Step 1

```
- What can we say about readability of this code?
  What are the skills required to write such a program?

- Regarding the application domain, could you characterize
  the expressivity?
  The configurability of the code to change pins or behavior?
  Its debugging capabilities?

- Regarding the performance of the output code, what kind of
  parallelism is expressed by the use of the DDRx registers?

- What if we add additional tasks in the micro-controller code,
  with the same frequency? With a different frequency?
```

- The readability is not good, mainly due to the manipulation of the PINs using vectors of bits. The skills required are not very high provided you can write C and have the right documentation regarding the ports configuration and usage.
- The expressivity is the highest because we have access to all the possibilities of the machine, provided we have a documentation for all the system primitives. The configurability is very high The debugging is difficult due to the operations being low-level: if the program does not have the desired effects it could for example be due to a bit that was written to the wrong value, or a write operation that was not conducted due to a wrong writing mode and which would fail silently; all errors which are not on the same conceptual level as the desired output.
- The pins of a single register can all be modified at the same time with binary operations, which allows to jump from any state of the register to the other. This is a type of memory parallelism which affects the state of the components.
- It is possible to add any number of tasks on the same frequency due to the sequential nature of the code. Adding tasks with different frequencies would be difficult because it would require computations in order to synchronize different clocks. For example if we have a task T1 every 200ms and another T2 every 500ms, a possibility is to wait 100ms every task T1 in order to do T2. This is not scalable as is.

Step 2

```
- Is the readability problem solved?

- What kind of parallelism can still be expressed?

- Who is the public targeted by this "language"?

- Is this language extensible enough to support new features?
  What is the price for the developer?
```

- Not completely : it is now more easy the operations which require to turn on or off one PIN, but the more complex register states have become less understandable due to the lack of binary operations. However, the rest is more readable.
- The kind of register parallelism permitted by binary operators is not available anymore. What is left is some kind of algorithmic parallelism, for example by ligthing up common cathodes on the 7seg.
- This language is useful for people who want to code but not do the electric system and wiring part. Probably beginners in code with teachers doing the wiring and code-preparing part.
- This language is made of a few C++ constants and functions that interact in a more user-readable way with the Arduino system. It is infinitely extensible, since it is still C++.
- The developper has to read and understand a documentation that he has not written himself. Furthermore, as we saw, some things such as memory parallelization are now hidden to him. But in a general way, he gains readability and ease of use due to functions that are specific to the targeted system (an Arduino).

Step 3

Beforehand: When we did this step, the code proposed was using the Producer/Consumer implementation of a finite state machine. Even though it seemed a bit contradictory to the proposed model, after thinking about it we figured that we could see this the Producer/Consumer as an implementation to finite-state-automata and implemented it this way. The questions that follow have therefore been answered with the Producer/Consumer approach in mind (ii), and then answered with the functional model (i).

i) The code in `functional-version` is a re-implementation of this step, using the functional paradigm where states call each others. We tweak this a little by accepting that a state calls itself with different arguments, instead of defining all different states as separate functions.

ii) The code in `producer-consumer-version` is that first implementation. Since it comes second conceptually, we denote it with ii).

- `Does introducing a convention solve the readability issue?`

- `How to extend an app with a new feature?`
  `Does the approach prevent one to perform invasive changes`
  `in the existing behavior to introduce a new one?`

- `How to extend the code so that to support new features, e.g.,`
  `memory-less tasks, state-full tasks, different frequencies?`

- i) The readability improves in the part of the state functions because the behaviour can now be immediately identified as a state-machine is now immediate to see - at least for someone who is used to using these. However, the setup steps and different functions manipulating the led and the 7seg are difficult to understand without context.
- ii) The concept of finite state machine helps to understand easily the states of the components, but the implementation, with FLAGS in order to run different FSM in parallel, makes the code hard to comprehend.
- i) We can either add one state and transitions to it, which is quite non-invasive and easy to make. But more probably, it will be necessary to modify most of the states in order to introduce new features - which will be invasive and might break previous behaviour.
- ii) In order to add new features, it is easy to add new corresponding `consumer` and `producer` functions and just add them to the execution loop. The separation between inputs (consumers) and actions (producers) allows to perform changes to previous behaviors in a relatively safe way.
- i) Memory-less tasks require to add a similar branching option at the end of every state function, which branches to the state executing the memory-less task when triggered. This is painful, because every state needs to have this piece of code. It is possible to pass an argument `state` at every branching which would hold several state variables; and use it to execute state-full tasks. Different frequencies are a bit tricky. However, it is possible to implement them if by using a

single timer, and by passing a state containing timestamps of the last transition of every single finite-state-automaton being executed. This approach will be implemented (and more thoroughly described) in step4.

- ii) It is possible and easy to simply add new flags and message producers / consumers and run them in parallel to the other FSMs. This covers the case of memory-less tasks (no FLAGS) and state-full tasks (use all defined FLAGS). The case of different frequencies does not seem obvious to implement, because the algorithm which runs all the FSMs in parallel is a simple `while loop` i.e. there is no real concept of temporality inside our code except for the `delay` between each loop. In order to unify FSMs running on different frequencies, a temporal model, one with timers, may be required.

Step 4

```
- What are the pros/cons associated to the meta-modeling approach?
What is the cost of defining a meta-model?
What is difficult in this activity?

- From the user point of view, what does it change?
Is the approach usable for large apps?

- Consider the LED app and the counter one as two separate models.
Is it possible to automate the creation of the final app
based on these two models?

- What about the readability of the generated code compared
to the previous one "by hand"? Its debugging capabilities ?
Its extensiveness?

- Explain the interest of modeling in terms of genericity,
functional property verification.
```

- The main pros that seem to come with meta-modeling are: readability, modularity and robustness. The readability comes from the fact that all actions are classified (into classes and enumerations), with a clear view of the dependency between them thanks to the UML modeling of our system. The modularity is not a given but is more obtainable in a well-designed meta-model because, when the possibilities of the system (components, signals and actions) are clearly defined and in an independent manner, each part of them acquires a good reusability. For example, the actions

which allow to print numbers on the 7seg can be reused in any state of any fsm. Robustness is achieved because the executed code is no longer written by hand but generated. It means that most of the errors become logic errors (for example, by giving the wrong pin to an actuator, or setting up the wrong transition between two states) rather than language or syntaxic errors.

  - One drawback currently is that, after the meta-model is defined, translating it in Java classes is cumbersome as it requires to defined many new objects, sometimes in a way that is a bit repetitive - though this surely can be tackled by more abstraction. For example, the necessary use of Arrays.asList as soon as several elements have to be given is annoying. Maybe it comes from the fact that we are still using a GPU to write our model, with fixed parsing tactics that do not suit our needs. Another con is compatibility with other programs or libraries.
  - Time: it takes some time to define the meta-model, implement it, write it, etc. For the task at hand, it takes more time than it would have to write the code already in C. But the result surely is rewarding. It probably doesn't cost any expressivity, because the translation to code allows the use of any function, behavior etc. (for example, we could reuse the binary operators on registers from the Action class).
  - The main difficulty was to translate the meta-model into C code in a way that would scale to many elements and that would, obviously work. In particular, defining several state machines and then (this was our approach) fusing them to obtain a machine which runs them all the same time in a recursive way is a bit tricky.

- The user can now define its program in terms of components, sensors, interactions between them and planned behaviors. This behavioral description probably allows to focus on the functionality rather than the technicalities of the programming. Nothing seems to prevent it, and it is already reusable-as-is in some sense: if the code generation scales well, if the abstraction classes do not become bloated (though it is always possible to create new ones). Furthermore, the qualities of the generated code make it so that this is in fact surely the best option for large apps.
- It is indeed possible. Our approach was to model each of them as a Fsm (finite-state-automaton) class. The code generation machine then realizes an union of all the defined Fsm (by doing the product of the states into big single states, and unifying the transitions). This works well for two Fsm but have not tested it for more yet, and some more work is need to have full frequencies support.
- The generated code is actually quite readable and has a very regular syntax due to being generated. It is a bit verbose right now (28/09/18) due in part to not enough using abstractions for actions (defining functions action_%s

would greatly diminish verbosity for example). Its debugging capabilities are probably identical: there are more lines so it is difficult to read it as a whole, but the code patterns are regular and generic so an error is quickly picked up by the debugger and easily found. However, it is probably still bad in order to debug logic errors (wrongly defined pin, etc.) As said, it can be extensive (500 lines program as of now, but will probably be lessened soon).

- Modeling allows to defined classes and their interactions with other classes. This restrains their possibilities to specific actions and possibilities of realizations (types), which in turn allows to ask that some properties would be enforced on these structures. A first easy one was to specify the type of the variables inside the UML diagram: it ensured that these variables would be initialized to computable values. There are other structural properties that could be verified.

Step 6

```
- Who is the intended user for such a language?

- What is the cost of reusing this existing DSL for
the developer in terms of code?

- What is the cost of adding a new task of our domain?

- Was is the cost of adding a new hardware target?

- The Lustre language comes with its own ecosystem
(test, formal verification), what are the generic properties
we can imagine to prove from our domain?
```

- Lustre is clearly designed for users that want to work at the conceptual level of the program. The level of abstraction entirely gets rid of the underlying platform we are developing for. There are no longer pins, frequencies and signals. The behavior is only described in terms of dataflow and synchronous programming.
- Because of such an abstract description of application behaviors, code is highly reusable.
- However, the costly part remains bridging the gap between this abstract description and the specific implementation for our current domain. As have been experienced in this step of the Lab, this process is quite tedious. Coding the intended high-level behavior in lustre is a really smooth

experience, whereas glueing the generated C code from lustre with the domain-specific code can prove to be a tad painful. To be more specific to the question, adding a new task to the domain requires one to:

  - define the new abstract behavior in the lustre code
  - generate the new C code
  - implement the newly defined C routines to take into account the abstract behavior.
  - bridge that with the hardware lib.

- Adding another hardware target does not require one to fiddle with lustre anymore. The only thing that needs to be modified is the part of the C library which provides an high-level interface to the hardware. My personal assessment is that it is relatively easy to do.
- Working with a language with such an ecosystem can prove to be very useful when you are trying to prove properties on the execution of your program. One could for example prove that two values are forever alternating, and do this formally, without having to consider hardware.