OCaml Programming Tutorial

Funktionale Programmierung (Technische Universität München)

OCaml Tutorial

Solving tasks with functional programming often requires a different way of thinking about problems than imperative programming, as recursion is used in place of classical loops, and data structure are immutable.

This little tutorial is designed to help you get started with functional programming programming in OCaml. You'll understand how to debug your programs with `utop` and be a little more familiar with recursion and pattern matching.

If you want to follow along, make sure the current working directory of your shell is the folder containing this README

`utop` and the meaning of `.mli`

Your assignment repositories are always setup in such a way that you can run `dune utop src` (or `dune utop .` inside `src`).

In this repository, both `assignment.ml` and `assignment.mli` are empty.

If we want to debug anything, we need to first write some code. Let's start by writing a function that computes the sum of a list of integers. You know this one!

```
let rec sum = function
  | [] -> 0
  | x::xs -> x + sum xs
```

Let's add this to `assignment.ml` and run `dune utop src`. You will get the following output:

`Error (warning 32 [unused-value-declaration]): unused value sum.`

So far, so bad! `utop` sees that we do not use the value `sum`. Remember, functions are just values. We can try to get around this and change the name to `_sum`, which indicates that we know this value isn't used. Running `dune utop src` again is successful! Now, let's try:

```
utop # Assignment._sum [1,2,3];;
Error: Unbound value Assignment._sum
```

...no real progress made here 😕 Hey, what's this `assignment.mli` file for anyways? It's a `header` file!

If we want utop to recognize a value, we need to include it inside the header with its type signature.

How do we do that? Simple! First, let's rename our function back to `sum`. Now, we add the following line to `assignment.mli`.

```
val sum : int list -> int
```

Try to run `dune utop src` again and...

```
utop # Assignment.sum [1;2;3];;
- : int = 6
```

Nice! If you're unsure what type your values have, don't worry, the OCaml platform displays the inferred type above automatically. When in doubt, try using that 😉

Two things you should also keep in mind: Unfortunately, `utop` does not include functionality to reload your program after changes have been made. You'll have to exit and re-run `dune utop src`. Also, if you want to use a value that's defined somewhere else in your program **you need to define that value before you use it!**

Recursion and pattern matching

You know how to recurse over lists and how to pattern-match on them. You also know that you can pattern-match on constructors of your custom types. This section aims to help you to have an easier time solving recursive problems and how to express them a little more idiomatically.

To make it easy to follow, let's do something silly: Implement multiplication recursively. Whacky!

When solving a recursive problem, you always think about two things: The base case and the recursive-case.

Base-Case

This is where your recursive function terminates. No recursive calls are made here. Without writing any code yet, think about this for `mul x y`. We know: `x * 0 = 0` and `0 * y = 0`

Any number times zero is zero, amazing. How to put this into code? Pattern matching of course!

```
let rec mul x y = match x, y with
  | 0, _ -> 0
  | _, 0 -> 0
  | _    -> (* TODO recursive case! *)
```

What's happening here? We first construct a tuple out of `x` and `y` using `,` and then match against the tuple! That way we can match two expressions at the same time. We could have bracketed it, but those are optional and often left out. When in doubt, just add them 😃 And, yes, you can pattern match on `int`!

Recursive case

What happens then if both x and y are not zero? Well, multiplication is just repeated addition. So just write x + mul x (y-1)? Not so fast! What happens if either x or y are negative? We can just flip the sign of the negative value and return a negative value. This is important, because our base case is at 0! If we kept counting down y we'd get infinite recursion. What if both are? Positive result, but flip both signs!

This is gonna be one ugly if ... then ... else - or is it? You probably guessed it, pattern matching saves us yet again!

We get our final result:

```
let rec mul x y = match x, y with
  | 0, _ -> 0
  | _, 0 -> 0
  | _    -> match x < 0, y < 0 with
    | true, true  -> mul (-x) (-y)
    | true, false -> - mul (-x) y
    | false, true -> - mul x (-y)
    | _           -> x + mul x (y-1)
```

Using the same tuple-matching trick from earlier, we get a really concise way of grouping these results! Here, we match against a tuple of type bool * bool.

Don't forget to edit your .mli file for testing 😉

In the solution directory you can also find an example using a top-level helper. We hope you've learned a little bit and wish you all the best in your functional programming journey!