**Erlang Syntax**

**String**

Characters: $a, $n

String: a list of integers  
"hello\7" : [104,101,108,111,7]

**Operators**

Arithmetic: +, -, \*, /, div (get Integer), rem (mod)

Equal value: ==, /= (!=), =:= (type, value), =/= (! ===)

Boolean: and, or, xor, not, andalso, orelss

**Lists**

synstax: [ ], [ head | remain ]

Operator: ++, --;  
eg: L ++ [aa]. ; L -- [aa].

**Function**  
start with lowercase letter  
fun(0) -> 1;  
fun(N) -> when N>0 -> N\*fun(N-1);  
fun(\_) -> others.  
% \_ represent don't care variable

**Print**  
io:format("~p~p",[Num1,Num2]).  
io:fwrite("~p ~p",[Num1,Num2]).

**Module Complie**

module name should be same like file name without extension  
.erl file  
-module(module*name).  
-compile(export*all).

**Type Check**  
is*atom/1  
is*function/1  
is*boolean/1  
is*record/1

**-spec**  
use this to define a function arguments' type and return type

-spec Function(Arguments\_type) -> RT.

-spec Function(Arguments::Type) -> RT.

**Record**  
data like json  
-record ( record\_name , { some\_field , some\_default = " yeah !", unimaginative\_name }).  
if record in .hrl file, this should be included in .erl file  
-include(module\_name.hrl).  
e.g:  
-record(robot, {name,type=industrial, hobbies, details=[ ] })  
#rebot{name="Mechatron", type = handmade,details = ["Moved by a samll man inside"]}.  
%access field  
variable#rebot.name

**Type**  
define a data structure more convenient than record  
-type btree()::{empty}|{node, term(),btree(),btree()}.

**Control Structures**

if

X > Y ->

true;

true -> % works as else branch

false

end

case expression of

value1 -> statement#1;

value2 -> statement#2;

valueN -> statement#N

end.

**-spawn**  
creates a new process and returns the pid.

spawn(Module, Name, Args) -> pid()

**Message Passing**  
use flush(). can get message from shell.  
PID ! msg is non-blocking, it will send message msg to process PID

Pid ! Message

% send multiple messages

Pid1 ! Message, Pid2 ! Message, Pid3 ! Message

Pid1 ! (Pid2 ! (Pid3 ! Message))

Pid1 ! Pid2 ! Pid3 ! Message

receive blocks until a message is available in the mailbox;

receive

Pattern1 when Guard1 ->

ToDo1;

Pattern2 when Guard2 ->

ToDo2;

\_Other ->

Catch\_all

after time->

timeout

% after part will triggered if time milliseconds have passed without receiving a message that matched the pattern

end

e.g:

-module ( echo ).

-export ([ start /0]).

echo () ->

receive

{From , Msg} ->

From ! { Msg },

echo ();

stop -> true

end .

start () ->

Pid = spawn ( fun echo /0) ,

% Returns pid of a new process

% started by the application of echo /0 to []

Token = " Hello Server !",

% Sending tokens to the server

Pid ! { self (), Token },

io: format (" Sent ~s~n",[ Token ]),

receive

{ Msg } ->

io: format (" Received ~s~n", [Msg ])

end ,

Pid ! stop .

% Stop server

make\_ref(). can get a global reference objects

**Semaphore**

-module(sem).

-compile(export\_all).

start\_sem(Init) ->

spawn(?MODULE,sem\_loop,[Init]).

sem\_loop(0) ->

receive

{release} ->

sem\_loop(1)

end;

sem\_loop(P) when P>0 ->

receive

{release} ->

sem\_loop(P+1);

{acquire,From} ->

From!{ack},

sem\_loop(P-1)

end.

acquire(S) ->

S!{acquire,self()},

receive

{ack} ->

done

end.

release(S) ->

S!{release}.

**Links**  
link(Pid)  
link(spawn(fun module\_name:fun\_name/N))  
unlink/1 can tear the link down

**counter**

-module(ex1).

-compile(export\_all).

start(N) ->

%% Spawns a counter and N turnstile clients

C = spawn(?MODULE ,counter\_server ,[0]),

[ spawn(?MODULE ,turnstile ,[C,50]) || \_ <- lists:seq(1,N)],

C.

counter\_server(State) ->

%% State is the current value of the counter

receive

{bump} ->

counter\_server(State+1);

{read,From} ->

From!State,

counter\_server(State)

end.

turnstile(\_C,0) ->

%% C is the PID of the counter, and N the number of

%% times the turnstile turns

done;

turnstile(C,N) when N>0 ->

C!{bump},

turnstile(C,N-1).

**print letter before number**

-module(barr).

-compile(export\_all).

start(N) ->

B = spawn(?MODULE,loop,[2,2,[]]),

spawn(?MODULE,client1,[B]),

spawn(?MODULE,client2,[B]),

ok.

% loop(N,M,L)

% the main loop for a barrier of size N

% M are the number of threads yet to reach the barrier

% L is the list of PID,Ref of the threads that have already reached the barrier

loop(N,0,L) ->

[ Pid!{ok,Ref} || {Pid,Ref} <- L ],

loop(N,N,[]);

loop(N,M,L) ->

receive

{From,Ref} ->

loop(N,M-1,[{From,Ref}|L])

end.

reached(B) ->

R = make\_ref(),

B!{self(),R},

Receive

{ok,R} ->

ok

end.

client1(B) ->

io:format("a~n"),

reached(B),

io:format("1~n").

client2(B) ->

io:format("b~n"),

reached(B),

io:format("2~n").

**promela**

active spawn a process type. active protype P(){}  
init is the first process that is activated  
runinstantiates a process

init {

n = 1;

atomic {

run P(1, 10);

run P(2, 15)

}

}

assert();

do

:: i > N -> break

:: else ->

sum = sum + i;

i++

od;

for (i : 1 .. N) {

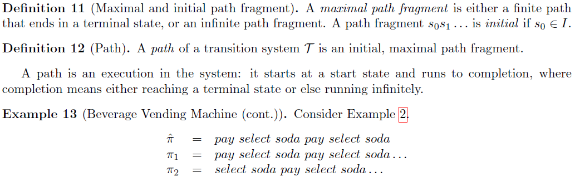
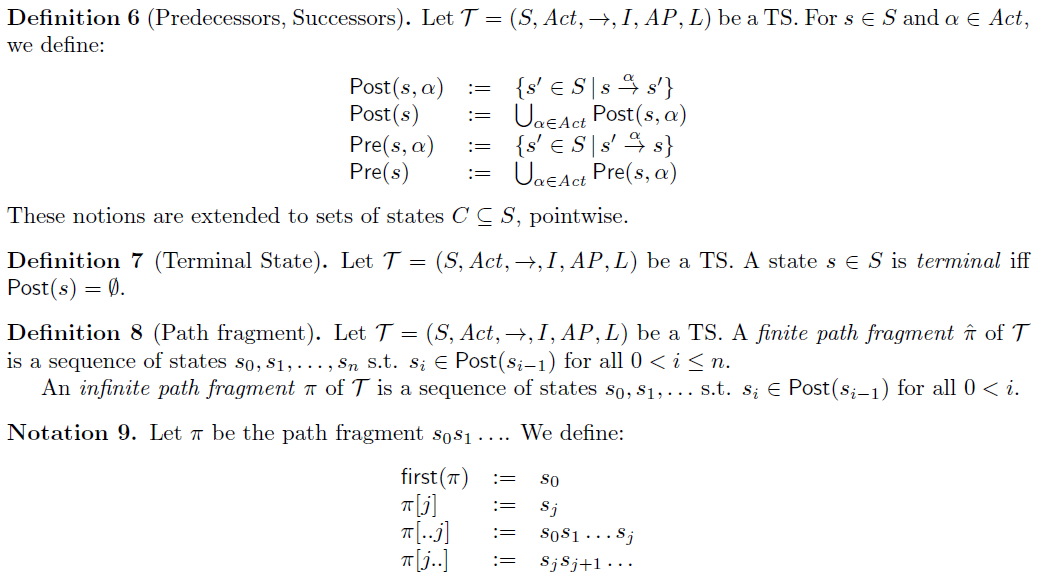
sum = sum +i;

}

**Transition Systems**

Graphical user interface, text, application

Description automatically generated

****