ERLANG

# Innehåll 1(2)

- Innehåll (2 oh)
- Introduktion (4 oh)
- Komma igång (1 oh)
- Datatyper (2 oh)
- Programmering, sekventiella program (8 oh)
- Programmering, parallella program (5 oh)

forts. . .

# Innehåll 2(2)

- Felhantering (6 oh)
- Behaviours – design patterns (1 oh)
- Diverse: macron, bitsyntax, koduppgradering (6 oh)
- Koppling till andra språk (1 oh)
- Verktyg som följer med (1 oh)
- Avslutning/Läsa mer (2 oh)
- Större exempel ifall vi hinner (4 oh)

# Historia

CSLab på Ericsson letar efter ett språk att programmera telefonväxlar i.

**1982–85** Experiment med ett 20-tal existerande språk, bl.a. Lisp, Prolog, Parlog.

**1986–** Experiment med Erlang. En interpretator skriven i Prolog kör Erlangkod. Man provar att skriva en MD 110.

**1991** JAM.

**1993** Erlang blir distribuerat.

**1997** BEAM.

**1998** Erlang blir Open Source.

## Användning

- AXD 301
- Bluetail→Altheon→Nortel: Mail robustifier, Web prioritizer
- Mobility server
- Sendmail
- One2one
- NETSim

# Egenskaper

- Funktionellt
- Dynamisk typning
- Garbage collection
- Väldigt lättviktigt att skapa processer
- Iteration genom rekursion
- Liten kodvolym $\Rightarrow$ ökad produktivitet, färre fel.
- Feltolerans

# Var passar Erlang och var inte

## Erlang i sitt esse

- Distribuerade/parallella problem
- Kommunikation
- Hög tillgänglighet (robust kod, live koduppgradering)

## Inte riktigt Erlangs cup of tea

- Scriptspråk (SAE, escript)
- Matrisberäkningar, signalbehandling, bildbearbetning, ...
  Interface:a kod i annat språk: externt program/inlänkad driver

# Komma igång

**Hemsidan** `http://www.erlang.org/`

- Dokumentation: halva Erlangboken, API
- Ladda hem Open Source Erlang

**Detta dokument** hittar du på:

`http://www.lysator.liu.se/~tab/erlang/upplysning.html`

**Starta Erlang** på Lysator:

```
lysator% module add erlang
lysator% erl
Erlang (BEAM) emulator version 5.0.2 [source]

Eshell V5.0.2  (abort with ^G)
1>
```

# Datatyper: atomer och tal

```
> atom.
atom
> 'Atom'.
'Atom'
> 'klurig \'atom'.
'klurig \'atom'

% kommentarer start med %
% de varar till radslut
```

```
> 1.
1
> 1.74.
1.74000
> 16#abc.        % basen 16
2748
> 13#9a.
127

% skifta vänster 55 bitar
> 43 bsl 55.
1549238271815450624
```

# Datatyper: listor, tupler, pid

```
> [1,2,a,[b]].
[1,2,a,[b]]
> [1 | [2 | [a | []]]].
[1,2,a]
> "abcd".
"abcd"
> [$a, $b, $c, $d].
"abcd"

> {state, 7}.
{state,7}

> self().
<0.26.0>
```

# SekvProg: mönstermatchning 1(3)

- Variabler binds bara en gång.
- Bunden variabel: matchning.
- Matchning endast ”åt vänster”.

```
A = 10
```
$\Longrightarrow$ Matchar, `A` binds till `10`

```
{B, C} = {10, klopp}
```
$\Longrightarrow$ Matchar, `B`$\mapsto$`10`, `C`$\mapsto$`klopp`

```
{D, D, E} = {xx, xx, 17}
```
$\Longrightarrow$ Matchar, `D`$\mapsto$`xx`, `E`$\mapsto$`17`

```
{F, F, G} = {xx, yy, 17}
```
$\Longrightarrow$ Matchar inte

# SekvProg: mönstermatchning 2(3)

`[H | T] = [1, 2, 3, 4]`

$\Longrightarrow$ Matchar, `H`$\mapsto$`1`, `T`$\mapsto$`[2, 3, 4]`

`[H1, H2 | Rest] = [1, 2, 3, 4]`

$\Longrightarrow$ Matchar, `H1`$\mapsto$`1`, `H2`$\mapsto$`2`, `Rest`$\mapsto$`[3, 4]`

`{ok, Result} = {ok, 10}`

$\Longrightarrow$ Matchar, `Result`$\mapsto$`10`

```
case file:open("flerp", [read]) of
    {ok, IoDevice} ->
         ...
    {error, Reason} ->
         ...
end
```

# SekvProg: mönstermatchning 3(3)

**Vad händer då det inte matchar?**

```
> A = 10.
10
> A = 30.
** exited: {{badmatch,30},[{erl_eval,expr,3}]} **
```

# SekvProg: Funktioner

```
count_elems([]) -> 0;
count_elems([Elem | Rest]) -> 1 + count_elems(Rest).

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

**Svansrekursiv variant:**

```
count_elems(L) -> count_elems(L, 0)

count_elems([], Acc) -> Acc;
count_elems([Elem | Rest], N) -> count_elems(Rest, N + 1).
```

# SekvProg: List comprehension

```
qs([]) ->
    [];
qs([Pivot | L]) ->
    qs([X || X <- L, X < Pivot]) ++
    [Pivot] ++
    qs([X || X <- L, X >= Pivot]).
```

`[X || X <- L, X < Pivot]` utläses: Listan av alla `X` sådana att `X` hämtas från listan `X` och det är sant att `X < Pivot`.

# SekvProg: Högre ordningens funktioner

```
qs(LT, []) ->
    [];
qs(LT, [Pivot | L]) ->
    qs(LT, [Elem || Elem <- L, LT(Elem, Pivot)]) ++
    [Pivot] ++
    qs(LT, [Elem || Elem <- L, not LT(Elem, Pivot)]).
```

```
> L = [{a,1}, {b,2}, {e,6}, {f,3}, {g,2}].
> F = fun({_,N}, {_,M}) -> N < M end.
> qs(F, L).
⇒ [{a,1}, {b,2}, {g,2}, {f,3}, {e,6}]
```

# SekvProg: Moduler 1(2)

```
-module(sort).
-export([qs/1, qs/2]).

qs([]) ->
    [];
qs([Pivot | L]) ->
    qs([X || X <- L, X < Pivot]) ++
    [Pivot] ++
    qs([X || X <- L, X >= Pivot]).

qs(LT, []) ->
    [];
qs(LT, [Pivot | L]) ->
    qs(LT, [Elem || Elem <- L, LT(Elem, Pivot)]) ++
    [Pivot] ++
    qs(LT, [Elem || Elem <- L, not LT(Elem, Pivot)]).
```

# SekvProg: Moduler 2(2)

```
> c("sort").
{ok,sort}
> sort:qs([1, 2, 5, 4, 6, 3]).
[1,2,3,4,5,6]

> sort:module_info(exports).
[{qs,1},{qs,2},{module_info,0},{module_info,1}]
```

- Flat modulrymd

## ParProg: spawn, !, receive

- Starta en process med `spawn`
- `spawn` returnerer en *process-id*
- Skicka meddelanden med `Pid ! Msg`
- Ta emot med `receive`
- "Send and pray" gäller, d.v.s. man är inte garanterad att meddelandet kom fram.

# ParProg: Klient-server

## Server

```
-module(sort_server).
-export([start/0, loop/0]).
start() ->
    spawn(?MODULE, loop, []).
loop() ->
    receive
        {Client, {qsort, L}} ->
            Client ! sort:qs(L),
            loop();
        stop ->
            done;
        X ->
            loop()
    end.
```

## Klient

```
L = [1, 2, 5, 4, 6, 3].
Server = sort_server:start().
Server ! {self(), {qsort, L}}.
receive
    Result ->
        {ok, Result}
after 10000 ->
    timeout
end.
```

# ParProg: Om processer

## Att tänka på angående spawn

- Funktionen som ska spawnas måste vara exporterad.
- `spawn` misslyckas aldrig.

## Registrera namn för processer

- Registrera ett namn för en process: `register(name, Pid)`
- Därefter är det möjligt att använda processens namn för att skicka meddelanden: `name ! msg`.
- Slå upp process-id:n för en process: `whereis(name)`.
- Se vilka processer som är registrerade: `registered()`.

# ParProg: Distribution

- En erlangnod motsvarar (oftast) en OS-process.
- En erlangprocess finns i en erlangnod.
- Nodnamn bestäms oftast när man startar noden.
- Ta reda på nodens namn: `node()`
- Starta en process på en annan nod:
  `spawn(Node, Mod, Fun, Args) -> Pid`.
- Utföra ett funktionsanrop på en annan nod:
  `rpc:call(Node, Mod, Fun, Args)`.
- Se till vilken nod en process-id hör: `node(Pid)`.

# ParProg: Distribution: synlighet

- Automatisk koppling till nätverket då man skickar meddelande eller dylikt till annan nod.
- Se ifall en annan nod lever:
  `net_adm:ping(OtherNode) -> pong | pang`.
- Direkt synlig hos alla noder i nätverket.
  Lista alla andra noder: `nodes() -> [Node]`

# Felhantering: catch och throw

`catch` fångar:

- programfel
- värden som kastats med `throw`

```
> Y = (catch 1 = 2).
> Y.
{'EXIT',{{badmatch,2},[{erl_eval,expr,3}]}}

case catch mod:fn(A1, A2) of
    {'EXIT', Reason} ->
        {programming_error, Reason};
    Value ->
        {ok, Value}
end
```

## Felhantering: Fånga fel från andra processer

1. Ställ in att fånga *exit-meddelanden*:
   `process_flag(trap_exit, true)`
2. Länka till process:
   `link(PidB)` eller `spawn_link(Mod,Fun,Args)`
3. Ta emot exit-meddelanden med `receive`

```
receive
    {'EXIT', ExitingPid, Reason} ->
        restart_worker(...)
        ...
end
```

# Felhantering: olika orsaker till exit-meddelanden

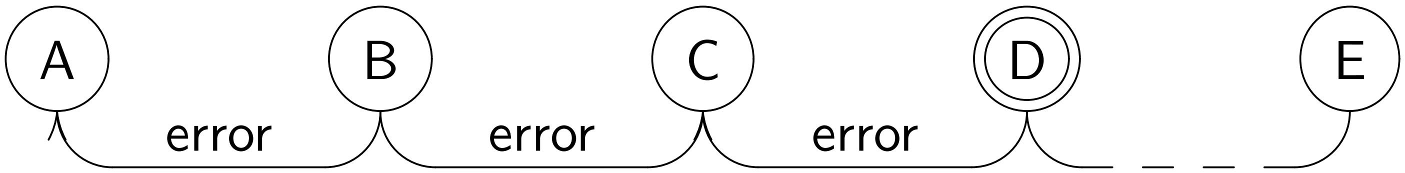
`normal` Processen fick slut på kod att exekvera.

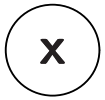
`killed` Processen har avslutats med `exit(Pid, kill)`. Då dör den ovillkorligen.
`exit(Pid, Reason)` skickar ett meddelande
`{'EXIT', FromPid, Reason}` till `Pid`.

**Annat** Programmeringsfel.

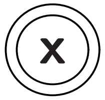
# Felhantering: felpropagering

Om man inte ställer in att fånga felmeddelanden med `process_flag(trap_exit, true)`, så kommer exit-meddelandet att propagera vidare till andra länkade processer.



är en process



är en process som gjort `process_info(trap_exit, true)`

# Felhantering: Sammanfattning

- `catch` fångar fel i egen process.
- `process_info(trap_exit, true)` + `link(Pid)` fångar fel i annan process.

# Felhantering: Sammanställning

| Fångar exit | | Reason normal | kill | Other |
|---|---|---|---|---|
| Nej | Mottagande process | Ignorerar | Dör | Dör |
| | Vidare-befordrar | - | killed | Other |
| Ja | Mottagande process | Tar emot {'EXIT',Pid,normal} | Dör | Tar emot {'EXIT',Pid,Other} |
| | Vidare-befordrar | - | killed | - |

## Behaviours – design patterns

**gen_server** Ramverk för en server.

**gen_event** Mekanism för prenumeration/utskick

**gen_fsm** Ramverk för en tillståndsmaskin.

**application** Ramverk för tillämpningsprogram. Hanterar konfigurering, start, stop, koduppgradering, nerstängning, processmigrering.

**supervisor** Ramverk för övervakning. Hanterar återstart då en övervakad enhet kraschar, övervakningshierarkier.

**supervisor_bridge** Övervakning av program som inte från början var skrivna för att ingå i ett övervakat system.

**Läs mer** om behaviours i Design Principles i webdokumentationen.

## Diverse: Macron (preprocessorjox)

```
-define(MY_CONSTANT, 20).
-define(TRACE, 1).
-ifdef(TRACE).
-define(PRINT(X), io:format("~p:~p: ~p~n", [?MODULE, ?LINE, X])).
-else.
-define(PRINT(X), do_nothing).
-endif.

fn() ->
    ?PRINT({"My constant:", ?MY_CONSTANT}).

> test:fn().
test:14: {"My constant:",20}
```

# Diverse: Bitsyntax: typer 1(3)

`<<Value:Size/TypeSpecifierList>>`

**TypeSpecifierList**

**Type** `integer`, `float`, `binary`

**Signed** `signed`, `unsigned`

**Endian** `big`, `little`

```
> <<X:32/little-signed-integer>> = <<1,2,3,4>>.
> X.
```

$\Rightarrow$ 67305985

# Diverse: Bitsyntax 2(3)

```
-module(z80cpu).
-export([execute/2]).

execute(Regs, <<2#01:2, R:3, S:3>>) ->
    %% LD r, s
    ld_r_s(Regs, r(R), r(S));
execute(Regs, <<2#11000110:8, N:8>>) ->
    %% ADD A, n
    add_r_n(Regs, a, N);
execute(Regs, <<2#10000:5, Reg:3>>) ->
    %% ADD A, r
    add_r_r(Regs, a, r(Reg)).

r(2#000) -> b;
r(2#001) -> c;
...
r(2#110) -> l;
r(2#111) -> a.
```

# Diverse: Bitsyntax: ett IP-paket 3(3)

```
-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, Rest/binary>> when HLen >= 5,
                                               4*HLen =< DgramSize ->
          OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
          <<Opts:OptsLen/binary, Data/binary>> = Rest,
          ...
    ... ->
          ...
end.
```

# Diverse: Ladda in kod i körande system

- Högst två kodversioner kan finnas samtidigt: en gammal och en ny.
- Den nya kodversionen anropas m.h.a. `modul:fn(Args)`.
- Om någon process, POld, kör den gamla versionen av en modul, och en ny version av modulen laddas in, så kommer POld att dö.
- Man får själv skriva rutiner för att uppgradera interna tillstånd i processer, databaser m.m.
- Application-behaviour:et m.fl. erbjuder ramverk för att uppgraderingsrutiner.

# Diverse: Ladda in kod i körande system: exempel

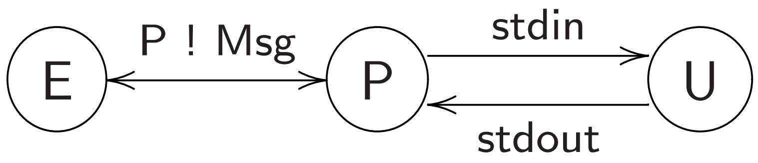
```
-export([loop/1, upgrade_state/2]).
-define(VERSION, {1,0}).

loop(State) ->
    receive
        {call, X} ->
            handle_call(X),
            loop(State);
        upgrade ->
            NewState = ?MODULE:upgrade_state(State, ?VERSION),
            ?MODULE:loop(NewState)
    end.

upgrade_state(OldState, {0,9}) -> OldState.
```

# Kopplina till andra språk

**JInterface** Erlang⟷Java

**ErlInterface** Erlang⟷C/C++

**Portar** Kommunicera via stdin/stdout till fristående OS-process.




**Inlänkad driver** En driver länkas dynamiskt in. Anrop från VM sker med vanliga C-funktionsanrop.

**CORBA** Kommunikation över IIOP.

# Verktyg

- HTTP-server
- ASN.1-kompilator, SNMP-toolkit
- yecc
- Orber (CORBA), COM-koppling
- Databaser: enkla minnes/disk-databaser, Mnesia, ODBC
  Mnesia: ”A heavy duty real-time distributed database”
- ErlInterface, JInterface
- Krypto: SSL, MD5, SHA, HMAC, DES
- GUI: Tcl/Tk, GS, Open-GL-koppling (separat bidrag)
- Debugger
- Profiler, Kodtäckningsanalysator, Korsreferensgenerator.

# Avslutning: Egna erfarenheter

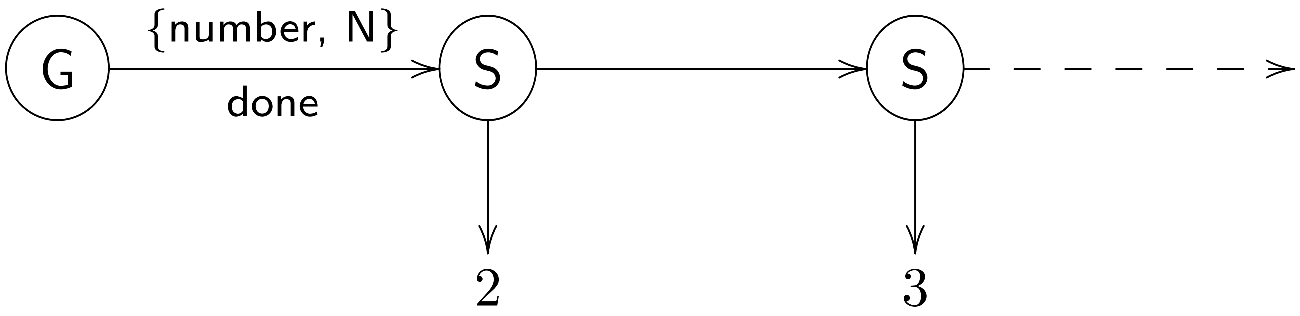
- Bra signal/brus-förhållande – lättläst kod på hög nivå
- Man ägnar sig mer åt själva problemet än i många andra språk.
- Parallellkörning är superenkelt
  - Privat minnesrymd
  - Ingen synkronisering/inga mutexar
  - Enkel och tydlig kommunikation mellan processer
  - Transparent distribution
- Håller måttet även i stor skala
- Hyfsat snabb exekvering (trots interpreterad bytekod)

## Avslutning: Läsa mer

- Open Source: `http://www.erlang.org/`
  mailinglista, FAQ, Ladda hem, Tutorials, Exempel
- Kommersiellt: `http://www.erlang.se/`
  Kurser, publikationer, support
- HiPE: `http://www.csd.uu.se/projects/hipe/`
  High performance Erlang – kompilerar Erlang till sparc v9-maskinkod
- Verification of Erlang Programs:
  `http://www.sics.se/fdt/Erlang/`
- ETOS: `http://www.iro.umontreal.ca/~etos/`
  Erlang→Scheme-kompilator
- Eddie: `http://eddie.sourceforge.net/`
  "a high availability clustering tool"

# Större exempel: Erathostenes såll

Idén:




```
-module(era).

-export([start/0, start/1, generator_init/2, sieve_init/1]).

start() ->
    start(100).

start(N) ->
    FirstSieve = start_sieve(2),
    start_generator(N, FirstSieve).
```

# Större exempel: Erathostenes såll: generatorn

```
start_generator(N, FirstSieve) ->
    spawn(?MODULE, generator_init, [N, FirstSieve]).

generator_init(N, FirstSieve) ->
    generator_loop(2, N, FirstSieve).

generator_loop(N, N, FirstSieve) ->
    %% Tell first sieve that we're done
    FirstSieve ! done,
    done;
generator_loop(CurrentNumber, N, FirstSieve) ->
    FirstSieve ! {number, CurrentNumber},
    generator_loop(CurrentNumber + 1, N, FirstSieve).
```

# Större exempel: Erathostenes såll: sållen

```
start_sieve(N) ->
    spawn(?MODULE, sieve_init, [N]).

sieve_init(N) ->
    io:format(" ~p", [N]),
    sieve_loop(N).

sieve_loop(N) ->
    receive
        done ->
            io:format("~n");     % exit
        {number, M} when M rem N == 0 ->
            %% N is a factor in M. Drop the number
            sieve_loop(N);
        {number, NewPrime} ->
            NextSieve = start_sieve(NewPrime),
            sieve_loop2(N, NextSieve)
    end.
```

# Större exempel: Erathostenes såll: fler såll

```
sieve_loop2(N, NextSieve) ->
    receive
        done ->
            %% Tell next sieve that we're done
            NextSieve ! done;    % exit
        {number, M} when M rem N == 0 ->
            %% N is a factor in M. Drop the number
            sieve_loop2(N, NextSieve);
        {number, M} ->
            %% N is not a factor in M. Send down the chain
            NextSieve ! {number, M},
            sieve_loop2(N, NextSieve)
    end.
```