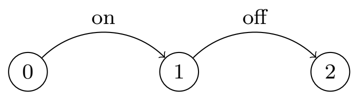
CPDS: LTS cheatsheet

Processes and actions

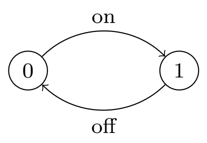
If x is an action and P a process then $(x \to P)$ describes a process initially engages in action x and behaves as described by P.

```
SWITCH_ONCE = (on -> off -> STOP).
```



Recursion

```
SWITCH = (on -> off -> SWITCH).
```



Trace

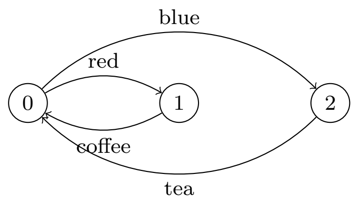
It represents one execution of a process.

```
on -> off -> on -> off -> on ->  off...
```

Choice

Given actions x and y then $(x \to P|y \to Q)$ describes a process that engages in either x or y and adopts the behaviour of P or Q respectively.

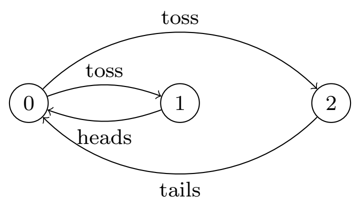
```
DRINKS = ( red -> coffee -> DRINKS
        | blue -> tea -> DRINKS).
```



Non-deterministic choice

Process $(x \to P|x \to Q)$ describes a process which engages in x then behaves P or Q. (Notice that the action, x, is the same. Which of the processes is run, P or Q, is randomly chosen)

```
COIN = (toss -> HEADS | toss -> TAILS),
HEADS = (heads -> COIN),
TAILS = (tails -> COIN).
```



Solution to class exercise: three DAYS

Copy the LTS code in LTSA to see the state diagrams.
`DAY1`: get up, then have tea, then go to work, then stop.
`DAY2`: do `DAY1` repeatedly.
`DAY3`: Do `DAY2` but choose between tea and coffee.
`DAY4`: Extends `DAY3` to include an `alarm` with a `snooze` button to be performed before getting up. Instead of getting up, you can snooze and go back to the start.

```
DAY1 = (up -> tea -> work -> STOP).

DAY2 = (up -> tea -> work -> DAY2).

DAY3 = (up -> DRINK),
DRINK = (tea -> WORK | coffee -> WORK),
WORK = (work -> DAY3).

DAY4 = (alarm -> ALARM),
ALARM = (up -> DRINK | snooze -> DAY4),
DRINK = (tea -> WORK | coffee -> WORK),
WORK = (work -> DAY4).
```

Indexed processes and actions

The following codes are equivalent.

```
BUFF = (in[i:0..3] -> out[i] -> BUFF).

BUFF = (in[0] -> out[0] -> BUFF
        | in[1] -> out[1] -> BUFF
        | in[2] -> out[2] -> BUFF
        | in[3] -> out[3] -> BUFF).
```

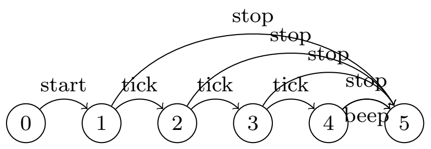
The parameters can be declared as `const` or `range` or as default for given processes.

```
const N = 1
range T = 0..N
range R = o..2*N

SUM = (in[a:T][b:T] -> TOTAL[a+b]),
TOTAL[s:R] = (out[s] -> SUM).

BUFF(N = 3) = (in[i:N] -> out[i] -> BUFF).
```

Guarded (conditional) actions

The choice (**when**B $x \to P$ $y \to Q$) means that x is only eligible when B is true.

```
COUNT(N = 3) = COUNT[0],
COUNT[i: 0..N] = (when (i < N) inc -> COUNT[i + 1]
                | when (i > 0) dec -> COUNT[i - 1]).

COUNTDOWN(N = 3) = (start -> COUNTDOWN[N]),
COUNTDOWN[i: 0..N] = (when (i > 0)
                        tick -> COUNTDOWN[i - 1]
                    | when (i == 0) beep -> STOP
                    | stop -> STOP).
```



Solution to class exercise: SENSOR

Model a sensor that measures the level of a tank. The level, initially 5, is measured from 0 to 9. It outputs `low` if the level is below 2, `high` if level is greater than 8, `normal` otherwise.

```
LEVEL(N=9) = SENSOR[5],
LEVEL[i:0..N] = (
        when (i < 2) low -> SENSOR[i]
        | when (i >= 2 && i < 8) normal -> SENSOR[i]
        | when (i >= 8) high -> SENSOR[i]),
SENSOR[i:0..N] = (
        when (i > 0) dec -> LEVEL[i - 1]
        | when (i < N) inc -> LEVEL[i + 1]).
```

Parallel composition

For processes P and Q, $(P||Q)$ represents the concurrent execution of P and Q.

```
ITCH = (scratch -> STOP).
CONVERSE = (think -> talk -> STOP).
||CONVERSE_ITCH = (ITCH || CONVERSE).
```

The previous code permits any of the following traces:

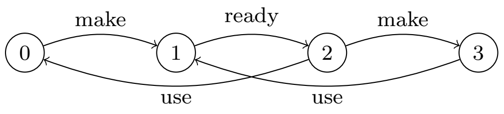
```
think->talk->scratch
think->scratch->talk
scratch->think->talk
```

Shared actions

If processes have actions in common, these actions are said to be shared. These must be executed at the same time by all processes that share it.

```
MAKER = (make -> ready -> MAKER).
USER = (ready -> use -> USER).
||MAKER_USER = (MAKER || USER).
```

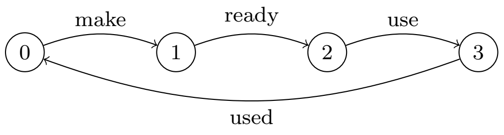
In the example above, the user can't `use` until the action `ready` happens. This can only happen after action `make`. Also, if `make` happens a second time, `ready` can not happen if the user has not `used` yet. This is, all concurrent processes must be synchronized in an action before the shared action for it to happen.



Handshake

```
MAKERv2 = (make -> ready -> used -> MAKERv2).
USERv2 = (ready -> use -> used -> USERv2).
||MAKER_USERv2 = (MAKERv2 || USERv2).
```

This code prevents the maker from keeping working if the thing it has made has not yet been used. Concurrent processes are forced to be synchronized. This is the handshake.



Class exercise: `||MICROWAVE`

Model the `MICROWAVE` using parallel composition, via handshaking with shared actions.

```
COOK = (put_food_in -> cook -> take_food_out -> COOK).
SET_HEAT = (put_food_in -> set_heat ->
        cook -> SET_HEAT).
SET_TIME = (put_food_in -> set_time ->
        cook -> SET_TIME).

||MICROWAVE = (COOK || SET_HEAT || SET_TIME).
```

Actually, the previous code forces you to put the food in before you can set the time and heat. We assume that is intended.

Process relabelling

Label $a : P$ gives one of the processes P the name a.

```
SWITCH = (on -> off -> SWITCH).
||TWO_SWITCH = (toilet:SWITCH || kitchen:SWITCH).
```

Processes can be labelled as an instance in an array.

```
||SWITCHES(N = 3) = (s[i:1..N]:SWITCH).
```

Finally, processes can be labelled by a set of prefix labels. `{a1,...,ax}::P` replaces every action label n with the labels `a1.n,...,ax.n`. For every transition $(n \to X)$ is replaced with transitions $(\{a1.n, ..., ax.n\} \to X)$.

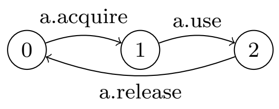
Class exercise: `RESOURCE_SHARE`

As an example of the prefix labels, see code below:

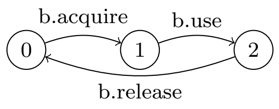
```
RESOURCE = (acquire -> release -> RESOURCE).
USER = (acquire -> use -> release -> USER).
||RESOURCE_SHARE = (a:USER || b:USER
                 || {a,b}::RESOURCE).
```

This translates to two independently operating users that share a resource.

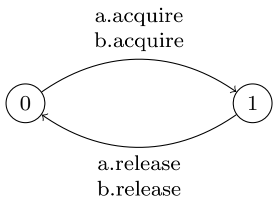
a:USER



b:USER

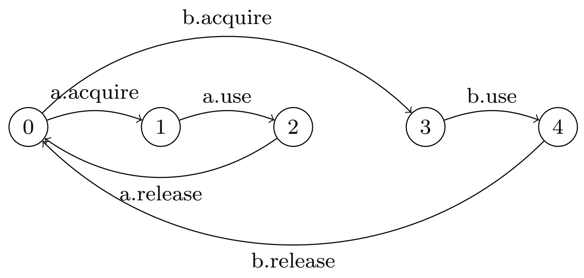


{a,b}::RESOURCE



The solution to the exercise would be the following LTS graph.

||RESOURCE_SHARE

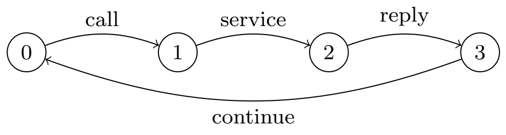


Action relabelling

Relabelling functions are applied to processes to change the names of action labels. It goes by the format `/{newlabel_1 / oldlabel_1, ..., newlabel_n / oldlabel_n}`.

```
CLIENT = (call -> wait -> continue -> CLIENT).
SERVER = (request -> service -> reply -> SERVER).
||CLIENT_SERVER = (CLIENT || SERVER) / {call / request,
                                       reply / wait}.
```

In this context, action `call` equals action `request` and action `reply` equals action `wait`. As can be seen on the animator, calling the action on the left hand side of the relabelling (`call` and `reply` in this case) will trigger the action on the right hand side (`request` and `wait`).

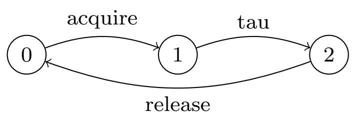


Action hiding

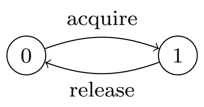
Hides an action x from process P alphabet. These actions still happen but silently. They are labelled `tau` and can not be shared between processes.

```
USER = (acquire -> use -> release -> USER) \{use}.
```

Hidden actions can be run using the `step` or `run` buttons in the animator.



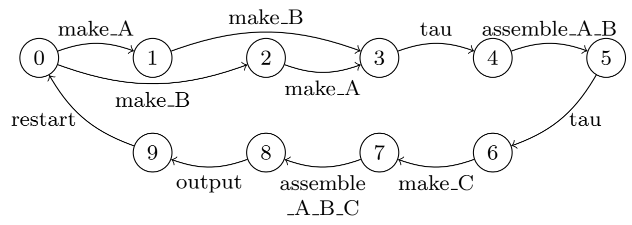
Minimized equivalent LTS graph:



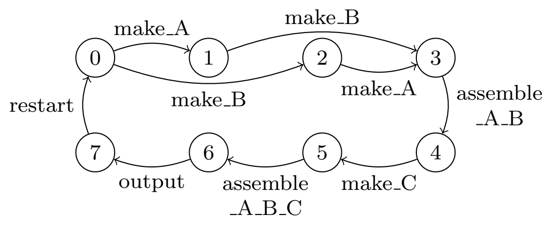
Class exercise: `||FACTORY`

```
MAKER_A = (make_A -> ready -> restart -> MAKER_A).
MAKER_B = (make_B -> ready -> restart -> MAKER_B).
ASSEMBLER_A_B = (ready -> assemble_A_B -> ready_two
                    -> ASSEMBLER_A_B).
ASSEMBLER = (ready_two -> make_C -> assemble_A_B_C
                    -> output -> restart -> ASSEMBLER).
||FACTORY = (MAKER_A || MAKER_B || ASSEMBLER_A_B
            || ASSEMBLER)
            \{ready, ready_two}.
```

Let's review all the concepts of LTS that coincide in this code.
`ready` and `ready_two` are hidden actions and will be shown as `tau` in the LTS graph. They are also shared actions, just like `restart`. This means that this will run as follows:
From the beginning, it is possible to choose between `make_A` and `make_B`. After both comes hidden `ready`, so it means that, regardless of the order, both `make` actions must happen before.
After `ready`, action `restart` from processes `MAKER_A` and `MAKER_B` can not be run until other processes reach it, as it is a shared action.
Now only `ASSEMBLER_A_B` is allowed to run the action `assemble_A_B`, which will allow hidden action `ready_two`.
Process `ASSEMBLER_A_B` can now begin again.
After action `ready_two`, process `ASSEMBLER` can run actions `assemble_A_B_C` and `output`, which will allow both `MAKER` processes and itself to run action `restart` and start again.
This is represented in the following graph:



The minimized version is:



Correctness

Let S be a sequential program.
Formula $\{P\}S\{Q\}$ is **partially correct** if every terminating computation of S that starts in state P terminates in state Q. (Does not mean all will terminate, but the ones that do will do it satisfying Q)
$\{P\}S\{Q\}$ is **totally correct** if every computation in S starts in state P and terminate in state Q. (All computations terminate, and all will end satisfying Q).
P is the precondition and Q the postcondition.

Safety

`CONVOY` and `CARS` model

```
const N = 3
range ID = 1..N

CAR = (enter -> exit -> CAR).

NOPASS1 = C_ENTER[1],
C_ENTER[i:ID] = ([i].enter -> C_ENTER[i % N + 1]).

NOPASS2 = C_EXIT[1],
C_EXIT[i:ID] = ([i].exit -> C_EXIT[i % N + 1]).

||CONVOY = ([ID]:CAR || NOPASS1 || NOPASS2).

||CARS = (red:CONVOY || blue:CONVOY).
```

The model above defines a car process that goes through actions `enter` and `exit`. `N` amount of cars are created per convoy with `[ID]:CAR`, which are identified by numbers from 1 to `N`, including the latter. This code equals `(1:CAR || 2:CAR || 3:CAR || ...)`.
Both `NOPASS` processes start off as `C[1]` [1]. This makes the first of a convoy the only one able to move. Moreover, actions `enter` and `exit` are shared across `[i]:CAR` and `C[i]`. This means that at the beginning, only `[1].enter` can be run, which will enable `[1].exit`, which will enable the next car to follow the same process, and so on. This is, this code does not allow overtaking.
Two concurrent convoys are created but no safety is implemented as to permitting traffic in only one way.

`BRIDGE` model

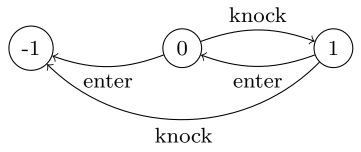
```
BRIDGE = BRIDGE[0][0],
BRIDGE[nRed:T][nBlue:T] =
    (when (nBlue == 0)
        red[ID].enter -> BRIDGE[nRed + 1][nBlue]
    | red[ID].exit -> BRIDGE[nRed - 1][nBlue]
    | when (nRed == 0)
        blue[ID].enter -> BRIDGE[nRed][nBlue + 1]
    | blue[ID].exit -> BRIDGE[nRed][nBlue - 1]).
```

The `BRIDGE` processes starts the bridge as empty, no car of any color is passing through. Its function is to count the amount of cars of each type on the bridge, allowing only cars of either.
Notice that the `when` clause only applies to the `enter` actions. This means that `exit` actions permit car counts to decrease below 0.

Safety properties

A safety property P defines a deterministic process that asserts that any trace in the alphabet of P is accepted by P. Those actions that are not part of the specified behaviour of P are transitions to the error state.
Unrelated example.

```
property POLITE = (knock -> enter -> POLITE).
```



Possible traces:

- `knock` → `enter` ✓
- `enter` ✗
- `knock` → `knock` ✗

Applied to the single lane problem.

```
property ONEWAY = (red[ID].enter -> RED[1]
        | blue[ID].enter -> BLUE[1]),
RED[i:ID] = (red[ID].enter -> RED[i+1]
        | when (i == 1) red[ID].exit -> ONEWAY
        | when (i > 1) red[ID].exit -> RED[i - 1]),
BLUE[i:ID] = (blue[ID].enter -> BLUE[i+1]
        | when (i == 1) blue[ID].exit -> ONEWAY
        | when (i > 1) blue[ID].exit -> BLUE[i - 1]).

||SingleLaneBridge = (CARS || BRIDGE || ONEWAY).
```

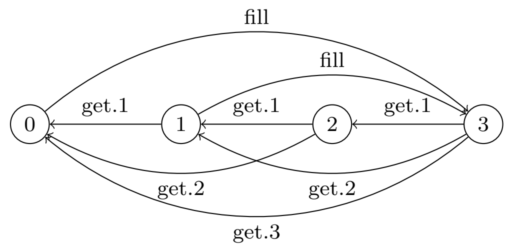
The addition of this safety property means that each time `ONEWAY` is reached the next action can only lead to either `RED[1]` or `BLUE[1]`. Anything else would trigger an error state.

[1] In the slides, `NOPASS1` and `NOPASS2` both initialize a homonym process `C[1]`. The two `C` processes are actually different despite their shared names and each works only within the scope of the bigger process, delimited by the dot (`.`). To avoid confusion, they have been renamed for the example

Class exercise: SUPERMARKET

```
const Min = 1
const Max = 3
SHELF = BOT[0],
BOT[i:0..Max] = (when (i > 0) get[k:1..i] -> BOT[i-k]
| when (i <= Min) fill -> BOT[Max]).

WORKER = (fill -> WORKER).

CLIENT = (get[1..Max] -> CLIENT).

||SUPERMARKET = (SHELF || WORKER || CLIENT).
```

Below, the LTS graph. For convenience, the number on the nodes represent the amount of bottles on the shelf.



Liveness

A liveness property asserts that something good eventually happens.

Progress

A progress property $P = \{a1, ..., an\}$ asserts that it in an infinite execution of a target system, at least one of the actions $\{a1, ..., an\}$ will be executed infinitely often.

Fair choice

If a choice over a set of transitions is executed infinitely often, then every transition in the set will be executed infinitely often.

```
COIN = (toss -> heads -> COIN
        | toss -> tails -> COIN).
```

If a coin is tossed an infinite number of times, we would expect that heads would be chosen infinitely often and that tails would be chosen infinitely often. These progress properties assure that is the case:

```
progress HEADS = {heads}
progress TAILS = {tails}
```

On the other hand, if the same progress properties were added to the code given below, the progress property `TAILS` could not be fulfilled.

```
TWOCOIN =  (pick -> COIN | pick -> TRICK),
TRICK = (toss -> heads -> TRICK),
COIN = (toss -> heads -> COIN | toss -> tails -> COIN).
```

This is because for an infinite amount of executions, and given the non-deterministic nature of the process, `tails` is not able to execute infinitely often, because one of the coins will never give tails.
Going back to the single lane bridge problem.

```
progress BLUECROSS = {blue[ID].enter}
progress REDCROSS = {red[ID].enter}
```

These progress properties aim to guarantee that blue cars enter (therefore end up crossing) infinitely often and red cars too. But it is possible to loop through the same type of cars: Enter one blue car, then another one, then the first exits, then a third enters. This would not be detected as progress violations. This is because fair choice means that eventually every possible execution will occur, including those in which cars do not starve.

Action priority

Action priority expresions describe scheduling properties.

High priority

High priority specifies a composition in which actions $\{a1, ..., an\}$ have higher priority than any other action in the process.

```
||C = (P || Q) << {a1, ..., an}
```

Low priority

Low priority specifies a composition in which actions $\{a1, ..., an\}$ have lower priority than any other action in the process.

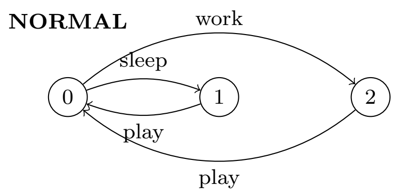
```
||C = (P || Q) >> {a1, ..., an}
```

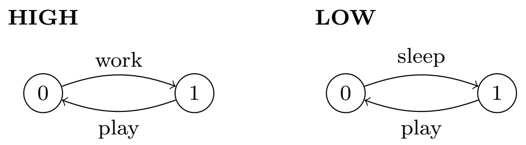
Both types of priority work by finding out which of the choices in the system include actions labeled by any of the labels included in the priority expressions. The lower priority ones (which could be actions $\{a1, ..., an\}$ marked as low priority or regular actions in comparison to high priority ones) are discarded.

Example

```
NORMAL = (work -> play -> NORMAL
        | sleep -> play -> NORMAL).
||HIGH = (NORMAL) << {work}.
||LOW = (NORMAL) >> {work}.
```

The code above would result in the LTS graphs below. Notice the differences.





Sources

Slides by Magee & Kramer

Unai Perez Mendizabal ©https://github.com/unaipme

CPDS: Erlang cheatsheet

You can run all the examples of this sheet with the Erlang shell (command `erl`).

Variables

Variables in Erlang start with a capital letter and are actually constant. Once assigned they can not be changed.

```
1> X = 123456789.
123456789
2> X.
123456789
3> X = 1234.
** exception error: no match of right hand sidevalue
 ↪   1234
```

All statements end with a point (.).

```
4> X * X
4> .
15241578750190521
```

Atoms

An atom is a literal, a constant with a name. Has no value by itself but can be used as a value.

```
1> foo.
foo
2> bar.
bar
```

They must start with a lower case letter and can not have other characters than alphanumeric characters, underscores (_) or @, unless they are enclosed in single quotes.

```
3> Length.
* 1: variable 'Length' is unbound
4> 'Length'.
'Length'
5> foo_bar@.
foo_bar@.
6> foo bar.
* 1: syntax error before: bar
```

Tuples

A single entity that groups a fixed amount of items is a tuple. It is analog to an object from OOP.

Creating tuples

```
1> FirstName = {firstName, joe}.
{firstName, joe}
2> LastName = {lastName, armstrong}.
{lastName, armstrong}
3> Person = {person, FirstName, LastName}.
{person,{firstName, joe},{lastName, armstrong}}
```

Reading from tuples

```
1> Point = {point, 10, 45}.
{point,10,45}
```

Given the `Point` variable created above, the numeric values can be extracted as follows:

```
2> {point, X, Y} = Point.
{point,10,45}
3> X.
10
4> Y.
45
```

Recall variables start with capital letter and atoms with lowercase. This means that, while `X` and `Y` are being used to store the values of the tuple, `point` is being used as a placeholder for pattern matching. The following line of code will not work:

```
5> {foo, X1, Y1} = point.
** exception error: no match of right hand side value
 ↪   {point,10,45}
```

Anonymous variable

If we are not interested in using atoms or pattern matching, certain part of the tuple's content can be discarded using the underscore (_) as anonymous variable.
Given the `Person` tuple created in the previous section, we can read the last name like this:

```
4> {_, {_, Name}, _} = Person.
{person,{firstName,joe},{lastName,armstrong}}
5> Name.
joe
```

Notice the underscore represents *something* we do not care about. The first two times it is used in the previous example, it is a placeholder for an atom. The third time, it is a placeholder for a whole tuple.

Functions and modules

An example on how to work with Erlang from outside the shell. We create a file with the following content:

```
-module(geometry).
-export([area/1]).
area({rectangle, Width, Height}) -> Width * Height;
area({circle, R}) -> 3.14159 * R * R.
```

`-module(<name>)` defines the name of the module that will include the functions declared in the file. The name must be the same as that of the file. With `-export([])`, we declare what functions we want to be accessible from outside the module itself (kind of like the `public` keyword in Java). Functions are listed in format `[name1/numArgs1,...]`, which permits the exportation of functions based on their name and the argument amount.

If a function receives multiple implementations, which is the case, these must be delimited by a semicolon (;), except for the last one, which must end with a point (.).
This definitely means that both implementations of `area` will be exported because both of them have just one argument. These arguments are in both cases tuples that allow for pattern matching as explained before: clauses beginning with lowercase are atom literals, and actual parameters otherwise. The module can now be loaded and used like this:

```
1> c(geometry).
{ok,geometry}
2> geometry:area({rectangle, 10, 5}).
50
3> geometry:area({circle, 1.4}).
6.15752
```

Class exercise: `weekDay`

```
-module(dates).
-export([classify_day/1]).

classify_day(monday) -> weekDay;
classify_day(tuesday) -> weekDay;
classify_day(wednesday) -> weekDay;
classify_day(thursday) -> weekDay;
classify_day(friday) -> weekDay;
classify_day(saturday) -> weekEnd;
classify_day(sunday) -> weekEnd.
```

Saved it as a file named `dates.erl` and opened a shell session in the same directory. Can be used as follows:

```
1> c(dates).
{ok,dates}
2> dates:classify_day(monday).
weekDay
3> dates:classify_day(saturday).
weekEnd
4> dates:classify_day(april).
** exception error: no function clause matching
 ↪   dates:classify_day(april) (dates.erl, line 4)
```

Class exercise: Factorial

```
-module(myfactorial).
-export([factorial/1]).

factorial(0) -> 1;
factorial(1) -> 1;
factorial(N) -> N * factorial(N - 1).
```

Notice that the least generic parameter definitions must be first, otherwise they will never be reached. For example, if `factorial(N)` was the first implementation, none of the other implementations would ever be run, because when calling `factorial(0)` it will get matched with `factorial(N)` before it gets the chance to match none of the others.
The file is saved as `myfactorial.erl`. Execution example:

```
1> c(myfactorial)
{ok,myfactorial}
2> myfactorial:factorial(5).
120
```

Concurrency basics

spawn

Creates a new concurrent process that runs the given function. Returns the process ID (PID), which can be used to send messages between processes.

```
Pid = spawn(Fun).
```

Message passing

By using the PID of the newly created concurrent process, we can send messages to it.

```
Pid ! Message.
```

Message reception

Messages passed using the previous method can be received with the keyword `receive`.

```
receive
    Pattern1 [when Condition1] -> Expression1;
    ...
    PatternN [when ConditionN] -> ExpressionN;
Other -> OtherExpression
end.
```

Concurrency example

```
-module(area_server0).
-export([loop/0]).

loop() ->
  receive
    {rectangle, Width, Height} ->
      io:format("Area of rectangle is ~p~n", [Width *
       ↪  Height]),
      loop();
    {circle, R} ->
      io:format("Area of circle is ~p~n", [3.14159 * R
       ↪  * R]),
      loop();
    Other ->
      io:format("The shape ~p is unknown~n", [Other]),
      loop()
  end.
```

How to run it:

```
1> c(area_server0).
{ok,area_server0}
2> Pid = spawn(fun area_server0:loop/0).
<0.67.0>
3> Pid ! {nothing}.
The shape {nothing} is unknown
{nothing}
4> Pid ! {circle, 5}.
Area of circle is 78.53975
{circle,5}
5> Pid ! {rectangle, 5, 8}.
Area of rectangle is 40
{rectangle,5,8}
```

Client-server

Notice that, even though the output of the spawned process can be seen, there is no direct interaction between the parent and child processes. The client-server paradigm implies the interaction between a client, which is responsible of starting the communication, and a server. This can be emulated in Erlang by passing the PID along with the message we send to the server:

```
-module(area_server1).
-export([loop/0, rpc/2]).

rpc(Pid, Request) ->
  Pid ! {self(), Request},
  receive ->
    Response -> Response
  end.

loop() ->
  receive
    {From, {rectangle, Width, Height}} ->
      From ! Width * Height,
      loop();
    {From, {circle, R}} ->
      From ! 3.14159 * R * R,
      loop();
    {From, Other} ->
      From ! {error, Other},
      loop()
  end.
```

This can be used like this from the shell:

```
1> c(area_server1).
{ok,area_server1}
2> Pid = spawn(fun area_server1:loop/0).
<0.67.0>
3> area_server1:rpc(Pid, {rectangle, 10, 5}).
50
4> area_server1:rpc(Pid, {triangle, 6}).
{error,{triangle,6}}
5> area_server1:rpc(Pid, {circle, 9}).
254.46879
```

This implementation comes with a problem: The `rpc/2` function will wait for any message, not just the message from the server with the appropriate response. This problem can be avoided by making this change to functions `rpc/2` and `loop/0`:

```
rpc(Pid, Request) ->
  Pid ! {self(), Request},
  receive
    {Pid, Response} -> Response
  end.

loop() ->
  receive
    {From, {rectangle, Width, Height}} ->
      From ! {self(), Width * Height},
      loop();
    {From, {circle, R}} ->
      From ! {self(), 3.14159 * R * R},
      loop();
    {From, Other} ->
      From ! {self(), {error, Other}},
      loop()
  end.
```

The change is not noticeable using the functions normally through the shell, but it enables security against interference.

Lists

Lists store variable amounts of elements. They can be created by enclosing the elements between brackets, delimited by commas.

Defining lists

```
1> Numbers = [1, 2, 4, 87, 3].
[1,2,4,87,3]
2> ShoppingList = [{apples, 10}, {pears, 6}].
[{apples,10},{pears,6}]
```

The first element of a list is the **head**, the rest is the **tail**. If `T` is a list, `[H|T]` is also a list, whose head is `H` and tail `T`. `T` must be a list, and `H` can, but does not need to. The vertical bar | separates them. `[]` is an empty list.

```
1> [a | [b, c] ].
[a,b,c]
2> [ [2] | [b, c] ].
[[2],b,c]
3> [2 | b, c ].
* 1: syntax error before: ','
3> [a | [] ].
[a]
4> [1, 2 | [3, 4, 5] ].
[1,2,3,4,5]
```

Reading from lists

Assuming a non-empty list `L`, expression `[X | Y] = L` will extract the head to `X` and the tail to `Y`. Both must be unused.

```
3> [ Bought | Remaining ] = ShoppingList.
[{apples,10},{pears,6}]
4> Bought.
{apples,10}
5> Remaining.
[{pears,6}]
6> [ Bought | ShoppingList ] = ShoppingList.
** exception error: no match of right hand side value
 ↪   [{apples,10},{pears,6}]
```

Adding or removing from a list

It is possible to remove a list from a list the expression `--` can be used. Specifically, in clause `L1 -- L2`, all elements from `L2` are removed from `L1`.

```
1> [1, 2, 3, 4, 5] -- [1, 3].
[2,4,5]
```

This can be used to remove a single element from the list.

```
2> N = 3.
3
3> L = [1, 2, 3, 4, 5].
[1,2,3,4,5]
4> L -- [N].
[1,2,4,5].
5> [H | T] = L.
[1,2,3,4,5]
6> T == L -- [H].
true
```

The expression `L1 ++ L2` can be used in a similar manner to add an element to list `L1`.

```
7> [1, 2, 3, 4] ++ [5, 6].
[1,2,3,4,5,6]
```

Notice that in both cases the order in which the lists are written matters.

```
8> [1, 2, 3, 4] ++ [5, 6].
[1,2,3,4,5,6]
9> [5, 6] ++ [1, 2, 3, 4].
[5,6,1,2,3,4]
10> [1, 2, 3, 4] -- [1, 3].
[2,4]
11> [1, 3] -- [1, 2, 3, 4].
[]
```

List processing functions

Some simple list processing functions are available in module `lists`. From the shell, they can be used by adding the prefix `lists:` to the functions. Below, it is explained how they are implemented.

```
sum([H | T]) -> H + sum(T);
sum([]) -> 0.
1> lists:sum([1, 2, 3, 4, 5]).
15
2> lists:sum([]).
0
```

Function `sum` will return the head added to the sum of the tail recursively. This will run until the tail is an empty list, in which case it will return 0.

```
map(_, []) -> [];
map(F, [H | T]) -> [F(H) | map(F, T)].
1> lists:map(fun(X) -> X + 1 end, [1, 2, 3, 4, 5]).
[2,3,4,5,6]
2> lists:map(fun(X) -> X * X end, [2 | [4, 8] ]).
[4,16,64]
3> lists:map(fun(X) -> 0.145 * X + 3.5774 * X * X end,
 ↪  []).
[]
```

The `map` function takes a fun (Erlang anonymous functions, this is, functions that are not stored nor associated to a name) as an input and a list. The function returns a list with the mapped head and recursively calls itself to map the tail. If the list is empty, it returns an empty list. This results in a list the elements of which have been transformed by means of the given fun.

```
member(H, [H | _]) -> true;
member(H, [_ | T]) -> member(H, T);
member(_, []) -> false.
1> lists:member(4, [1, 2, 3, 4, 5]).
true
2> lists:member(76, [1, 2, 3, 4, 5]).
false
```

The `member` function takes an element and a list, and checks whether the element is a member of the list. It is implemented using pattern matching. See the first implementation: If the head and the input element are the same, it returns true. If they are not the same, it recursively iterates through the tail. If the list is empty, then it is not a member.

```
reverse(L) -> reverse(L, []);
reverse([H | T], L) -> reverse(T, [H | L]);
reverse([], L) -> L.
1> lists:reverse([1, 2, 3, 4, 5]).
[5,4,3,2,1]
```

The `reverse` function reverses a list so that its first elements become the last and viceversa. It does so by taking just a list and turning it into two parameters, the list itself and a new empty list. Then, recursively, it will take the head of the original list and put it in the last position, until no more elements are left.

Strings and lists

If all elements in a list of integer numbers correspond to a printable character from the ASCII table, Erlang will print the list as a string instead of a number list.

```
1> [1, 2, 3, 66, 82, 69, 75, 33].
[1,2,3,66,82,69,75,33]
2> [66, 82, 69, 75, 33].
"BREK!"
```

Function `sep(L, N)` returns the division from element `N` of list `L` into two lists `L1` and `L2` such that the concatenation of `L1` and `L2` results in `L`.

```
sep(L, 0) -> {[], L};
sep([H | T], N) ->
  {L1, L2} = sep(T, N-1),
  {[H | L1], L2}.
1> sep([1, 2, 3, 66, 82, 69, 75, 33], 3).
{[1,2,3],"BREK!"}
```

Note: I have not been able to find an official implementation of function `sep`.

List comprehension

It is a construct to create lists based on existing lists. It works on a similar manner as `map`. It follows the set-builder notation. The part before the double vertical bar (`||`) is the output function applied to the output list obtained from the right hand side of the bars. Then the input list or lists have to be defined. The input is then followed by a set of predicates used to filter the input list and generate an output list.

```
1> [ X || X <- [1, 2, 3, 4], X > 1, X /= 3 ].
[2,4].
2> lists:seq(0,50).
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
 23,24,25,26,27,28|...]
3> [ X * 2 || X <- lists:seq(0,50), X * X > 3, X * X <
 ↪  50 ].
[4,6,8,10,12,14].
4> [ X + 1 || X <- lists:seq(0,15), X > 3, X < 14, X /=
 ↪  10].
[5,6,7,8,9,10,12,13,14]
```

If more than a list is defined, a cartesian product of all is generated.

```
5> [ {X + 1, Y} || X <- [1, 2, 3], Y <- [a, b, c] ].
[{2,a},{2,b},{2,c},{3,a},{3,b},{3,c},{4,a},{4,b},{4,c}]
6> [ X || X <- [1, 2, 3, 4] , Y <- [3, 4, 5, 6], X == Y ].
[3,4]
```

Permutation calculation

```
perm([]) -> [[]];
perm(L) -> [[H | T] || H <- L, T <- perm(L -- [H])].
```

The output list, formed by `[H | T]`, will be a cartesian product of every element on the input list `L` as the head and all possible recursive permutations of the remaining elements.

```
7> perm:perm([c, a, t]).
[[c,a,t],[c,t,a],[a,c,t],[a,t,c],[t,c,a],[t,a,c]]
```

Sequential quicksort

QuickSort is an algorithm for sorting. It can be implemented as follows in Erlang.

```
qs([]) -> [];
qs([H | T]) ->
    LT = [X || X <- T, X < H],
    GE = [X || X <- T, X >= H],
    qs(LT) ++ [H] ++ qs(GE).
```

When the list is not empty, it uses list comprehension to get the lists of smaller and greater or equal elements compared to the head `H`. Then recursively orders this lists and returns a concatenation of all ordered lists.

Execution time

Function `now()` returns `{MegaSecs, Secs, MicroSecs}` corresponding to elapsed time since 00:00 GMT January 1, 1970.
Function `apply(M, F [, Arg1,..., ArgN])` is equivalent to running function `F` from module `M` with arguments `Arg1,...,ArgN`.
Function `chrono(M, F, P)` returns time it takes function `M:F` with parameters `P` to run.

```
chrono(M, F, P) ->
    {_, Seconds, Micros} = now(),
    T1 = Seconds + (Micros / 1000000.0),
    apply(M, F, P),
    {_, Seconds2, Micros2} = now(),
    T2 = Seconds2 + (Micros2 / 1000000.0),
    T2 - T1.
```

The current module `M` can be resolved using `?MODULE`.

Straightforward parallelism

Here is an example of how to run a quick sort with a large list parallelly.

```
rcv(P) -> receive {P, X} -> X end.
```

This first function just handles the reception of the messages. As mentioned in the client-server section, it will just receive the messages coming from the process with PID `P`.

```
pqs2(P, L) ->
    if
        length(L) < 100000 ->
            P ! {self(), qs(L)};
        true ->
            [H | T] = L,
            LT = [X || X <- T, X < H],
            GE = [X || X <- T, X >= H],
            P1 = spawn(?MODULE, pqs2, [self(), LT]),
            P2 = spawn(?MODULE, pqs2, [self(), GE]),
            L1 = rcv(P1),
            L2 = rcv(P2),
            P ! {self(), L1 ++ [H] ++ L2}
    end.
```

Function `pqs2` is designed to be run parallelly, as it receives a PID `P` to which it sends the results. If the list `L` is not larger than a certain threshold (100000), it just runs the regular quick sort function defined in the previous section. Otherwise, it splits the list in two lists and runs itself recursively in child processes. Then waits for the result to arrive and sends it to the parent process.

```
pqs(L) ->
    P = spawn(?MODULE, pqs2, [self(), L]),
    rcv(P).
```

Finally, function `pqs` spawns a child `pqs2` process and then awaits the result.

Receive with a timeout

We can add the expression `after Time` to the receive clause for the process to stop waiting for a message if no matching message has arrived in within `Time` milliseconds.

```
receive
    Pattern1 [when Guard1] -> Expression1;
    ...
after Time ->
    Expressions
end.
```

This expression can be used to build a sleep function.

```
sleep(T) ->
    receive
    after T ->
        true
    end.
```

CPDS: Java concurrence cheatsheet

Mutual exclusion

Interference is the destructive update caused by the uncontrolled interleaving of read and write actions. The solution is to give methods mutually exclusive access to shared objects. This can be made in Java using the keyword `synchronized`.

```java
public class SynchronizedCounter {
    private int value = 0;

    public synchronized void increment() {
        int temp = value;
        try { System.sleep(400); } catch (Exception e) {}
        value += temp + 1;
    }
}
```

Only one thread will be able to access method `increment` at a given time. The keyword `synchronized` can also be used to create atomic blocks of code that mutually exclude threads on the use of an object.

```java
public class Turnstile extends Thread {
    private Counter counter;

    public Turnstile(Counter counter) {
        this.counter = counter;
    }

    @Override
    public void run() {
        while (true) {
            try { Thread.sleep(200); } catch (Exception
             ↪  e) {}
            synchronized (counter) {
                counter.increment();
            }
        }
    }

}
```

In the case above, the method `increment` needs not be synchronized, because its usage already is.

Monitors

The lack of coordination between threads can cause problems, for example with a bounded buffer. It has a fixed number of slots, so trying to put items when it is full or trying to read them when it is empty causes bugs. Monitors are the solution.

- Method `wait` makes the thread wait to be notified by another thread. The waiting thread does not retain the synchronization lock while waiting, it must try to acquire it when notified.
- Method `notify` wakes up a single thread that is waiting on the object.
- Method `notifyAll` wakes up all threads that are waiting on the object.

Class exercise `MICRO_ACCOUNT`

Consider a bank dealing with micro accounts. You can just deposit or withdraw one euro in each operation, and there is a bound of `M` for the max amount of money.
The monitor for `MICRO_ACCOUNT`:

```java
public synchronized void deposit() throws
 ↪  InterruptedException {
    System.out.println(Thread.currentThread().getName()
     ↪  + " tries to deposit");
    if (i == max) {
        System.out.println("Account is in max. " +
         ↪  Thread.currentThread().getName() + "
         ↪  exits");
    } else {

         ↪  System.out.println(Thread.currentThread().getName()
         ↪  + " deposits");
        i++;
        notifyAll();
    }
}

public synchronized void withdraw() throws
 ↪  InterruptedException {
    System.out.println(Thread.currentThread().getName()
     ↪  + " tries to withdraw");
    while (i == 0) wait();
    System.out.println(Thread.currentThread().getName()
     ↪  + " withdraws");
    i--;
}
```

Deadlock and timeout

Deadlock occurs in a system when all its threads are blocked and there are no eligible actions to perform. This can happen when the processes share resources used under mutual exclusion without preemption.
Timeout is a partial solution to deadlock. If a thread waits too much for a resource to be released, it can just stop waiting after a predefined amount of time.
