Projects 2 and 3: Automated coffee machine!

Team project

Programming Languages and Environments
Department of Computer Science, NOVA FCT

May 5, 2025
Version β

Versions

α Initial version of the problem statement. (May 1, 2025)

β Typos, clarified code listings. (May 5, 2025)

Preamble

This assignment covers the second and third projects of the course. The second project is about the representation of a Petri net in OCaml, the third project is to implement the semantics of a Petri net, the firing of a transition and the analysis of the status of a network. To understand exactly what is requested on each phase, read along. The second project is worth 25% of the final grade and the third project is worth 50% of the final grade. The due date for both projects is the same, May 28, 2025. The projects are to be done in teams of two elements. The project is to be developed in OCaml and the code should be written in a functional style. You are not allowed to use any imperative constructs, such as mutable data structures or loops.

Introduction

Vending machines, and coffee machines in particular, are increasingly more automated and need precise and correct control systems. One of the most common control structures in automated machines is Petri nets[1]. This project is about the modelling of automated systems, like a coffee machine, using Petri nets. The goal is to implement a function that given a sequence of events, validates the events and returns the final state of the system. **Petri nets** are a mathematical modelling language used to describe the behaviour of systems. They consist of a graph with places, transitions, arcs, and tokens (see Figure 1). **Places**, depicted as circles in the diagram, represent the states of the system and they hold **Tokens** which are depicted as small black circles. Places can hold up to a maximum number of tokens, called the capacity of a place. **Transitions** are depicted as black thin rectangles and represent events that are triggered from the outside context and change the state of the system. **Arcs** are the edges of the graph, they connect places to transitions and transitions to places. Transitions define the flow of tokens through the network, which represent the current state of the system. The weight of the arcs determines how many tokens are taken/added in a given place. All of these attributes are configured when creating places, transitions, and arcs and depicted in the diagrams. If not stated in the diagram, the default capacity of a place is infinite and the number of tokens of a place is zero. The default number of consumed and produced tokens (weight of the arc) is one.

The firing of a transition is only possible when the transition is enabled. A transition is enabled when all the input places have the necessary number of tokens determined by the weight of the arcs

[1] https://en.wikipedia.org/wiki/Petri_net

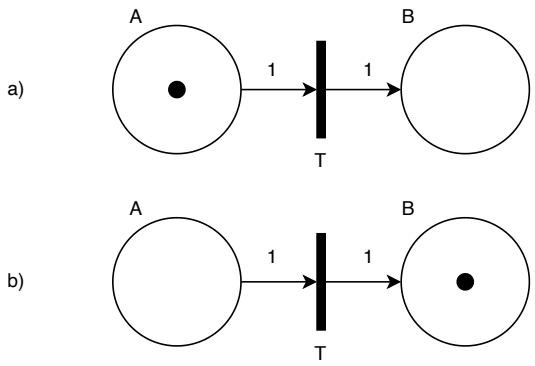

Figure 1: Petri net places, transitions, and tokens.

and all output places have the capacity to receive the number of tokens that each arc produces. When a transition fires, each arc consumes the prescribed number of tokens (in the arc) from its input places and produces the corresponding tokens in its output place. Recall, the number of tokens consumed and produced is defined by the weight of the arcs connecting the places and transitions. In Figure 1a), transition `T` can fire because there is a token in place `A`. When it fires it consumes one token from `A`, as specified by the number `1` in the outgoing arc from `A`, and produces one token in place `B`, as specified by the number `1` in the incoming arc to `B`. We refer to arcs as incoming and outgoing when referring to places. This example is represented in a test named `test_firing` in the test file `buildfire.ml` of the template repository.

Example: Coffee machine

When referring to the example of the coffee machine, consider the Petri net in Figure 2 with the following places: "Idle", "Coin Inserted", "Choice Made", "Water Heated", "Grain Ground", "Coffee Ready"; and the following transitions "insert coin", "make choice", "heat water", "grind coffee", "brew coffee", "remove cup". Notice that by convention we use lower case for transitions and upper case for places. Transitions are events that can occur in the system that can be triggered by the user (*e.g.*, pressing a button) or internally by the machine itself (*e.g.*, heating water). In this coffee machine, the "Idle" place has a capacity of one token, meaning that the machine can only be idle or busy, and since there is only one token in the system, can only attend one request at a time. By triggering the transition "insert coin", the token is transported to place "Coin Inserted", indicating that the machine is now ready to accept a choice. The transition "make choice" can be triggered at this point, moving the token to "Choice Made". Notice that two tokens will be placed in this place, allowing two transitions pointed by outgoing arcs to be triggered. The transitions "heat water" and "grind coffee" are enabled at this point. Since the places "Water Heated" and "Grain Ground" have a maximum capacity of one token, it cannot heat the water twice (same for grind coffee). The transition "brew coffee" can be triggered when both places "Water Heated" and "Grain Ground" have a token in them. Finally, the transition "remove cup" can be triggered to remove the cup and return to the initial state. Petri nets are a powerful tool for modelling and analysing the behaviour of concurrent systems. This behaviour can be observed in the tests included in file `coffeemachine.ml` in the provided test folder.

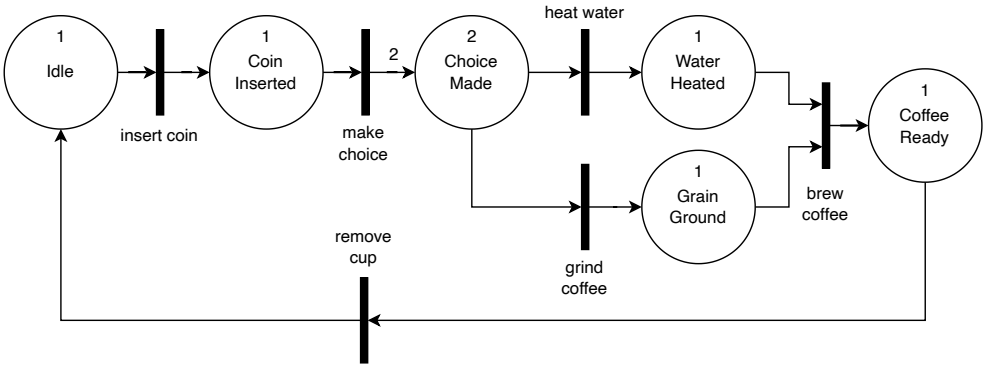

Figure 2: A Petri Net for a simple coffee machine.

Problem statement

Your mission is to represent a Petri net in OCaml, by implementing module `PetriNet` whose signature is splitted across Figure 3 and Figure 4, regarding Project 2 and Project 3 respectively. This module signature specifies an Abstract Data Type that exposes an opaque type `petri_net` and functions that allow the construction of Petri nets. Your assignment is to implement all the functions in the module and test them with the examples given. This signature is provided in the file named `petrinet.mli` in the template repository. The signature includes a brief description of each function.

Project 2

Project 2 includes all the definitions, but the functions `fire`, `fire_all`, and `is_stuck`, as shown in Figure 3. All the operations that build a Petri net will be tested to grade this part of the assignment. The operations that build a Petri net are: `empty`, `mk_place`, `mk_transition`, `mk_outgoing_arc`, `mk_incoming_arc`, `add_place`, `add_transition`, and `add_arc`. The operations that query the state of a Petri net are: `is_enabled`, `marking_of_place`, `places`, `transitions`, and `arcs`.

Project 3

Project 3 includes the implementation of the functions `fire`, `fire_all`, and `is_stuck`, as shown in Figure 4. These functions implement the semantics of a Petri net. The function `fire` performs a transition in the Petri net, moving tokens from places to places. The function `fire_all` performs a sequence of transitions in the Petri net. The function `is_stuck` checks if the Petri net is stuck, meaning that there are no enabled transitions.

```OCaml
type petri_net
type place
type transition
type arc
(** [empty] is an empty petri net. *)
val empty : petri_net

(** [mk_place name] creates a place with the given name. *)
val mk_place : string → place

(** [mk_transition name] creates a transition with the given name. *)
val mk_transition : string → transition

(** [mk_outgoing_arc p t w] creates an arc from place [p] to transition [t] with weight [w]. *)
val mk_outgoing_arc : place → transition → int → arc

(** [mk_incoming_arc t p w] creates an arc from transition [t] to place [p] with weight [w]. *)
val mk_incoming_arc : transition → place → int → arc

(** [add_place p c t net] adds place [p] to the petri net [net].
     The place is initialized with capacity [c] and initial token count [t].
     If a place with the same name already exists, it is replaced. *)
val add_place : place → int → int → petri_net → petri_net

(** [add_transition t net] adds transition [t] to the petri net [net].
     If a transition with the same name already exists, it is replaced. *)
val add_transition : transition → petri_net → petri_net

(** [add_arc a net] adds arc [a] to the petri net [net].
    Returns None if the arc is not well defined in the petri net. *)
val add_arc : arc → petri_net → petri_net option

(** [is_enabled t net] checks if transition [t] is enabled in the petri net [net]. *)
val is_enabled : transition → petri_net →  bool

(** [marking_of_place p net] returns the number of tokens in place [p] in the petri net [net].
    Returns None if the place does not exist in the petri net. *)
val marking_of_place : place → petri_net → int option

(** [places net] returns the list of places in the petri net [net]. *)
val places : petri_net → place list

(** [transitions net] returns the list of transitions in the petri net [net]. *)
val transitions : petri_net → transition list

(** [arcs net] returns the list of arcs in the petri net [net]. *)
val arcs : petri_net → arc list
```

Figure 3: Module signature for a Petri net regarding Project 2.

```
(** [fire t net] fires the transition [t] in the petri net [net] if it is enabled,
    returning the resulting petri net. Returns None if the transition does not exist or
    is not enabled. *)
val fire : transition → petri_net → petri_net option

(** [fire_all ts net] fires the transitions in the list [ts] in the petri net [net] in sequence,
    returning the resulting petri net. Returns None if any of the transition cannot be fired. *)
val fire_all : transition list → petri_net → petri_net option

(** [is_stuck net] checks if the petri net [net] is stuck (if there are no enabled transitions). *)
val is_stuck : petri_net → bool
```
OCaml

Figure 4: Module signature for a Petri net regarding Project 3.

Representation of a Petri net (Project 2)

In this section, we show how to represent a Petri net in OCaml. The representation is based on the module signature in Figure 3. The example of the coffee machine in Figure 2, is built as follows.

First, we use functions `mk_place` and `mk_transition`, to create all the values for places and transitions.

```
let idle = mk_place "Idle"
let coin_inserted = mk_place "Coin Inserted"
let choice_made = mk_place "Choice Made"
let water_heated = mk_place "Water Heated"
let grain_ground = mk_place "Grain Ground"
let coffee_ready = mk_place "Coffee Ready"

let insert_coin = mk_transition "insert coin"
let make_choice = mk_transition "make choice"
let heat_water = mk_transition "heat water"
let grind_coffee = mk_transition "grind coffee"
let brew_coffee = mk_transition "brew coffee"
let remove_cup = mk_transition "remove cup"
```
OCaml

Then we use `empty`, `add_place` and `add_transition` to add places and transitions to an empty Petri net.

```
let p0 = empty
       |> add_place idle 1 1
       |> add_place coin_inserted 1 0
       |> add_place choice_made 2 0
       |> add_place water_heated 1 0
       |> add_place grain_ground 1 0
       |> add_place coffee_ready 1 0

       |> add_transition insert_coin
       |> add_transition make_choice
       |> add_transition heat_water
       |> add_transition grind_coffee
       |> add_transition brew_coffee
       |> add_transition remove_cup
```
OCaml

Adding arcs requires dealing with errors which are modelled with values of type `option`. To use the same style (with pipes) we need to define a special pipe operator that ignores the next computation

when the previous one returns `None`. This is done with the operator `|>>`, defined in the template repository in a module called `Utils`, shown below.

```
let (|>>) x f = match x with                                    OCaml
  | None → None
  | Some y → f y
```

Now, the construction of the Petri net is concluded by adding all the arcs.

```
let pi = p0 |> add_arc (mk_outgoing_arc idle insert_coin 1)     OCaml
            |>> add_arc (mk_incoming_arc insert_coin coin_inserted 1)
            |>> add_arc (mk_outgoing_arc coin_inserted make_choice 1)
            |>> add_arc (mk_incoming_arc make_choice choice_made 2)
            |>> add_arc (mk_outgoing_arc choice_made heat_water 1)
            |>> add_arc (mk_incoming_arc heat_water water_heated 1)
            |>> add_arc (mk_outgoing_arc choice_made grind_coffee 1)
            |>> add_arc (mk_incoming_arc grind_coffee grain_ground 1)
            |>> add_arc (mk_outgoing_arc water_heated brew_coffee 1)
            |>> add_arc (mk_outgoing_arc grain_ground brew_coffee 1)
            |>> add_arc (mk_incoming_arc brew_coffee coffee_ready 1)
            |>> add_arc (mk_outgoing_arc coffee_ready remove_cup 1)
            |>> add_arc (mk_incoming_arc remove_cup idle 1)
```

At this point, it is possible to check the status of the Petri net. For example, we can check if the transition "insert coin" is enabled. Since `pi` is now a value of type `option` we need to define a function that checks if the transition is enabled.

```
  let is_enabled_opt p t =                                      OCaml
    match p with
      None → failwith "Error in construction"
      Some p → if is_enabled insert_coin p then
                 Printf.printf "Enabled"
               else
                 Printf.printf "Not enabled"
```

We can now check if the transition "insert coin" is enabled by calling

```
is_enabled_opt pi insert_coin                                   OCaml
```

Executing a Petri net (Project 3)

Given a properly built Petri net, a sequence of firing events can happen. In this case, we ordered two coffees in a row.

```
let pf = pi |>> fire insert_coin                                OCaml
            |>> fire make_choice
            |>> fire heat_water
            |>> fire grind_coffee
            |>> fire brew_coffee
            |>> fire remove_cup
            |>> fire insert_coin
            |>> fire make_choice
            |>> fire grind_coffee
            |>> fire heat_water
            |>> fire brew_coffee
            |>> fire remove_cup
```

In the end, we should have a petri net where the "insert coin" transition is enabled again.

```OCaml
is_enabled_opt pf insert_coin
```

Tests and execution

To develop and test your project, you need to accept a GitHub classroom assignment and clone the corresponding git repository. The link will be sent by email. The repository contains the template for your code and a set of tests that validate the programs with various target scores. These tests rely on the `OUnit2` library, which provides a framework for unit testing in OCaml. Before running the tests, ensure that the package is installed by executing the following command:

```Shell
$ opam install ounit2
```

Once the package is installed, you can run all the tests using `dune test`. If all tests pass, you should see output similar to this, showcasing the number of tests run for each task and the total execution time:

```Shell
.........
Ran: 9 tests in: 0.11 seconds.
OK
........
Ran: 8 tests in: 0.11 seconds.
OK
.......
Ran: 7 tests in: 0.11 seconds.
OK
```

To test the projects separately, you need to execute the following commands:

```Shell
$ dune build -f @test_project_2
```

and

```Shell
$ dune build -f @test_project_3
```

Moreover, to execute your implementation without running the test suite, you can do so by executing the following command:

```Shell
$ dune exec petrinets
```

Assignment Submission

This assignment is to be completed and submitted via **GitHub Classroom**. You can access the invite link in CLIP Messages. The repository is a template that includes all the necessary starter code and instructions. The **deadline** for submission is **May 28 at 23:59** (Lisbon time). You must push your final version, including code and any required documentation, to your GitHub Classroom repository before the deadline. **We will only consider the last commit before the deadline and late submissions will not be accepted**. Please ensure that your repository is correctly structured and contains all the necessary files. It is strongly recommended to test your code thoroughly before submitting. If you experience any issues with GitHub Classroom, contact us as soon as possible.

Instructions on Git and GitHub Classroom

1. Click the GitHub Classroom invitation link sent to your institutional email. This will redirect you to GitHub and prompt you to join a team.

2. This is a team project with teams of 2 students. Create a team using your student numbers as the team name with the lower number first and separated by an _.

3. Once the assignment is accepted, you will receive an email with a link to a repository with the project setup. Clone the repository to your local machine using your preferred Git client. You can use the command line interface (CLI) or the Git integration in Visual Studio Code. If you choose the CLI, you must first configure a personal access token for authentication. Instructions are available here. To clone the repository using the CLI, run:

   ```shell
   $ git clone https://github.com/Lap-2025/project-2-petrinets-[...]
   ```

   To clone the repository using Visual Studio Code, follow the user interface instructions when opening a new window and selecting the cloning repository option.

4. As you work on the project, commit your changes locally whenever you reach a stable state, ideally at the end of each session or after completing a specific task. To stage your changes (*i.e.*, inform git to keep track of the files you changed) and commit them, run:

   ```shell
   $ git add .
   $ git commit -m "A message describing the updates."
   ```

5. After committing, push your changes to the remote GitHub repository. This keeps your work backed up and visible to instructors.

   ```shell
   $ git push
   ```

Notes on Delivery:

DO NOT paste code directly on github hoping that it will compile. Test first locally and then push your changes.

Remarks

Recent advancements in Large Language Models (LLMs) have significantly changed the way people write code, offering faster development and easier access to solutions. However, LLMs can produce incorrect results or hallucinate information, making it crucial to verify their outputs. As disclosed at the start of the course, you are allowed to use these tools in the project, but **you must take full accountability** for your submission. If any part of the solution is generated with the assistance of an LLM (*e.g.*, ChatGPT, Copilot, etc.), the submission must explicitly state where and how LLMs were used (*e.g.*, as a comment).

You are encouraged to discuss general aspects of the project with each other (including in the Discord channel). But, when it comes to finding detailed solutions and write concrete code, it has

to be an internal effort by each individual. Solving problems and writing code requires intellectual effort, but only with effort can you evolve.

Beware of academic fraud. Each group is responsible for its project, has to produce original code, and cannot show or offer, directly or indirectly, on purpose or unintentionally, its code to another group. Note that it is much better to have zero in one of the three projects, than to be immediately excluded from the LAP course.

Have fun!