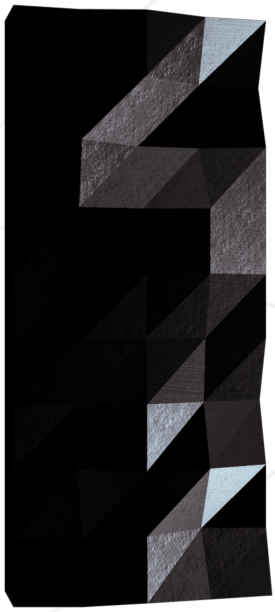

OCaml Module 00

Basic syntax and semantics

Summary: In this first OCaml module, you will discover the basic syntax and semantics of the language: values, types, operators, let bindings, functions and recursion. You can find the e-learning videos to get you started here.

Version: 1

Contents

Chapter I

Foreword

Black metal is an extreme subgenre and subculture of heavy metal music. Common traits include fast tempos, a shrieking vocal style, highly or heavily distorted guitars played with tremolo picking, raw (lo-fi) recording, unconventional song structures and an emphasis on atmosphere. Artists often appear in corpse paint and adopt pseudonyms.

During the 1980s, several thrash and death metal bands formed a prototype for black metal. This so-called first wave included bands such as Venom, Bathory, Mercyful Fate, Hellhammer and Celtic Frost. A second wave arose in the early 1990s, spearheaded by Norwegian bands such as Mayhem, Darkthrone, Burzum, Immortal and Emperor. The early Norwegian black metal scene developed the style of their forebears into a distinct genre. Norwegian-inspired black metal scenes emerged throughout Europe and North America, although some other scenes developed their own styles independently.

Figure I.1: Abbath from the Norwegian black metal band Immortal

Chapter II

OCaml modules, 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.
- You are only allowed to use the `OCaml` syntaxes you learned about since the OCaml module 00 up this current module or project. You are not allowed to use any additional syntax, modules and libraries unless explicitly stated otherwise.
- The assignments must be done in order. The graduation will stop at the first failed assignment.
- 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!
- No coding style is enforced during the `OCaml` piscine. You can use any style you like, no restrictions. Keep in mind 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 complete the assignments. However, some OCaml modules 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 III

Day-specific rules

For this day, you must respect directions and outputs perfectly. A single character mismatch means that the exercice is wrong, althought you are still free to wrap theses outputs as you wish. For instance, the first exercice of the days expects the words "positive" or "negative" followed by a new line. You made three tests to prove your work:

This output is right:

```
$> ocamlopt ft_test_sign.ml
$> ./a.out
positive
positive
negative
$>
```

This output is also right:

```
$> ocamlopt ft_test_sign.ml
$> ./a.out
Test with [42]: positive
Test with [0]: positive
Test with [-42]: negative
$>
```

This output is WRONG:

```
$> ocamlopt ft_test_sign.ml
$> ./a.out
positive positive negative
$>
```

This output is also WRONG:

```
$> ocamlopt ft_test_sign.ml
$> ./a.out
Test with [42]: [positive]
Test with [0]: [positive]
Test with [-42]: [negative]
$>
```

Chapter IV

Exercise 00: ft_test_sign

	Exercise 00
Exercise 00: ft_test_sign	
Turn-in directory : *ex*00/	
Files to turn in : `ft_test_sign.ml`	
Allowed functions : `print_endline`	

Write a function `ft_test_sign` of type `int -> unit` that displays `"positive"` or `"negative"` followed by a new line, depending on the sign of the parameter. `0` is always considered positive.

Exemples in the interpreter:

```
# ft_test_sign 42;;
positive
- : unit = ()
# ft_test_sign 0;;
positive
- : unit = ()
# ft_test_sign (-42);;
negative
- : unit = ()
#
```

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation.

Chapter V

Exercise 01: ft_countdown

	Exercise 01
Exercise 01: ft_countdown	
Turn-in directory : *ex*01/	
Files to turn in : `ft_countdown.ml`	
Allowed functions : `print_int and print_char`	

Write a function `ft_countdown` of type `int -> unit` that displays a countdown from the parameter's value down to `0` and a new line after each value. If the value is negative, just display `0` and a new line.

Exemples in the interpreter:

```
# ft_countdown 3;;
3
2
1
0
- : unit = ()
# ft_countdown 0;;
0
- : unit = ()
# ft_countdown (-1);;
0
- : unit = ()
#
```

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation.

Chapter VI

Exercise 02: ft_power

	Exercise 02
Exercise 02: ft_power	
Turn-in directory : *ex*02/	
Files to turn in : `ft_power.ml`	
Allowed functions : `Nothing`	

Write a function `ft_power` of type `int -> int -> int` that returns first parameter power the second parameter. Both parameters will always be positives or equal to `0`, but both will never be equal to `0` at the same time.

Exemples in the interpreter:

```
# ft_power 2 4;;
- : int = 16
# ft_power 3 0;;
- : int = 1
# ft_power 0 5;;
- : int = 0
#
```

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation.

Chapter VII

Exercise 03: ft_print_alphabet

	Exercise 03
Exercise 03: ft_print_alphabet	
Turn-in directory : *ex*03/	
Files to turn in : `ft_print_alphabet.ml`	
Allowed functions : `char_of_int, int_of_char and print_char`	

Write a function `ft_print_alphabet` of type `unit -> unit` that displays the alphabet on a single line followed by a new line.

Exemple in the interpreter:

```
# ft_print_alphabet ();;
abcdefghijklmnopqrstuvwxyz
- : unit = ()
#
```

Be sure to provide a tests suite to prove that your funciton works as intended during peer-evaluation ("tests suite" might be a little bit of an over-statement here). Obviously, printing 26 chars one after the other will be considered cheating. You are allowed only one use of `print_char` in the exercice excluding the `print_char` for the newline at the end of input.

Chapter VIII

Exercise 04: ft_print_comb

	Exercise 04
Exercise 04: ft_print_comb	
Turn-in directory : *ex*04/	
Files to turn in : `ft_print_comb.ml`	
Allowed functions : `print_int and print_string`	

Write a function `ft_print_comb` of type `unit -> unit` that displays in ascending order all the different combinaisons of 3 digits, each digit different from the 2 others, and the 3 digits also in ascending order. Each combinaison is separated from the next one by a comma and a space. Finish your display by a new line.

You must have something that starts an finishes that way :

```
# ft_print_comb ();;
012, 013, 014, 015, 016, 017, 018, 019, 023, <more numbers>, 789
- : unit = ()
#
```

As additonnal informations, `987` is not part of the sequence because `789` is already part of it. Also note that for instance, `999` is not part of the sequence because the 3 digits are not different from the 2 others.

Displaying the right answer in a big string without actually computing it will be treated as cheating during the peer-evaluation.

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation (“tests suite” might be a little bit of an over-statement here as well).

Chapter IX

Exercise 05: ft_print_rev

	Exercise 05
Exercise 05: ft_print_rev	
Turn-in directory : *ex*05/	
Files to turn in : `ft_print_rev.ml`	
Allowed functions : `print_char, String.get and String.length`	

Write a function `ft_print_rev` of type `string -> unit` that prints its `string` parameter in reverse order, one character at a time, ending with a new line.

Exemple in the interpreter :

```
# ft_print_rev "Hello world !";;
! dlrow olleH
- : unit = ()
# ft_print_rev "";;

- : unit = ()
#
```

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation.

Chapter X

Exercise 06: ft_string_all

	Exercise 06
Exercise 06: ft_string_all	
Turn-in directory : *ex*06/	
Files to turn in : `ft_string_all.ml`	
Allowed functions : `String.get and String.length`	

Write a function `ft_string_all` of type `(char -> bool) -> string -> bool`. To help you get on tracks, the first parameter, of type `char -> bool`, is a function. As this function returns a `bool`, it can therefore be refered to as a "predicate" function.

So, the function `ft_string_all` takes a predicate function and a string as parameters, and applies each character of the string to the predicate function. If the predicate is `true` for every character of the string, `ft_string_all` returns `true`. Otherwise, if the predicate function is `false` for at least one character, `ft_string_all` returns `false`.

Exemples in the interpreter :

```
# let is_digit c = c >= '0' && c <= '9';;
val is_digit : char -> bool = <fun>
# ft_string_all is_digit "0123456789";;
- : bool = true
# ft_string_all is_digit "012EAS67B9";;
- : bool = false
#
```

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation.

Chapter XI

Exercise 07: ft_is_palindrome

	Exercise 07
Exercise 07: ft_is_palindrome	
Turn-in directory : *ex*07/	
Files to turn in : `ft_is_palindrome.ml`	
Allowed functions : `String.get and String.length`	

Write a function `ft_is_palindrome` of type `string -> bool` that returns `true` if the string parameter is a palindrome character by character, `false` otherwise. If you intend to use your previous function `ft_string_all`, please embed its code in the file `ft_is_palindrome.ml` as well. The empty string is a palindrome.

Exemples in the interpreter:

```
# ft_is_palindrome "radar";;
- : bool = true
# ft_is_palindrome "madam";;
- : bool = true
# ft_is_palindrome "car";;
- : bool = false
# ft_is_palindrome "";;
- : bool = true
```

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation.

Chapter XII

Exercise 08: ft_rot_n

	Exercise 08
Exercise 08: ft_rot_n	
Turn-in directory : *ex*08/	
Files to turn in : `ft_rot_n.ml`	
Allowed functions : `char_of_int, int_of_char and String.map`	

Write a function `ft_rot_n` of type `int -> string -> string`. Let `n` the first parameter and `str` the second parameter, `ft_rot_n` rotates each lower case and upper case alphabetical characters of `str` of `n` in acsending order. The value `n` will always be positive.

Exemples in the interpreter:

```
# ft_rot_n 1 "abcdefghijklmnopqrstuvwxyz";;
- : string = "bcdefghijklmnopqrstuvwxyza"
# ft_rot_n 13 "abcdefghijklmnopqrstuvwxyz";;
- : string = "nopqrstuvwxyzabcdefghijklm"
# ft_rot_n 42 "0123456789";;
- : string = "0123456789"
# ft_rot_n 2 "OI2EAS67B9";;
- : string = "QK2GCU67D9"
# ft_rot_n 0 "Damned !";;
- : string = "Damned !"
# ft_rot_n 42 "";;
- : string = ""
# ft_rot_n 1 "NBzlk qnbjr !";;
- : string = "OCaml rocks !"
```

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation.

Chapter XIII

Exercise 09: ft_print_comb2

	Exercise 09
Exercise 09: ft_print_comb2	
Turn-in directory : *ex*09/	
Files to turn in : `ft_print_comb2.ml`	
Allowed functions : `print_char, print_int`	

Write a function `ft_print_comb2` of type `unit -> unit` that displays each unique combinaisons of two numbers, each one between `00` and `99`, in ascending order. Each combinaison is separated from the next one by a comma and a space. Finish your display by a new line.

You must have something that starts an finishes that way :

```
# ft_print_comb2 ();;
00 01, 00 02, 00 03, 00 04, 00 05, <more numbers>, 00 99, 01 02, <more numbers>, 97 99, 98 99
- : unit = ()
#
```

Displaying the right answer in a big string without actually computing it will be treated as cheating during the peer-evaluation.

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation (“tests suite” might be a little bit of an over-statement as well).