The exact syntax of your solution does NOT matter during the exam!

What matters is the correctness of the solution itself, and not whether you make a mistake with the language syntax!

For example, it doesn't matter if you write P(s) or ~~s.P~~ or ~~P[s]~~, as long as the P operation on semaphore s is the semantically correct operation at the place where you wrote it!

Concurrent statement:

```
co
      statement1;
||
      …
||
      statementN;
oc
```

Process:

```
process foo {
      …
}
```

A process declaration occurs at the syntactic level of a procedure/function declaration; it is not a statement, but `co … oc` *is a statement.*

An array of processes:

```
process bar[i = 1 to N] {
      …
}
```

These N processes are distinct processes, and they execute in an arbitrary order.

Array declaration:

```
int a[n];                 // uninitialized
int b[n] = (0, ..., 0);   // initialized with all zeros
int c[n] = ([n] 0);       // initialized with all zeros
```

Await statement:

```
<await (boolean_condition_here) list_of_statements; >
```

Example: `<await (x>0) x = x – 1; >`
Example: `<await(count>0); >` *// condition synchronization only – no statements, just a delay*

Atomic block = await statement without the `await` part:

```
<list_of_statements;>
```
Example: `< x = x + y; >`
Example: `< x = x + 1; y = y + 1; >`

Pre- and post-conditions:
```
{x == 0}  x = x + 1;  {x == 1}
```

Semaphores:
```
sem s;
sem another_semaphore = 1;
sem forks[5] = ([5] 1);       // array of semaphores
```

Operations on semaphores:
```
P(s);
V(s);
```

Monitors:

Example:
```
monitor MyMonitor {
     int x;
     cond cv;  // condition variable

     procedure do_something(int a) {
          …
     }
}
```

Declaration of a condition variable:

```
cond cv;
```

Operations on condition variables:

```
empty(cv)
wait(cv);
signal(cv);
```

Other operations – prepare a list before the exam!

Channels:

```
chan ch(type1 id1, …, typen idn);
```

Example: `chan input(int i);`
Example: `chan input(int);  // name of the parameter is omitted`

Primitives for channels:

send …
receive …
Recall what they mean before the exam!

CSP:

```
process A { … B!e; … }
process B { … A?x; … }
```

Guarded commands notation:
https://mitt.uib.no/courses/29697/files?preview=3634661

Guarded communication:

```
if B1; C1 -> S1;
[] …
[] Bn; Cn -> Sn;
fi

do B1; C1 -> S1;
[] …
[] Bn; Cn -> Sn;
od
```

Make sure you know before the exam how these work!

Modern CSP: prepare a small cheat sheet yourself

RPC:

```
module module_name
     // signatures of exported operations
body
     // variable declarations
     // initialization code
     // procedure bodies for exported operations
     // local procedures and processes
end module_name
```

Signature of an operation:

```
op op_name(formal_parameters) returns result_type
```

Body of an operation:

```
procedure op_name(formal_parameters) returns result_type
result_name {
     …
}
```

Calling a module's procedure:

```
call opname(arguments);
```

Rendezvous:

```
in op1(formal_params_1) and B1 -> S1;
[] …
[] opn(formal_params_n) and Bn -> Sn;
ni
```

Special data type:

```
queue of (type) name_of_queue;
```

Example:

```
queue of (int) q1;
queue of (int, string) q2;    // elements are tuples
```

Operations on queue: assume that any operations that you might need are defined. For example:

```
add_element_to(q1, 42);
remove_element_from(q2, last_element);
```