**-spec concat(Lists :: [[T]]) -> [T].**

concat([]) -> [];

concat([[]]) -> [];

concat([H|Lists]) -> H ++ concat(Lists).

**-spec sumsquares(LON :: [number()]) -> number().**

sumsquares([]) -> 0;

sumsquares([Num | Rest]) -> (Num \* Num) + sumsquares(Rest).

**-spec map2(fun((A,B) -> C), [A], [B]) -> [C].**

map2(\_, [], \_) -> [];

map2(\_, \_, []) -> [];

map2(F, [A | As], [B | Bs]) -> [F(A, B) | map2(F, As, Bs)].

**-spec lookup([{Key, Value}], Key) -> [Value].**

lookup([], \_) -> [];

lookup([{K, Value} | Rest], Key) ->

case K of

Key -> [Value | lookup(Rest, Key)];

\_ -> lookup(Rest, Key)

end.

**mapInside(F, Llt) -> foldr(fun(Ls, Res) -> [map(F,Ls)|Res] end, [], Llt).**

**-spec count\_matching(Pred::fun((T) -> boolean()), Lst::list(T)) -> non\_neg\_integer().**

count\_matching(Pred, Lst) -> count\_matching(Pred, Lst, 0).

count\_matching(\_, [], Count) -> Count;

count\_matching(Pred, [Head|Rest], Count) ->

case Pred(Head) of

true -> count\_matching(Pred, Rest, Count + 1);

false -> count\_matching(Pred, Rest, Count)

end.

**-spec substaddr(SD::salesdata:salesdata(), New::string(), Old::string()) -> salesdata:salesdata().**

substaddr(#group{gname = Name, members = Memb}, New, Old) ->

if Memb =:= [] -> #group{gname = Name, members = []};

true -> #group{gname = Name, members = lists:map(fun (SD) -> substaddr(SD, New, Old) end, Memb)}

end;

substaddr(#store{address = Addr, amounts = Amnt}, New, Old) ->

if Addr =:= Old -> #store{address = New, amounts = Amnt};

true -> #store{address = Addr, amounts = Amnt}

end.

case proplists:is\_defined(Candidate, List) of

true -> [if Candidate == C -> {C, X + 1}; true -> {C, X} end || {C, X} <- List];

false -> [{Candidate, 1}] ++ List

end.

**-spec start() -> pid(). % stateless Power**

start() -> spawn(fun server/0).

-spec server() -> none().

server() ->

receive

{Pid, power, N, M} -> Pid!{answer, pow(N, M)}

end,

server().

**-spec start(State::integer()) -> pid(). % var server**

start(State) -> spawn(?MODULE, varserver, [State]).

-spec varserver(State::integer()) -> none().

varserver(State) ->

receive

{assign, NewVal} -> varserver(NewVal);

{Pid, fetch} -> Pid!{value, State},

varserver(State)

end.

**-spec start() -> pid(). % logger/noter**

start() -> spawn(?MODULE, log\_server, [[]]).

-spec log\_server(Logged::any()) -> none().

log\_server(Logged) ->

receive

{Pid, log, Entry} -> Pid!{self(), logged}, log\_server(Logged ++ [Entry]);

{Pid, fetch} -> Pid!{self(), log\_is, Logged},

log\_server(Logged)

end.

**-spec start(Running::integer()) -> pid(). % barrier synch**

start(Running) -> spawn(?MODULE, barrier\_synch, [Running, []]).

-spec barrier\_synch(Running::integer(), List\_pid::any()) -> none().

barrier\_synch(Running, List\_pid) ->

receive

{Pid, done} -> Pid!{self(), ok},

if (Running =< 1) ->

lists:foreach(fun(X) -> X!{self(), continue} end, List\_pid),

Pid!{self(), continue},

barrier\_synch(0, []);

true -> barrier\_synch(Running - 1, List\_pid ++ [Pid])

end;

{Pid, how\_many\_running} -> Pid!{self(), number\_running\_is, Running},

barrier\_synch(Running, List\_pid)

end.

**% Twitter server**

-spec start() -> pid().

start() -> spawn(twitterserver, tserver, [[]]).

-spec tweet(pid(), string()) -> tweeted.

tweet(SPid, TweetText) ->

io:fwrite("Tweeting.. ~n", []),

SPid ! {self(), tweet, TweetText},

receive

{\_, ok} ->

tweeted

end.

-spec fetch(pid(), non\_neg\_integer()) -> [string()].

fetch(SPid, N) ->

SPid ! {self(), fetch, N},

receive

{\_, here, TweetStrings} ->

TweetStrings

end.

-spec tserver([string()]) -> none().

tserver(Tweets) ->

receive

{Pid, tweet, TweetText} ->

Pid ! {self(), ok},

tserver([TweetText | Tweets]);

{Pid, fetch, N} ->

Pid ! {self(), here, lists:sublist(Tweets, N)},

tserver(Tweets)

end.

**% Compositor server**

start() -> spawn(compositor, cserver, [(fun(X) -> X end)]).

cserver(Fun) ->

receive

{Pid, apply\_to, Value} ->

Result = Fun(Value),

Pid ! {self(), value\_is, Result},

cserver(Fun);

{Pid, compose\_with, NewFun} ->

Pid ! {self(), composed},

cserver(fun(X) -> NewFun(Fun(X)) end);

{Pid, start\_over\_with, NewFun} ->

Pid ! {self(), started\_over},

cserver(NewFun)

end.

**% Semaphore Server**

start(N) -> spawn(semaphore, semserver, [N, []]).

semserver(N, Waitlist) ->

receive

{Pid, lock} ->

if

N > 0 ->

Pid ! {self(), go\_ahead},

semserver(N-1, Waitlist);

true ->

semserver(0, [Pid | Waitlist])

end;

{Pid, done} ->

Pid ! {self(), thanks},

if

Waitlist == [] ->

semserver(N+1, Waitlist);

true ->

lists:last(Waitlist) ! {self(), go\_ahead},

semserver(N, lists:droplast(Waitlist))

end

end.

**% Notification server**

start(Name) -> spawn(notifier, noteserver, [Name, []]).

noteserver(EName, Observers) ->

receive

{Pid, register} ->

Pid ! {self(), registered},

noteserver(EName, [Pid | Observers]);

{Pid, observers} ->

Pid ! {self(), Observers},

noteserver(EName, Observers);

{Pid, announce} ->

lists:foreach(fun(X) -> X ! {self(), {event, EName}} end, Observers),

Pid ! {self(), announced},

noteserver(EName, Observers)

end.

**% Shared Variable Server, *next page.***

start(Var) -> spawn(sharedvarserver, varserver, [Var]).

varserver(Var) ->

receive

{Pid, {run, F}} ->

NV = F(Var),

Pid ! {self(), {result, NV}},

varserver(NV);

{Pid, see} ->

Pid ! {self(), Var},

varserver(Var)

end.

**% Catalog Server**

-spec start() -> none().

start() -> spawn(catalogserver, cserver, [[]]).

-spec cserver(KeyList :: [any()]) -> none().

cserver(KeyList) ->

receive

{Pid, {associate, Key, Value}} ->

Pid ! {self(), ok},

cserver(lists:keystore(Key, 1, KeyList, {Key, Value}));

{Pid, {lookup, Key}} ->

Search = lists:keyfind(Key, 1, KeyList),

case Search of

false ->

Pid ! {self(), {value\_is, undefined}};

{\_, Val} ->

Pid ! {self(), {value\_is, Val}}

end,

cserver(KeyList)

end.

**% Average Stateless Server**

-spec start() -> pid().

start() -> spawn(average, avgserver, []).

-spec avgserver() -> none().

avgserver() ->

receive

{Pid, {average, LON}} -> Pid ! {self(), {average\_is, avg(LON)}}

end,

avgserver().

-spec avg(LON :: [number()]) -> number().

avg(LON) -> lists:sum(LON) / len(LON).

-spec len(LON :: [number()]) -> number().

len(LON) -> lists:sum([ 1 || \_ <- LON ]).

**% Election Server**

-module(electionserver).

-export([start/0, electionserver/1, vote/2, results/1, count\_vote/2]).

-spec start() -> pid().

start() ->

spawn(?MODULE, electionserver, [[]]).

-spec electionserver(List::list()) -> none().

electionserver(List) ->

receive

{Pid, Candidate} ->

Pid!{self(), vote},

electionserver(count\_vote(Candidate, List));

{Pid} ->

Pid!{self(), results, List},

electionserver(List)

end.

-spec count\_vote(Candidate::atom(), List::list()) -> list().

count\_vote(Candidate, List) ->

case proplists:is\_defined(Candidate, List) of

true ->

[if Candidate == C -> {C, X + 1}; true -> {C, X} end || {C, X} <- List];

false ->

[{Candidate, 1}] ++ List

end.

-spec vote(ES::pid(), Candidate::atom()) -> ok.

vote(ES, Candidate) ->

ES!{self(), Candidate},

receive

{ES, vote} -> ok

end.

-spec results(ES::pid()) -> [{atom(), non\_neg\_integer()}].

results(ES) ->

ES!{self()},

receive

{ES, results, List} -> lists:sort(List)

end.

**% Event detector**

-module(eventdetector).

-export([start/2, eventdetector/3]).

-spec start(InitialState::any(), TransitionFun::atom()) -> none().

start(InitialState, TransitionFun) ->

spawn(?MODULE, eventdetector, [InitialState, TransitionFun, []]).

-spec eventdetector(State::any(), TransitionFun::atom(), List::list()) -> none().

eventdetector(State, TransitionFun, Lst) ->

receive

{Pid, add\_me} ->

Pid ! {added},

eventdetector(State, TransitionFun, [Pid] ++ Lst);

{Pid, add\_yourself\_to, EDPid} ->

EDPid ! {self(), add\_me},

receive

{added} -> Pid ! {added}

end,

eventdetector(State, TransitionFun, Lst);

{Pid, state\_value} ->

Pid ! {value\_is, State},

eventdetector(State, TransitionFun, Lst);

Atom ->

{NewState, Event} = TransitionFun(State, Atom),

if

Event /= none ->

lists:foreach(fun(P) -> P ! Event end, Lst),

eventdetector(NewState, TransitionFun, Lst);

true ->

eventdetector(NewState, TransitionFun, Lst)

end

end.