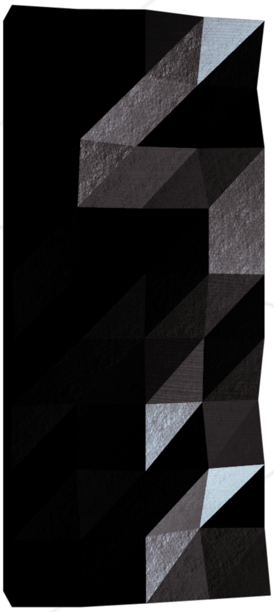

# Ocaml

## Object-Oriented Programming - 2

*Summary: The main theme of this module is to introduce the object oriented programming style with OCaml.*

*Version: 1.00*

# Contents

# Chapter I

# Foreword

It was a pleasure to burn.

It was a special pleasure to see things eaten, to see things blackened and changed. With the brass nozzle in his fists, with this great python spitting its venomous kerosene upon the world, the blood pounded in his head, and his hands were the hands of some amazing conductor playing all the symphonies of blazing and burning to bring down the tatters and charcoal ruins of history. With his symbolic helmet numbered 451 on his stolid head, and his eyes all orange flame with the thought of what came next, he flicked the igniter and the house jumped up in a gorging fire that burned the evening sky red and yellow and black. He strode in a swarm of fireflies. He wanted above all, like the old joke, to shove a marshmallow on a stick in the furnace, while the flapping pigeon-winged books died on the porch and lawn of the house. While the books went up in sparkling whirls and blew away on a wind turned dark with burning.

Montag grinned the fierce grin of all men singed and driven back by flame.

He knew that when he returned to the firehouse, he might wink at himself, a minstrel man, burnt-corked, in the mirror. Later, going to sleep, he would feel the fiery smile still gripped by his face muscles, in the dark. It never went away, that smile, it never ever went away, as long as he remembered.

By `Ray Bradbury`, in `Fahrenheit 451`.

# Chapter II

# General rules

- Your project must be realized in a virtual machine.
- Your virtual machine must have all the necessary software to complete your project. These softwares must be configured and installed.
- You can choose the operating system to use for your virtual machine.
- You must be able to use your virtual machine from a cluster computer.
- You must use a shared folder between your virtual machine and your host machine.
- During your evaluations you will use this folder to share with your repository.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a `0` during the evaluation.
- We encourage you to create test programs for your project even though this work **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded. If Deepthought is assigned to grade your work, it will be done after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

# Chapter III

# Ocaml piscine, general rules

- Every output goes to the standard output, and will be ended by a newline, unless specified otherwise.
- The imposed filenames must be followed to the letter, as well as class names, function names and method names, etc.
- Unless otherwise explicitly stated, the keywords `open`, `for` and `while` are forbidden. Their use will be flagged as cheating, no questions asked.
- Turn-in directories are `ex00/`, `ex01/`, . . . , `exn/`.
- You must read the examples thoroughly. They can contain requirements that are not obvious in the exercise's description.
- Since you are allowed to use the `OCaml` syntaxes you learned about since the beginning of the piscine, you are not allowed to use any additional syntaxes, modules and libraries unless explicitly stated otherwise.
- The exercices must be done in order. The graduation will stop at the first failed exercice. Yes, the old school way.
- Read each exercise FULLY before starting it! Really, do it.
- The compiler to use is `ocamlopt`. When you are required to turn in a function, you must also include anything necessary to compile a full executable. That executable should display some tests that prove that you've done the exercise correctly.
- Remember that the special token `";;"` is only used to end an expression in the interpreter. Thus, it must never appear in any file you turn in. Regardless, the interpreter is a powerfull ally, learn to use it at its best as soon as possible!
- The subject can be modified up to 4 hours before the final turn-in time.
- In case you're wondering, no coding style is enforced during the `OCaml` piscine. You can use any style you like, no restrictions. But remember that a code your peer-evaluator can't read is a code he or she can't grade. As usual, big functions are a weak style.
- You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are given a certain amount of freedom in how you choose to do the

exercises. However, some piscine day might explicitly cancel this rule, and you will have to respect directions and outputs perfectly.

- Only the requested files must be turned in and thus present on the repository during the peer-evaluation.
- Even if the subject of an exercise is short, it's worth spending some time on it to be absolutely sure you understand what's expected of you, and that you did it in the best possible way.
- By Odin, by Thor! Use your brain!!!

# Chapter IV

# Day-specific rules

- For every exercise you are required to turn-in a full program, with examples to prove your classes are working correctly. You may use any function or operator you want or see fit to use in your examples — as long as you don't need to link an external library to use it.
- Each exercise in today's subject is meant to be a sequel to the previous one; as a consequence, you will likely need to use your previous exercises to solve the one you're working on.
- As an obvious consequence, any exercise with 0 points for any reason (not turned in, does not compile, crashes, etc.) means the defence will not continue any further.
- You are generally free to use any type of data structure you want. But every time a method returns an associative list, it must be sorted in ascending order by its index.
- All chemical formulae for molecules must be written using the Hill notation.
- All the classes you implement today must be functional. Any imperative class in your code means no points for the entire exercise.
- Your code will not be modified during defences, which means a non-tested feature is a non-functional feature.
- For each exercise you are required to write a Makefile, which will compile your entire exercise. You can use OCamlMakefile to write your Makefile.

# Chapter V

# Exercise 00: Atoms

| | Exercise : 00 |
|---|---|
| Our whole universe was in a hot, dense state... | |
| Turn-in directory : *ex*00/ | |
| Files to turn in : `*.ml, Makefile` | |
| Forbidden functions : `None` | |

You will write a virtual class named `atom` for...an atom. Today we're doing some chemistry! An atom is defined by a few things:

- A `name`, of type `string`.
- A `symbol`, of type `string`.
- An `atomic_number`, of type `int`.

All these things have to be methods, and you have to be able to set what they return through the atom's constructor. Of course, feel free to look up what an atom is if you don't understand what these are.

Your class will contain at least a `to_string` method to describe shortly what your atom is, and an `equals` method to compare atoms; aside from that, you can implement anything you think is suitable or you may need in the next exercises.

To go along with your atom virtual class, you will also write some real atoms, which will of course inherit your `atom` class. You have to write at least the following classes:

- `hydrogen`
- `carbon`
- `oxygen`
- And three more atoms of your choice, as long as they really exist.

# Chapter VI

# Exercise 01: Molecules

| | Exercise : 01 |
|---|---|
| Then nearly fourteen billion years ago expansion started. Wait... | |
| Turn-in directory : *ex*01/ | |
| Files to turn in : `*.ml, Makefile` | |
| Forbidden functions : `None` | |

Now that you have your atoms, you can make some molecules! If you don't know what a molecule is, please look it up before beginning this exercise.

Your molecule class is virtual, and it needs at least the following things:

- A `name`, of type `string`.
- A `formula`, of type `string`.

All these things have to be methods. Of course, feel free to look up what a molecule is if you don't understand what these are.

One very important constraint: you have to store the molecule's atoms internally and this atom storage must be used to compute the molecule's chemical formula. Also, the chemical formula must be formatted using the `Hill notation`. As such, your constructor will only accept a name and a list of atoms.

The chemical formula is simple: it's a small string which describes what atoms and how many of them are contained in the molecule. Let's build an example with `Trinitrotoluene` (or `TNT` for short). We get the list of atoms, and we end up with:

- 3 atoms of `Nitrogen`
- 5 atoms of `Hydrogen`
- 6 atoms of `Oxygen`
- 7 atoms of `Carbon`

We know the symbols of these atoms are `N`, `H`, `O` and `C`. All we have to do is enumerate their symbols and their quantity, right? So you would get something like: $N_3H_5O_6C_7$.

But **NO! WAIT!** That's not how it works. The Hill notation says we get carbon, then hydrogen, then everything else in alphabetical order. So the result is: $C_7H_5N_3O_6$. Of course, you can't really write subscripts in a terminal, so writing it behind the symbol as it is is fine, like so: `C7H5N3O6`.

Your class will contain at least a `to_string` method to describe shortly what your molecule is, and an `equals` method to compare molecules; aside from that, you can implement anything you think is suitable or you may need in the next exercises.

To go along with your molecule class, you will write some real molecules, which will of course inherit your `molecule` class. You have to write at least the following classes:

- Water ($H_2O$))
- Carbon dioxyde ($CO_2$)
- And three more molecules of your choice, as long as they really exist.

Some elements like salts, ions and other things are not considered as molecules in their strictest meaning. However, for today we'll keep it simple and assume that a molecule is simply an aggregation of atoms, as long as its overall electrical charge is neutral.

# Chapter VII

# Exercise 02: Alkanes

| | Exercise : 02 |
|---|---|
| The Earth began to cool, the autotrophs began to drool... | |
| Turn-in directory : *ex*02/ | |
| Files to turn in : `*.ml, Makefile` | |
| Forbidden functions : `None` | |

Molecules are cool, but today we're focusing on a particular type of molecules, which are alkanes. Alkanes are a family of simple molecules composed of just carbon and hydrogen, which means we can create alkanes easily! The formula of an acyclic alkane is $C_nH_{2n+2}$. The name of the alkane simply depends on the value you give to $n$.

As such, your alkane's constructor only needs one parameter, which is $n$. It must be able to guess the name and the formula from this only $n$ parameter.

> You don't have to handle every possible alkane (because actually, you can't). Your data should be able to handle $1 \leq nleq12$; you won't be tested (and have the right to refuse to be tested) with $n > 12$.

Note that an alkane is still a molecule, which means you still have to provide the following methods:

- `name`
- `formula`
- `to_string`
- `equals`

To go along with your alkane class, you will write...some real alkanes! As usual. Of course, they will inherit your `alkane` class and you will write at least the following ones:

- `methane`
- `ethane`

- `octane`

# Chapter VIII

# Exercise 03: Reactions

| | Exercise : 03 |
|---|---|
| Neanderthals developed tools, we built a wall (we built the pyramids!) | |
| Turn-in directory : *ex*03/ | |
| Files to turn in : `*.ml, Makefile` | |
| Forbidden functions : `None` | |

Now we're doing interesting things! Chemical reactions. A chemical reaction is a simple thing, really. You have some molecules at the start of your reaction, and you have...some molecules at the end. There is only one really important rule to follow, which is: you must have the same exact amount of atoms at the start and at the end of your reaction. "Nothing is lost, nothing is created, everything is transformed". If you've never heard that sentence, look up Antoine de Lavoisier and what he has done for chemistry.

Basically, you can instantiate a new chemical reaction with a list of molecules. Then you can call a `get_result` method to get the result.

All that explanation was good, but now let's talk about code! You will write a virtual class named `reaction`, which will be instantiated with two collections of molecules, one for the start and one for the end of your reaction.

Your class will also provide the following virtual methods:

- `get_start: (molecule * int) list`
- `get_result: (molecule * int) list`
- `balance: reaction`
- `is_balanced: bool`

Your class doesn't have to do anything: all you have to provide is the structure. But if you think hard enough, you might think that some behaviour is common to all possible chemical reactions. Feel free to implement those you feel useful to implement. *wink wink*

# Chapter IX

## Exercise 04: Alkane combustion

|  | Exercise : 04 |
|---|---|
| Math, science, history unravelling the mystery... |  |
| Turn-in directory : *ex*04/ |  |
| Files to turn in : `*.ml, Makefile` |  |
| Forbidden functions : `None` |  |

Now that we know what a chemical reaction is, let's do a real one! Do you remember the alkanes I had you write a few exercises ago? Because now we're going to burn them. Our class will be named `alkane_combustion`, and it will be a subtype of `reaction` (what did you expect?).

But first, let's clear up some theory. An alkane combustion means you start with an alkane and some molecular oxygen, and you end up with carbon dioxyde and water. Always. That means our reaction will be (for example, with methane):

$$\mathrm{CH_4 + O_2 \rightarrow CO_2 + H_2O}$$

But wait! It's not actually that simple. I did say you had to have the same exact amount of atoms at the start and at the end, right? And if you look at what I just wrote, that doesn't really add up: you start with 1 carbon, 4 hydrogen and 2 oxygen, and you end with one carbon, 2 hydrogen and 3 oxygen; so you end up with not enough hydrogen and too much oxygen.

That's where `stoichiometrical coefficients` come in. Of course don't forget to look up the word if it sounded like Hebrew to you. But basically they are coefficients you add to the molecules in your reaction, so that you have the same amount of atoms on both sides. As a result:

$$\mathrm{CH_4 + 2\,O_2 \rightarrow CO_2 + 2\,H_2O}$$

Now on both sides you have 1 carbon, 4 hydrogen and 4 oxygen. Good, everything works!

Now let's talk about code. You will construct your class with a list of `alkane` objects, and it will implement the `reaction` class's methods:

- `get_start`: returns the list of molecules at the beginning of the reaction. Throws an exception if the reaction is not balanced.
- `get_result`: returns the list of molecules at the end of the reaction. Throws an exception if the reaction is not balanced.
- `balance`: returns a new alkane combustion, with the right (and smallest possible) stoichiometrical coefficients so that your combustion is balanced.
- `is_balanced`: `true` if your reaction is balanced, false otherwise.

Your balance method will have to process the list of alkanes, possibly removing duplicates, and then compute all the adequate coefficients for your reaction to work.

# Chapter X

# Exercise 05: Incomplete combustion

| | Exercise : 05 |
|---|---|
| It all started with the Big Bang! (BANG!) | |
| Turn-in directory : *ex*05/ | |
| Files to turn in : `*.ml, Makefile` | |
| Forbidden functions : `None` | |

Balancing alkane combustions was the main goal of your day, and if you succeeded in doing that, congratulations! I'm very proud of you. This final exercise is for those of you who want to go a bit further (but you still have to do it if you want full points on the day. `:)`)

You will take your `alkane_combustion` class and add a new method to it, called `get_incomplete_results`. And by incomplete results, I mean incomplete combustion. There are cases when your alkane isn't provided enough oxygen to completely burn, which leads to carbon monoxyde or soot being created. For example, with ethane and propane:

$$\mathrm{C_2H_6 + 3\,O_2 \rightarrow CO_2 + CO + 3\,H_2O}$$
$$\mathrm{C_2H_6 + 2\,O_2 \rightarrow CO + C + 3\,H_2O}$$
$$\mathrm{C_3H_8 + 4\,O_2 \rightarrow 2\,CO_2 + C + 4\,H_2O}$$
$$\mathrm{C_3H_8 + 3\,O_2 \rightarrow CO_2 + 2\,C + 4\,H_2O}$$
$$\mathrm{C_3H_8 + 3\,O_2 \rightarrow CO_2 + 2\,C + 4\,H_2O}$$
$$\mathrm{C_3H_8 + 3\,O_2 \rightarrow 2\,CO + C + 4\,H_2O}$$

And so on...

Of course, these aren't the only possible solutions. There might be solutions with no carbon dioxyde at all, but obviously if there's no carbon monoxyde then it's a complete combustion and it doesn't count. Your new method will be able to change the amount of oxygen in the reaction to compute the possible outcomes, which will be returned with type `(int * (molecule * int) list) list`, using the amount of oxygen as the list's index.

Obviously, chemical reactions are much more complex than that, and usually involve the molecules' internal structure. Today we'll keep it simple and consider that a reaction is valid if and only if their stoichiometrical coefficients add up correctly. That's it. If you can work out all the possible outcome using CO2, CO, C and H2O, I'm happy with that.

This exercise is not mandatory.

# Chapter XI

# Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your folders and files to ensure they are correct.

> The evaluation process will happen on the computer of the evaluated group.