Tutorial #1 - Type systems and derivations

▷ OCaml (`http://caml.inria.fr`) is a functional programming language developed at INRIA, distributed with a compiler `ocamlc` and an interaction loop `ocaml`.

In order to write OCaml code, the most direct way consists in launching `emacs` on a `.ml` file, and then running `M-x tuareg-mode`. Then, start an interaction loop using the `C-c C-s` shortcut, and using the default interpreter name (`ocaml`).

Execute every expression in the interaction loop using the `C-x C-e` shortcut.

Exercice 1: Playing with OCaml

The evaluation of an expression in OCaml consists in :

Each of the following expression is meant to be executed with the OCaml interpreter and related to the expressions of the λ-calculus studied during the class. Afterwards, this list can be considered as a cheat sheet for finding the syntax of expressions.

```
true;;                    (* Booleans *)
false;;
0;;                       (* Integers *)
65535;;
if true then 7 else 14;;  (* Conditional *)
fun x → x + 1;;           (* Abstraction *)
(fun x → x + 1) 8;;       (* Application *)
"abcde";;                 (* Strings *)
"meta"^"for";;            (* String concatenation *)
```

▷ To bind a name to a value at the toplevel : `let x = 1;;`.
To bind a name to a value as a local expression : `let y = "a" in y ^ y;;`.

The following code is provided as an example of the infamous factorial function :

```
let rec fact x =          (* rec indicates that the function is recursive *)
  if (x <= 0) then 1
  else let y = x-1 in     (* declaration of a local binding *)
       x * fact y
```

Exercice 2: Algebraic data types

In this exercise, we introduce a particular sort of types named *algebraic data types*, also called *sum types* in OCaml. Such datatypes are defined using the following construction :

```
type ⟨type_name⟩ =
  | ⟨Constr_1⟩ of ⟨Type_1⟩
  | ⟨Constr_2⟩ of ⟨Type_2⟩
    ...
  | ⟨Constr_n⟩ of ⟨Type_n⟩
```

Example :

```
type value =
  | Int of int
  | Float of float;;
```

In order to manipulate sum types, OCaml contains a construct named *pattern-matching*, working as follows :

```
match ⟨expr⟩ with
  | ⟨p_1⟩ → ⟨expr_1⟩
  | ⟨p_2⟩ → ⟨expr_2⟩
    ...
  | ⟨p_n⟩ → ⟨expr_n⟩
```

Example :

```
let string_of_value x =
 match x with
  | Int i   → string_of_int i;
  | Float f → string_of_float f;;
```

For further reading on the subject, one can find at the address `https://caml.inria.fr/pub/docs/oreilly-book/html/book-ora016.html` some documentation and examples about the definition of types and the pattern-matching in OCaml.

1. Write a sum type `weight` for the manipulation of weights in kilos, in pounds and in carats.

2. Write the translation function that can convert any element of type `weight` in kilos. (*Cultural hint :* 1kg = 2.205 lbs = 5000 carats)

Exercice 3: Simple untyped λ-calculus

Consider the following definition of a language based on the untyped λ-calculus discussed during the class :

Syntax			Evaluation rules
t ::=		*terms*	$\dfrac{t_1 \to_\beta t_1'}{(t_1\ t_2) \to_\beta (t_1'\ t_2)}$
	x	*variable*	
	λx.t	*abstraction*	
	(t t)	*application*	$\dfrac{t \to_\beta t'}{(v\ t) \to_\beta (v\ t')}$
v ::=		*values*	
	λx.t	*abstraction value*	$(\lambda x.t_1\ t_2) \to_\beta [x \mapsto t_2]t_1$

Let us translate this definition into OCaml.

▷ First, retrieve the code given in the sources, compile it with `make` and load the `main.ml` file in your editor. Start the interaction loop (`C-c C-s`) with the command : `ocaml syntax.cma`.

Execute (`C-x C-e`) the two first lines (`open Syntax;;` and `open Parser_helper;;`). After that, it becomes possible to copy-paste the following examples into the buffer and test them.

Consider the following definition in the file `syntax.ml`, that corresponds to a sum type describing the grammar of the language :

```
type id = string                    (* Identifiers *)

type term =
  | TmVar  of id                    (* Variable *)
  | TmAbs  of id * term             (* Abstraction *)
  | TmApp  of term * term           (* Application *)
```

This code defines a type and a set of constructors for building λ-expressions : `TmVar`, `TmAbs` and `TmApp`. The following are examples of λ-expressions in OCaml :

```
TmAbs ("x", TmVar "x");;                    (* fun x → x          *)
TmAbs ("y", TmAbs ("z", TmVar "z"));;       (* fun y → fun z → z  *)
TmApp (TmAbs ("x", TmVar "x"), TmVar "z");; (* ((fun x → x) z)    *)
```

The sources also provide a parser to simplify the writing of complex expressions, given with the `parse` function :

```
parse "fun x → x";;                 (* → TmAbs ("x", TmVar "x") *)
```

The syntax of the language is (as much as possible) the same as in OCaml.

To understand this code, it is possible to read the function `term_to_string` that transforms a λ-expression into a string acceptable by LaTeX :

```
let rec term_to_string t =
  match t with
  | TmVar v      → "$"^(blue_string v)^"$"
  | TmAbs (v,e)  → "$\\lambda "^(blue_string v)^"$ . "^(term_to_string e)
  | TmApp (a,b)  → (term_to_string a)^" "^(term_to_string b)
```

Finally, the code contains functions for the substitution (`substitute`), the α-renaming (`rename`) and the β-reduction (`reduce_one` and `reduce`) of expressions, but most of these functions are not written.

1. Write the code for `substitute` and `rename`. The definition for these functions appears on the slides of the course.
2. Write the code for `reduce_one`. This function should return a λ-expression where only one β-reduction step has been applied.
3. Test your code with different examples. In particular, write a term whose reduction is not finite.

```
substitute "x" (TmVar "y") (parse "(fun x → x) x");;          (* → ((fun x → x) y) *)
rename     (parse "(fun x → x) (fun y → y)");;                (* → ((fun x1 → x1) (fun x2 → x2)) *)
reduce_one (parse "(fun x → x x) (fun y → y)") (Ctx []);;     (* → ((fun y → y) (fun y → y)) *)
reduce     (parse "(fun x → x x) (fun y → y)") (Ctx []);;     (* → (fun y → y) *)
```

Exercice 4: Simple extensions

In this exercise, we propose to extend the language so as to contain rules for booleans and integers. For the record, these rules are the following :

Syntax			Evaluation rules
t ::=	...	*terms*	$\dfrac{t_1 \to_\beta t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \to_\beta \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}$
	true, false	*booleans*	
	zero, succ t	*naturals*	
	if t then t else t	*if-then-else*	$\texttt{if true then } t_2 \texttt{ else } t_3 \to_\beta t_2$
	iszero t	*zero-equality*	$\texttt{if false then } t_2 \texttt{ else } t_3 \to_\beta t_3$
v ::=	...	*values*	$\dfrac{t \to_\beta t'}{\texttt{iszero } t \to_\beta \texttt{iszero } t'}$
	true, false	*boolean value*	
	nv	*numeric value*	$\texttt{iszero zero} \to_\beta \texttt{true}$
nv ::=		*numeric values*	$\texttt{iszero } (\texttt{succ t}) \to_\beta \texttt{false}$
	zero	*zero value*	
	succ nv	*successor value*	

The first objective consists in extending the language with the boolean expressions (`true`, `false` and `if .. then .. else`). For the sake of safety, copy your entire code in a new directory.

1. Extend the OCaml grammar for the λ-expressions with boolean expressions (3 new constructors).
2. Modify the parser file `parser.mly` so as to use these constructors (this mostly consists in adapting the file comments to your code).
3. Extend `term_to_string` and the matcher functions `is_abstr` and `is_value`.
4. Extend in order : `substitute`, `rename` and `reduce_one` (the `reduce` function should continue to work as before).

Now copy your code into a new directory, and apply the same operations to handle the natural numbers.

Exercice 5: Syntax derivations

The `bussproofs` LaTeX package allows to build derivation trees in a simple manner :

```
\begin{prooftree}
\AxiomC{}
\UnaryInfC{0}\UnaryInfC{iszero 0}
\AxiomC{}
\UnaryInfC{true}
\AxiomC{}
\UnaryInfC{false}
\TrinaryInfC{if (iszero 0) then t
                           else f}
\end{prooftree}
```

$$\dfrac{\dfrac{\dfrac{}{0}}{\text{iszero } 0} \quad \dfrac{}{\text{true}} \quad \dfrac{}{\text{false}}}{\text{if (iszero 0) then t else f}}$$

1. Construct a function that builds a derivation tree for a term.

You may take inspiration from the `term_to_string` function in the `syntax.ml` file. The code is already designed to print the result string on the standard output when using the `make latex` command, and generate a `syntax.pdf` file.

```
let leftlabel s =
  "\\LeftLabel{\\footnotesize[\\textcolor{green!70!black}{\\textsc{"^s^"}}]}\n"

let derivation_tree t =
  let rec derivation_tree_int t = match t with
    | TmZero | TmTrue | TmFalse → "\\AxiomC{}\n"
                                  ^(leftlabel "Ax")
                                  ^"\\UnaryInfC{"
                                  ^(term_to_string t)^"}\n"
    | TmSsucc u    → (derivation_tree_int u)
                     ^(leftlabel "Succ")
                     ^"\\UnaryInfC{"^(term_to_string t)^"}\n"
    | TmIsZ u      → (derivation_tree_int u)
                     ^(leftlabel "IsZero")
                     ^"\\UnaryInfC{"^(term_to_string t)^"}\n"
    | TmIf(i,u,e) → (derivation_tree_int i)
                     ^(derivation_tree_int u)
                     ^(derivation_tree_int e)
                     ^(leftlabel "If")
                     ^"\\TrinaryInfC{"^(term_to_string t)^"}\n"
  in "\\begin{prooftree}\n\sffamily\n"
     ^(derivation_tree_int t)
     ^"\\end{prooftree}\n";;
```