# Erlang Assignment for CPL 2014-15

**Abstract**

This document describes the Erlang assignment to be completed as part of the CPL course for the 2014-15 academic year. The assignment is to be completed individually; working in groups is not allowed (copying and plagiarising are not allowed either).

This document describes what the assignment consists of (Section 1) and how you have to deliver the assignment (Section 2).

## 1 Outline

This assignment is about implementing a fault-resistant, concurrent version of the popular game 2048 (Figure 1). For those who do not know 2048, these are


Figure 1: A screenshot from the online version of 2048.

some useful links, read them in order to familiarise yourself with the game:

- The game itself. Try not to play it too much, you have to implement it as well. http://gabrielecirulli.github.io/2048/

- A wikipedia description. http://en.wikipedia.org/wiki/2048_(video_game)

2048 is played on a 4×4 grid, with numbered tiles that slide when a player moves them. Every turn, a new tile will randomly appear in an empty spot on the board with a value of either 2 or 4. Tiles slide as far as possible in the chosen direction until they are stopped by either another tile or the edge of the grid. If two tiles of the same number collide while moving, they will merge into a tile with the total value of the two tiles that collided. The resulting tile cannot merge with another tile again in the same move. The game is won when a tile with a value of 2048 appears on the board.

Intuitively, each of the sixteen tiles that constitute the game grid must be implemented by an Erlang process called a `tile` process. Instead of sliding, the tiles will communicate and update their values based on each other's values. Moreover, tile processes will be randomly killed by a `blaster` process. Thus there must be a mechanism that identifies when these processes are killed so that another `tile` process is spawned to replace the killed one so that the game can continue seamlessly. You do not have to implement the whole game by yourself. A number of files are given to you, providing a starting point as described in Section 1.1. Your implementation duties are described in Section 1.2.

## 1.1 Starting Material

In Toledo, download the `Erlang-2048-assignment.zip` archive. Inside, you will find a number of files, here is a brief description of their functionalities:

**`main.erl`** This file contains two functions:

- `play/0` is used to run the game;
- `playnoblaster/0` is used to run the game *without the* `blaster` in order to test the correct implementation of `tiles` behaviour.

**`blaster.erl`** This file is not to be modified; it contains the `blaster` that kills `tile` processes.

**`gui.erl`** This file is not to be modified; it contains the implementation of a primitive, terminal-based GUI for basic interaction with the game (Figure 2). Instead of playing with arrows, the game is played either with `w a s d` or with `i j k l` keys. Each keystroke must be followed by the return key in order to be sent to the `gui` so that it is processed by the program. The `gui` itself is a process.

**`manager.erl`** This file can be modified; it provides a basic receive loop that forwards commands from the `gui` to the `tiles`. These modifications can be made to this file, as described in Section 1.2. Firstly, you should modify it to account for a process that monitors for killed tiles and respawns them. Secondly, you should change instruction `timer:sleep(700),` for more advanced concurrent functionalities. The `manager` itself is a process.

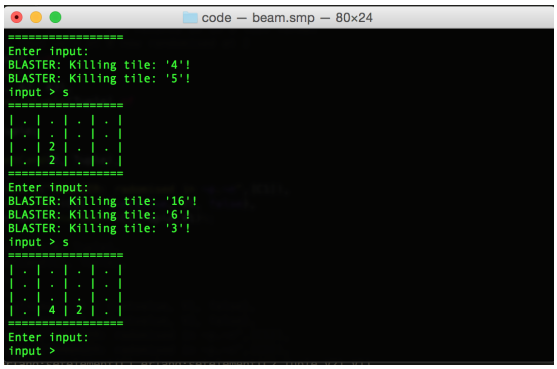


Figure 2: A screenshot from the Erlang GUI for 2048.

**`glob.erl`** This file is not to be modified; it contains global functions of general interest:

- `regformat/0` is the function to call to obtain the name to register a process with, given a number (the tile *Id*).
- `registerName/2` uses the `register/2` BIF to register a *pid*($2) with a *name*($1) but it ensures that that *name* is not registered yet.
- `indexof/2` returns the index of an *element*($1) in a *list*($2).
- `zeroesintuple/1` returns the number of 0es in a *tuple*($1).

**`debug.erl`** This file can be modified at will. Currently, the debugging flag is set to `false`. If that is set to `true`, the `debug/2` function can be used. This function is centralised so that if instead of writing to the console (the current debug method) you decide to write to a log, you have to modify a single line for that.

**`tile.erl`** This file provides a skeleton of the functionalities of a tile; it must be modified. These processes are not spawned by the code provided to you, you must edit it to account for `tile` spawning at the right place.

You can also add additional files to the code base.

## 1.2 Your Tasks

Your tasks for this assignment are subdivided in three points. Alongside the task description is how much it is worth in terms of the whole assignment.

**Task 1: tile behaviour (40%).** The goal of this task is to test your understanding of: (i) functional programming and (ii) some basics of the actor model. To test the correctness of this task, you can use the `main:playnoblaster/0` function.

You have to implement the `tile.erl` so that a tile behaves as it is supposed to in the game. To simplify your task, the behaviour of the tile has been summarised for you.

A tile has an *Id* telling its location on the grid (1 is the top-left and 16 is the bottom-right) as well as a *value* that tells what number it is being displayed on that tile. Finally, a tile has a boolean flag *merged* that tells whether the tile has been merged (explained below). Tiles can have other parameters, you are free to add those that you need.

The tile must be able to receive *at least* a number of messages (you can extend this set, not reduce it).

- `up, dn, lx, rx`. These are commands. Commands trigger tile behaviour as described below.
- `die`. This message causes the death of the tile.
- `{yourValue, Resp}`. This message causes the tile to answer to `Resp` its *value* and whether it is *merged*.
- `{set value, Future, NewMerged}`. This message updates the *value* and the *merged* flag of a tile to `Future` and to `NewMerged`.

**Tile behaviour**. Once a command is received, a tile moves its value as intended by that command. For example, when the `up` command is received, a tile moves its value upwards in the grid (if possible). To do so, a tile checks the values in all possible future locations reachable by the command. For example, the future locations for tile 9 for a command `up` are tiles 5 and 1 (Figure 3). The future location of tile 7 is 7.

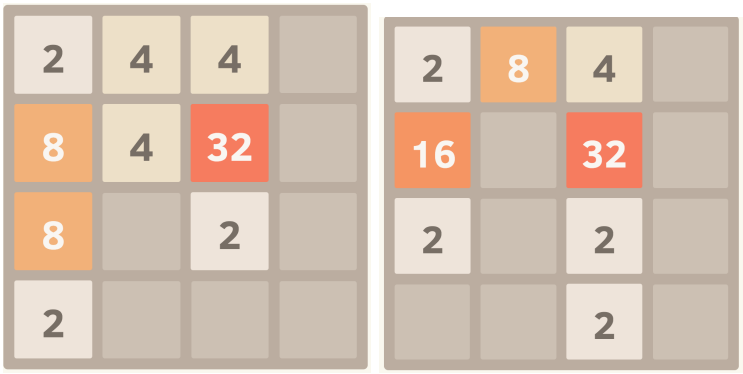


Figure 3: Execution of an `up` move.

To know the values of future locations, a tile sends all future locations a message `{yourValue, self()}`. Then, a tile detects the furthest location where it can move its value based on the responses it collects. A tile can move to a location if that location has not been merged this round and if

the value of the location is either 0 or it is equal to the tile's own *value*. Tiles cannot be skipped, so in Figure 3, tile 13 cannot send its value to tile 1.

Once assessed the furthest location that a tile can move to, a tile calculates the new value for that tile and then sends it a `{set value, Future, NewMerged}` message. There, `Future` is the new value that that tile will have and `NewMerged` is the boolean flag indicating whether the value is the result of a merge. In Figure 3, tile 9 sends message `{set value, 16, true}` to tile 5.

Finally, a tile updates its own *value*. If it caused another tile to update *value*, its new *value* becomes 0.

When updating other tiles it is important to know that their computations are finished, thus the propagation of commands is important.

**Command propagation.** Commands are initially sent to a row of tiles by the `manager`. Each tile is then responsible to perform the computations according to its behaviour, then to propagate the command to the next tile.

Given a command that makes the tiles move their values in a direction of the grid, that command is first sent to the last row of tiles for that direction. For example, the `up` command is sent to tiles 1, 2, 3 and 4. Once those tiles have executed, they propagate the command to the predecessor in the direction. For example, 1 propagates `up` to 5. Once done, 5 propagates `up` to 9 and so on.

Once the last row is reached, the command is not propagated. For example, once 13 receives `up`, it executes its behaviour but it does not send `up` to other tiles.

With these informations you can implement the behaviour of a `tile`.

**Hint:** instead of programming a general algorithm for the whole tile behaviour altogether do two things. Firstly, split that algorithm in subfunctions and intermediate steps whose correctness you can test. Secondly, write that algorithm only for a single direction of the game, then generalise it.

**Task 2: tile failure (40%).** The goal of this task is to test your understanding of failure in Erlang. To test the correctness of this task, you can use the `main:play/0` function.

Implement a mechanism that ensures that even when the `blaster` kills a `tile` processes, the game can be played anyway. You can modify the code in `manager.erl` to account for the new functionality. You are free to choose the way this is done, you will have to give a brief description of your solution in an accompanying document (described in Section 1.2).

**Task 3: tile concurrency (20%).** The goal of this task is to test your understanding of advanced concurrent communication in the actor model. To test the correctness of this task, you can use the `main:play/0` function.

Instruction `timer:sleep(700),` in `manager.erl` pauses the manager for 700 milliseconds, during which the tiles are supposed to finish their calculation. Then, the `manager` collects the data from the tiles and sends that data to the `gui` for showing.

Eliminate that instruction. The manager thus must wait for all tiles to have finished their calculation before spawning the `collector` that probes them for the values to display. Make sure the `manager` and the `tile`s communicate via message passing. Once all the computations are done, the `tile`s should inform the `manager` that it can collect its data. You are free to choose the way this is done, you will have to give a brief description of your solution in an accompanying document (described in Section 1.2).

# 2 Delivery and Important Dates

In order to complete the assignment, please deliver a single `.zip` archive through Toledo with:

- a folder `src` containing all source files;
- a single `.pdf` explanatory document (please use LaTeXwhen preparing documents). This document must not exceed two pages. It must describe the protocols used to solve Task 2 and Task 3. Describe what messages are exchanged by what processes and why.

Do not forget to write your name in all source files and in the document.

The deadline is **December 19th** at **23:59** Brussels time.

Good luck.