – 787K reds/sec
- Speedup in 21 years is 9M/245 = 36734
- N^21 = 36734 so N = 1.65 (65% / year)
- Hardware (1.15^21) * 245 = 4.6 Kreds/sec

1989 - Clarity

At the end of 1989

- Knew what the requirements were
- Could compile Erlang
- Knew the syntax
- Had an error recovery model (links, exceptions)
- Had a few users
- Had a course and material

Accepted ideas

- Links/process groups (one crash=all crash)
- Mailbox semantics
- Dynamic code change
- Error recovery model

  (we tried dozens)

Rejected ideas

- Named pipes

  and the pipe algebra

  split/merge/join/fanout

  (reappears as AMQP / FBP / ...)
- Mutable data

Requirements (1989)

- Handling a very large number of concurrent activities
- Actions to be performed at a certain point of time or within certain time
- Systems distributed over several computers
- Interaction with hardware
- 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

Teaching Erlang

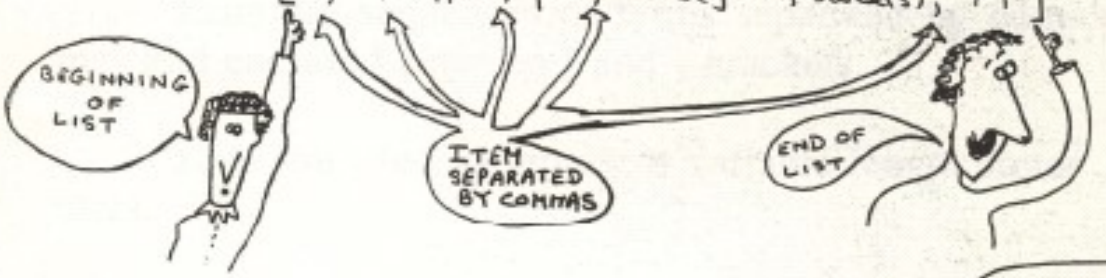
Before powerpoint

FUN WITH LISTS

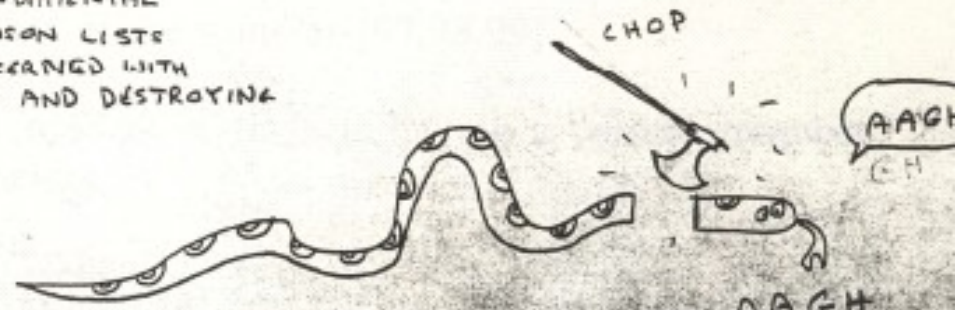
PART 3

A **LIST** IS A STRUCTURE THAT IS USED FOR REPRESETIN A VARIABLE NUMBER OF TERMS. LISTS ARE WRITTEN BY ENCLOSING THE ITEMS (WHICH ARE SEPARATED BY COMMAS) BETWEEN [AND] brackets

X = [1, 3, Apple, pie, likes(john, sweets), 79]



THE FUNDAMENTAL OPERATIONSON LISTS ARE CONCERNED WITH BUILDING AND DESTROYING LISTS



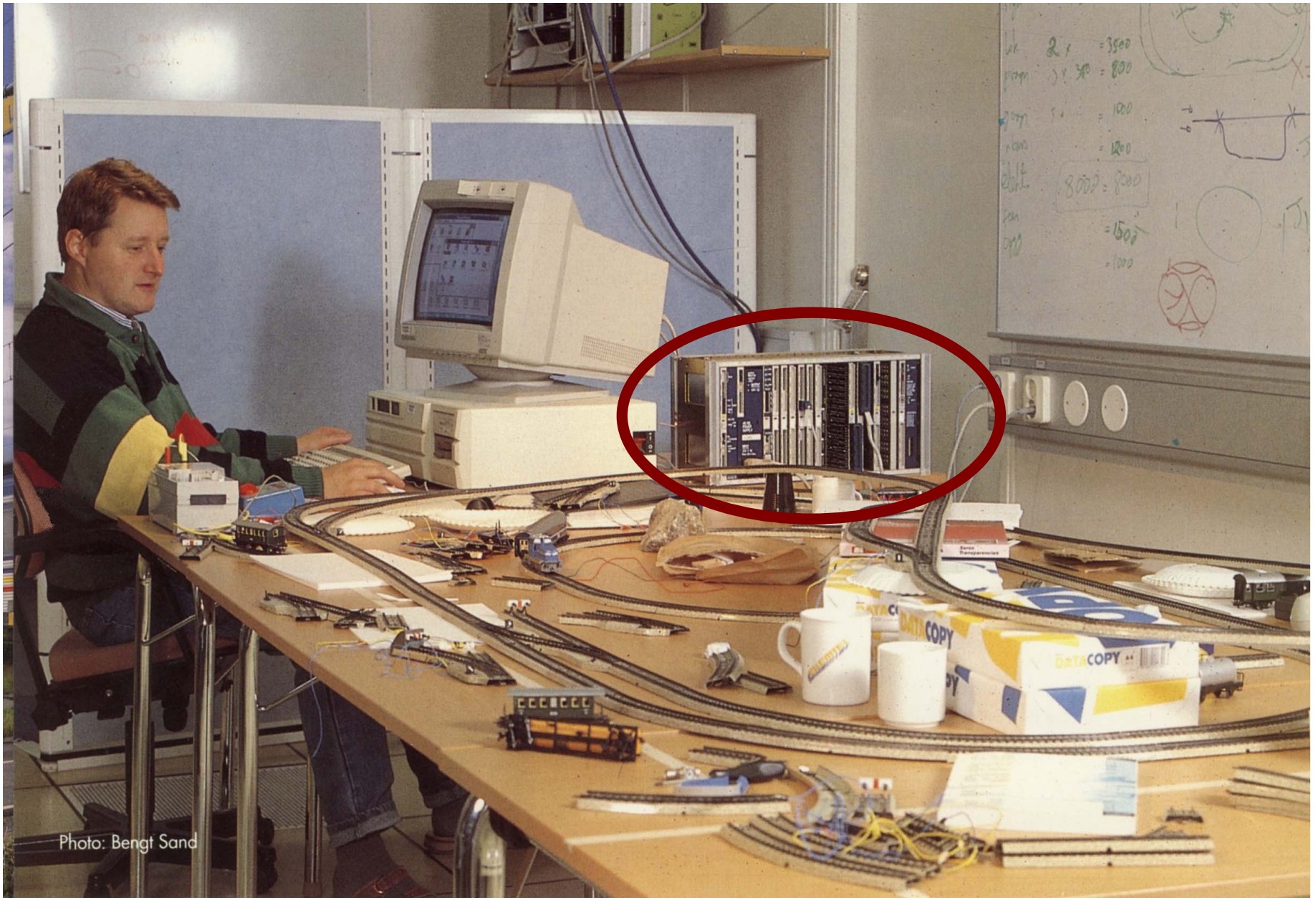
LISTS HAVE <u>HEADS</u> AND <u>TAILS</u> (LIKE SNAKES!)

A LIST WITH HEAD

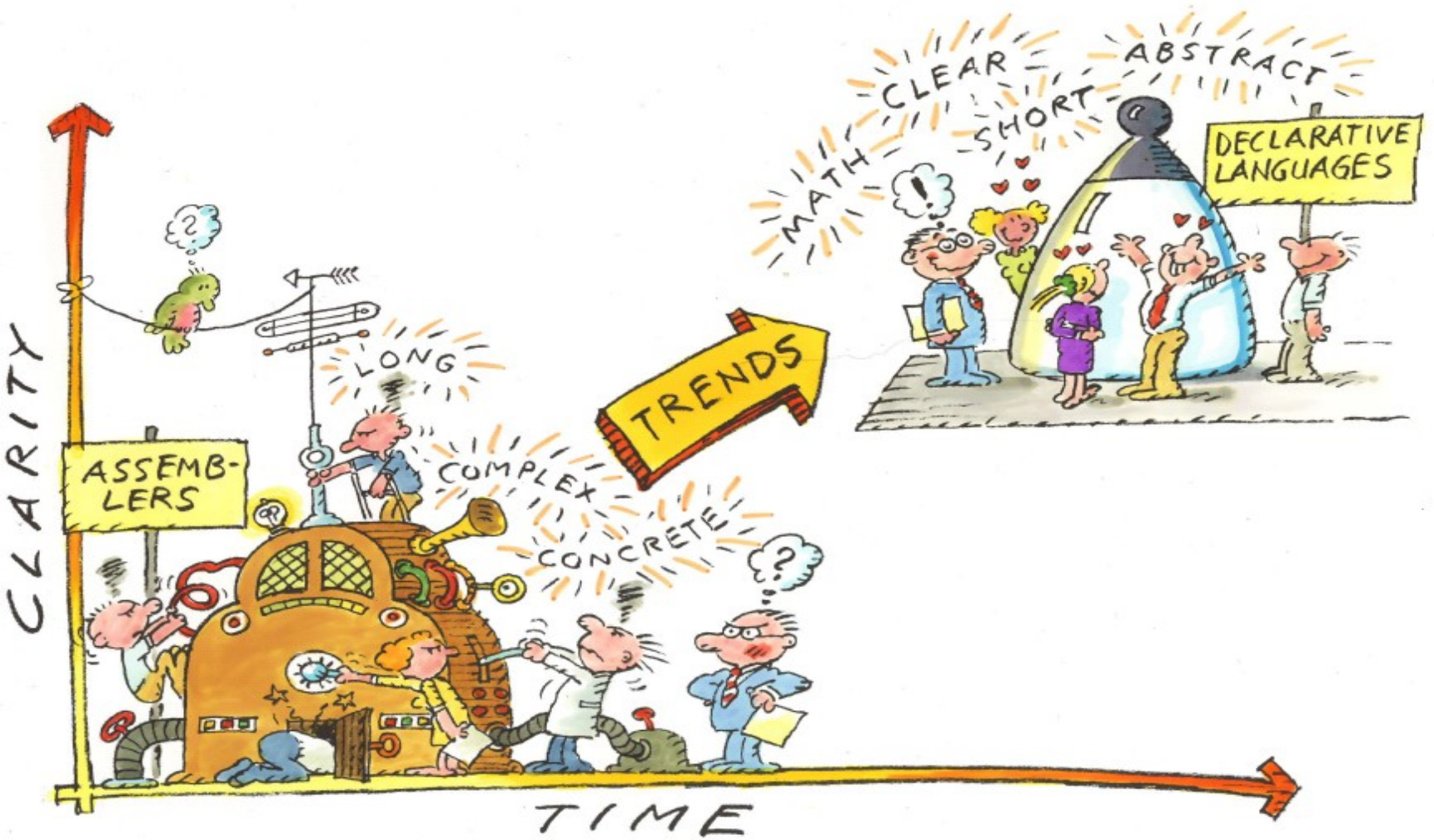
Middle

By 1990 things
were going
so well
that we
could
...

Buy a train set



Have nice slides made



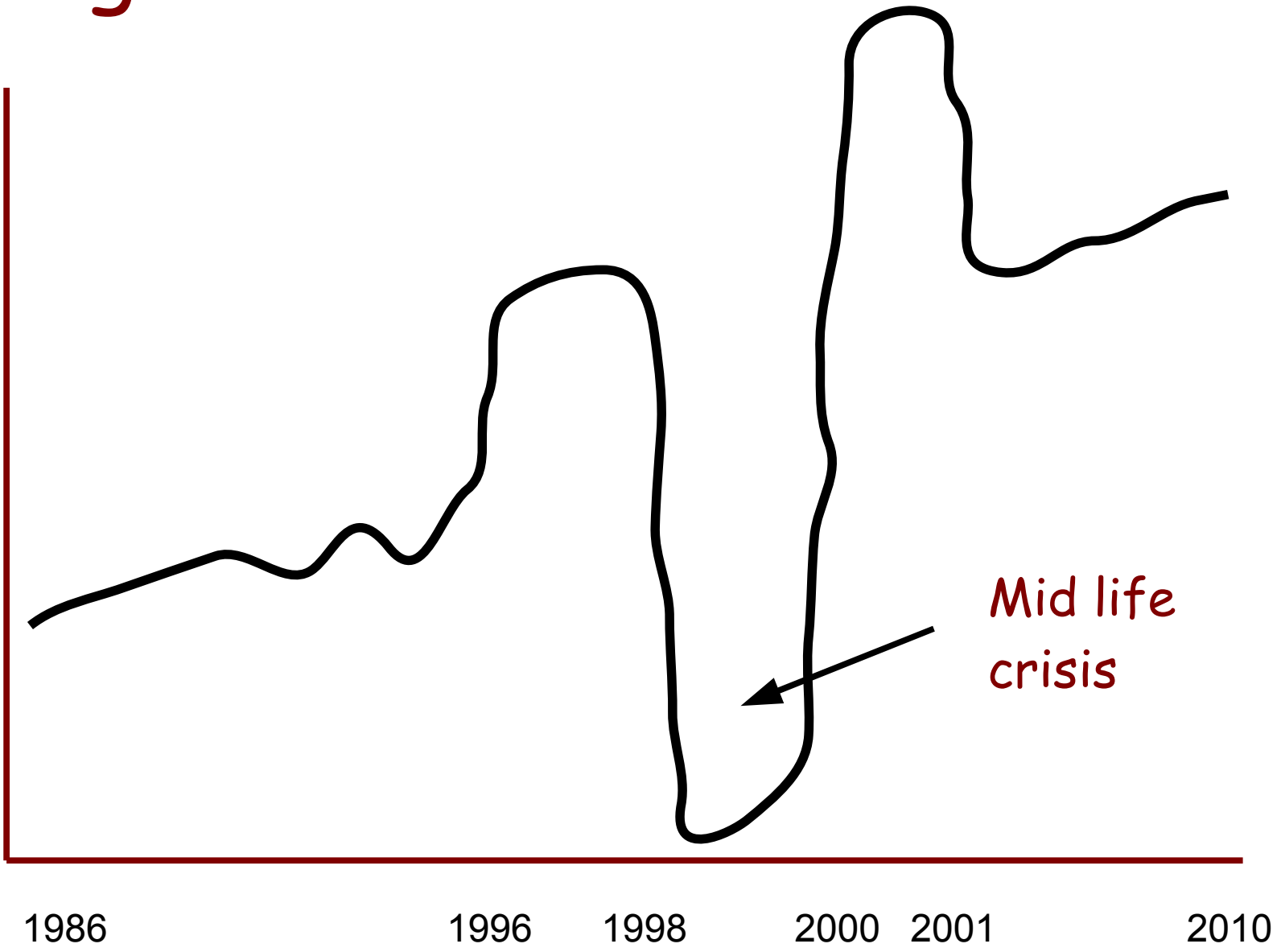
We added new stuff

- Distribution
- OTP structure
- BEAM
- HIPE
- Type tools
- Philosophy
- Bit syntax
- OTP tools
- Documented way of doing things

Concurrency Oriented Programming (MIT 2002)

Mid Life Crisis

Erlang Enthusiasm



- Banned (1998)
- Open Source (1998)
- Quit Ericsson
- IT Boom
- Startups
- Blutail Acquired $$$ (2000)
- IT Crash

Back at the farm

OTP maintains a low profile

- "Rename the project"
- "Don't frighten the users"
- "Keep head down"
- "Do some technical stuff"
- "Hope nobody notices us"

Mature

Becoming mainstream

- Long time to change anything big
- More demand for books/documentation/consultants/teaching
- Many success stores (not just one)
- Rapid change of small things (GIT hub)
- Easier to fund
- Hey, it works !

3 things missing

Hashmaps

```
foo(<{a:X, b:Y | T }>) ->
    ...

> foo(<{c:23, a:123, b:abc}>)

Binds X=123, Y=abc T=<{c:23}>
```

HOMS + introspection

> module_to_list(lists).
[{append,2,F1},{sort,1,F2}...]

> function_to_conc(F1).
“append([H|T], L) -> ...”

> function_to_abs(F1).
{function,append,2,[{clause,...}]}

Receive a fun

```
F = fun({foo,X}) -> ... end

receive(Fun)
```

1 big mistake

We lost too much prolog

```
friends(A, B) :- likes(A, X),likes(B, X).

friends(L) ->
  [{A,B} ||
    {likes,A,X} <- L, {likes,B,X1} <- L,
      X == X1}]
```

2 good things

Lightweight processes are ok

- Java "proved" GC
- Smalltalk "proved" messaging
- Erlang "proved" process belong to the PL

  NOT the OS

"An OS is what the language designers forgot"

OTP Behaviours

- Like Higher order functions
- Can encapsulate non functional concepts (like fail over etc.) in a precise way
- Enforce best practise
- All large teams to work together

3 great things

Bit Syntax

- Pattern matching over bits

```
unpack(<<Red:5,Green:6,Blue:5>>) ->
    ...
```

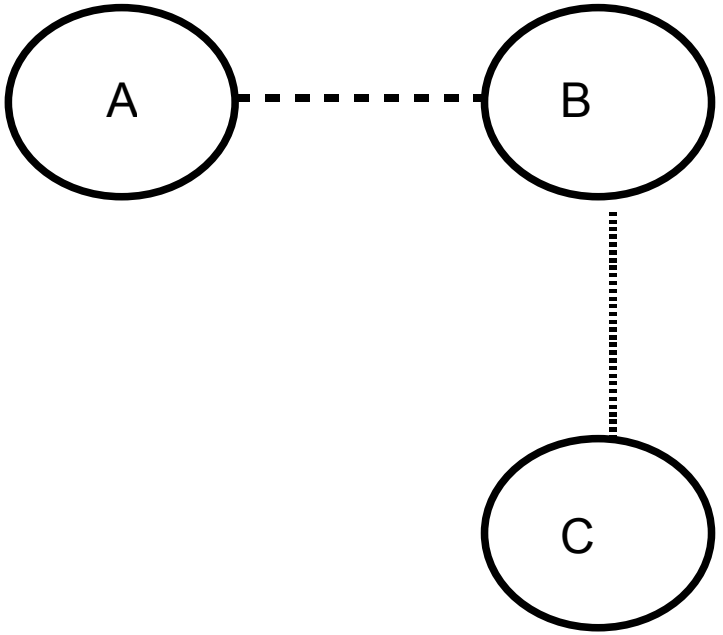
Due to Klacke
(Claes Vikström)

```
-define(IP_VERSION, 4).
-define(IP_MIN_HDR_LEN, 5).

DgramSize = size(Dgram),
case Dgram of
    <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
      ID:16, Flgs:3, FragOff:13,
      TTL:8, Proto:8, HdrChkSum:16,
      SrcIP:32,
      DestIP:32, RestDgram/binary>> when HLen>=5,
4*HLen=<DgramSize ->
        OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
        <<Opts:OptsLen/binary,Data/binary>> = RestDgram,
    ...
end.
```

(unpack Ipv4 datagram)

Links



A is linked to B
B is linked to C

If any process crashes an EXIT message is sent to the linked processes

This idea comes from the "C wire" in early telephones (ground the C wire to cancel the call)

Encourages "let it crash" programming

Non defensive programming

- Program only the happy case
- Let some other process fix the error
- "let it crash"

The good ideas

- Agent programming works
- Copying data is better than shared memory
- Messages are good to isolate things
- The bit syntax is great
- Pure works "most of the time"
- Defensive programming is not necessary "let it crash"

The End