- In a sequential program all execution states are totally ordered
- in a concurrent program all execution states of a given actor are totally ordered
- The execution state of the concurrent program as a whole is partially ordered
- An execution is nondeterministic if there is a computation step in which there is a choice what to do next
- Nondeterminism appears naturally when there is asynchronous message passing
  - Messages can arrive or be processed in an order different from the sending order

Reference Cell

```
cell =
  rec(λb.λc.λm.if(get?(m),
  seq(send(cust(m),c),
     ready(b(c))),
  if(set?(m),
     ready(b(contents(m))),
     ready(b(c)))))
```

Using the cell:

```
let a = new(cell(0)) in
     seq(send(a,mkset(7)),
          send(a,mkset(2)),
          send(a,mkget(c)))

treeprod = rec(λf.λtree.
     if(isnat(tree),
     tree,
     f(left(tree))*f(right(tree))))
```

Agha, Mason, Smith & Talcott (AMST)

```
factnk = rec(\f.\n.\k.
  if (izZero?(k)),
    1,
    n * f(n - 1)(k - 1))

B_factnk = rec(\f.\d.
  let
    c = 1st(d),
    n = 1st(2nd(d)),
    k = 2nd(2nd(d))
  in
    seq(
      send(c, factnk(n)(k)),
      ready(f)))

B_join = \c.\a.ready(\b.send(c, a / b))

B_comb = \d.
  let
    c, n, k,
    join = new (B_join(c)),
    num = new (B_factnk),
    denom = new (B_factnk)
,  in
    seq(
      send(num, pr(pr(n, k), join)),
      send(denom, pr(pr(k, k), join)))
----------------------------------------
example = new (B_comb)
send(example, pr(stdout, pr(3, 3)))
```

```
Btreeprod =
  rec(λb.λm.
     seq(if(isnat(tree(m)),
        send(cust(m),tree(m)),
        let newcust=new(Bjoincont(cust(m))),
          lp = new(Btreeprod),
          rp = new(Btreeprod) in
         seq(send(lp,
          pr(left(tree(m)),newcust)),
          send(rp,
            pr(right(tree(m)),newcust)))),
     ready(b)))


Bjoincont =
     λcust.λfirstnum.ready(λnum.
          seq(send(cust,firstnum*num),
               ready(sink)))
```

$$\frac{e \rightarrow_\lambda e'}{\alpha, [\mathsf{R} \blacktriangleright e \blacktriangleleft]_a \parallel \mu \xrightarrow{[\mathbf{fun}:a]} \alpha, [\mathsf{R} \blacktriangleright e' \blacktriangleleft]_a \parallel \mu}$$

$$\alpha, [\mathsf{R} \blacktriangleright \mathtt{new}(b) \blacktriangleleft]_a \parallel \mu \xrightarrow{[\mathbf{new}:a,a']} \alpha, [\mathsf{R} \blacktriangleright a' \blacktriangleleft]_a, [\mathtt{ready}(b)]_{a'} \parallel \mu$$

a' *fresh*

$$\alpha, [\mathsf{R} \blacktriangleright \mathtt{send}(a', v) \blacktriangleleft]_a \parallel \mu \xrightarrow{[\mathbf{snd}:a]} \alpha, [\mathsf{R} \blacktriangleright \mathtt{nil} \blacktriangleleft]_a \parallel \mu \uplus \{\langle a' \Leftarrow v \rangle\}$$

$$\alpha, [\mathsf{R} \blacktriangleright \mathtt{ready}(b) \blacktriangleleft]_a \parallel \{\langle a \Leftarrow v \rangle\} \uplus \mu \xrightarrow{[\mathbf{rcv}:a,v]} \alpha, [b(v)]_a \parallel \mu$$

k_0 = [send($\square$,a) $\blacktriangleright$ new(b5) $\blacktriangleleft$]$_a$ $\parallel$ {}

k_1 = [send(b,a)]$_a$, [ready(b5)]$_b$ $\parallel$ {}

$$\mathbf{k_0} \xrightarrow{[\mathbf{new}:a,b]} \mathbf{k_1}$$

k_2 = [nil]$_a$, [ready(b5)]$_b$ $\parallel$ {< *b* <= *a* >}

$$\mathbf{k_1} \xrightarrow{[\mathbf{snd}:a]} \mathbf{k_2}$$

k_2 = [nil]$_a$, [ready(b5)]$_b$ $\parallel$ {< *b* <= *a* >}

k_3 = [nil]$_a$,

[rec(λy.λx.seq(send(x,5),ready(y)))(a)]$_b$

$\parallel$ {}

$$\mathbf{k_2} \xrightarrow{[\mathbf{rcv}:b,a]} \mathbf{k_3}$$

k_4 = [nil]$_a$, [seq(send(a,5),ready(b5)))]$_b$

$\parallel$ {}

$$\mathbf{k_3} \xrightarrow{[\mathbf{fun}:b]} \mathbf{k_4}$$

k_4 = [nil]$_a$,

[seq($\square$,ready(b5)) $\blacktriangleright$ send(a,5) $\blacktriangleleft$]$_b$

$\parallel$ {}

$$\mathbf{k_4} \xrightarrow{[\mathbf{snd}:b]} \mathbf{k_5}$$

k_5 = [nil]$_a$, [seq(nil,ready(b5))]$_b$

$\parallel$ {< *a* <= *5* >}

k_5 = [nil]$_a$, [seq(nil,ready(b5))]$_b$

$\parallel$ {< *a* <= *5* >}

k_6 = [nil]$_a$, [ready(b5)]$_b$ $\parallel$ {< *a* <= *5* >}

$$\mathbf{k_5} \xrightarrow{[\mathbf{fun}:b]} \mathbf{k_6}$$

Semantics example summary

k_0 = [send(new(b5),a)]$_a$ $\parallel$ {}

k_6 = [nil]$_a$, [ready(b5)]$_b$ $\parallel$ {< *a* <= *5* >}

$$\mathbf{k_0} \xrightarrow{[\mathbf{new}:a,b]} \mathbf{k_1} \xrightarrow{[\mathbf{snd}:a]} \mathbf{k_2} \xrightarrow{[\mathbf{rcv}:b,a]} \mathbf{k_3} \xrightarrow{[\mathbf{fun}:b]} \mathbf{k_4}$$

$$\mathbf{k_4} \xrightarrow{[\mathbf{snd}:b]} \mathbf{k_5} \xrightarrow{[\mathbf{fun}:b]} \mathbf{k_6}$$

This sequence of (labeled) transitions from k_0 to k_6 is called a *computation sequence*.

. (60 points) The *last president* game consists of figuring out who is the last president in a game where n citizens numbered $1, 2, \ldots n$ form a ring and where every k^{th} citizen takes a turn becoming the president and then leaves the ring.

A message passing protocol to solve the game consists of a message `president(i,s)` passed around the ring, where i counts how many non-president citizens have been traversed, and s is the total number of non-president citizens still in the ring. Initially, the message `president(1,n)` is given to the first citizen. When a citizen receives the message `president(i,s)`, it does the following:

(a) If it has not been president and `s = 1`, then it is the last president.

(b) If it has not been president and `(i mod k) = 0`, then it becomes president, and sends the message `president(i+1,s-1)` to the next citizen in the ring.

(c) If it has not been president, and `(i mod k)` $\neq$ `0`, then it sends the message `president(i+1,s)` to the next citizen in the ring.

(d) If it has been president, then it forwards the message `president(i,s)` to the next citizen (modeling that it has left the ring.)

Using either the SALSA or the Erlang programming language, write an actor program to simulate the *last president* game given n and k, and print the number of the winner.

Hint: You may write two behaviors, a `Game` behavior to create the citizens, connect them, and start the game; and a `Citizen` behavior to implement the message passing protocol.

```
citizen(I, K, N, P) ->
  receive
     {connect, C} ->
     if I == length(C) ->
          citizen(I, K,lists:nth(1,C),P);
     true ->
          citizen(I,K,lists:nth(I+1,C),P)
     end;
     {president, T, S} ->
     if
     (not P) and (S == 1) ->
          io:format("~p~n", [I]),
          citizen(I, K, N, P);
     (not P) and ((T rem K) == 0) ->
          N ! {president, T + 1, S - 1},
          citizen(I, K, N, true);
     (not P) and ((T rem K) /= 0) ->
          N ! {president, T + 1, S},
          citizen(I, K, N, P);
     P ->
          N ! {president, T, S},
          citizen(I, K, N, P)
     end
  end.

game(N, K) ->
  Citizens = [spawn(p, citizen, [I, K,
nil, false]) || I <- lists:seq(1, N)],
  [Citizen ! {connect, Citizens} ||
Citizen <- Citizens],
  lists:nth(1, Citizens) ! {president, 1,
N},
  io:format("").
```

```
-module(p).
-export([game/2, citizen/4]).
```

% Data structure for partitions

```
-record(partition, {id, nodes,
colors, edges}).
```

% completely empties the mailbox of the current actor.

```
flush_buffer() ->
   receive
      AnyMessage ->
     flush_buffer()
      after 0 ->
     true
   end.
```

SALSA

=============Main.salsa============

```
module practice;
import practice.Comb;
behavior Main {
   void act(String[] argv) {
      Comb c = new Comb();
      c <- get(4, 3) @ standardOutput <-
                              println(token);
   }
}
```

============FactNK.salsa===========

```
module practice;
behavior FactNK {
   int get(int n, int k) {
      int result = 1;
      for (int i = n - k + 1; i < n + 1; i++){
         result *= i;
      }
      return result;
   }
}
```

============Comb.salsa============

```
module practice;
import practice.FactNK;
behavior Comb {
   int divide(Object[] args) {
      return (Integer) args[0] / (Integer)
args[1];
   }

   int get(int n, int k) {
      FactNK numer = new FactNK();
      FactNK denom = new FactNK();
      join {
         numer <- get(n, k);
         denom <- get(k, k);
      } @ divide(token) @ currentContinuation;
   }
}
```