- Introduction and overview
- Basic types, definitions and functions
- Basic data structures
- More advanced data structures
- Higher order functions
- Exceptions, input/output and imperative constructs
- **Modules and data abstraction**

Table of Contents

Guests

Structuring software with modules
Week 6 Echéance le déc 12, 2016 at 23:30 UTC

Information hiding
Week 6 Echéance le déc 12, 2016 at 23:30 UTC

Case study: A module for dictionaries
Week 6 Echéance le déc 12, 2016 at 23:30 UTC

Functors
Week 6 Echéance le déc 12, 2016 at 23:30 UTC

Modules as compilation units

- Project

TYPE ABSTRACTION USING A SIGNATURE (20/20 points)

Encapsulate the type and values given in the template in a module named `Exp`.

To make `e` abstract, assign a signature to the module `Exp` that makes the type `e` abstract and publish the functions `int`, `mul` and `add`.

Given that interface, the only way to build a value of type `e` is to use the functions `int`, `mul` `add` and `to_string`. Such functions are called *smart constructors* because they perform some smart operations when they build values.

These smart constructors enforce the invariant that an expression represented by a value of type `e` is always simplified, i.e. it does not contain a subexpression of the form `e * 1`, `1 * e`, `e * 0`, `0 * e`, `0 + e` or `e + 0`.

- The following expression should be accepted.

```
Exp.mul (Exp.int 0) (Exp.add (Exp.int 1) (Exp.int 2))
```

- The following expression should be rejected.

```
Exp.EMul (Exp.EInt 0) (Exp.EAdd (Exp.EInt 1) (Exp.EInt 2))
```

Unfortunately, turning `e` into an abstract data type prevents the user from pattern matching over values of type `e`. To allow pattern matching while forbidding the direct application of data constructors, OCaml provides a mechanism called `private types`. The interested student can get more information about this advanced (off-topic) feature here.

YOUR OCAML ENVIRONMENT

```
module Exp : sig
  type e
  val int : int -> e
  val mul : e -> e-> e
  val add : e -> e-> e
  val to_string : e -> string
end = struct
  type e = EInt of int | EMul of e * e | EAdd of e * e

  let int x = EInt x

  let mul a b =
    match a, b with
    | EInt 0, _ | _, EInt 0 -> EInt 0
    | EInt 1, e | e, EInt 1 -> e
    | a, b -> EMul (a, b)

  let add a b =
    match a, b with
    | EInt 0, e | e, EInt 0 -> e
    | a, b -> EAdd (a, b)

  let rec to_string = function
    | EInt i -> string_of_int i
    | EMul (l, r) -> "(" ^ to_string l ^ " * " ^ to_string r ^ ")"
    | EAdd (l, r) -> "(" ^ to_string l ^ " + " ^ to_string r ^ ")"
end;;

```

Evaluate >

Switch >>

Typechecl

Reset Templ

Full-screen |

Check & Sa

Exercise complete (click for details) 20 pts

Found Exp with compatible type.

Your module Exp is compatible. 5 pts

Type `Exp.e` is abstract as expected. 5 pts

Now I will check that you didn't change the behaviour.

v Testing `Exp.to_string` Completed, 10 pts

Computing and printing

```
(Exp.mul
   (Exp.mul
      (Exp.add
         (Exp.mul (Exp.int 3)
           (Exp.add (Exp.add (Exp.int 4) (Exp.add (Exp.int 0) (Exp.int 3)))
              (Exp.mul (Exp.mul (Exp.int 3) (Exp.int 4)) (Exp.int 0)))))
     (Exp.add (Exp.int 3)
        (Exp.add
           (Exp.add (Exp.add (Exp.int 3) (Exp.mul (Exp.int 3) (Exp.int 2)))
              (Exp.mul (Exp.add (Exp.int 4) (Exp.int 1))
                 (Exp.mul (Exp.int 1) (Exp.int 4)))) (Exp.int 2))))
```

Correct value 1 pt

`"((((3 * (4 + 1)) + 4) * (4 + 3)) * (3 + (((3 + (3 * 2)) + ((4 + 1) * 4)) + 2)))"`

Computing and printing

```
  (Exp.int 4)
```

Correct value `"4"` 1 pt

Computing and printing

```
  (Exp.int 1)
```

Correct value `"1"` 1 pt

Computing and printing

```
  (Exp.mul
     (Exp.mul
        (Exp.mul (Exp.int 2)
           (Exp.mul (Exp.mul (Exp.int 0) (Exp.int 3))
              (Exp.add (Exp.int 4) (Exp.mul (Exp.int 3) (Exp.int 3)))))
        (Exp.add
           (Exp.mul (Exp.mul (Exp.int 1) (Exp.add (Exp.int 3) (Exp.int 0)))
              (Exp.mul (Exp.int 2) (Exp.int 4)))
           (Exp.mul (Exp.add (Exp.int 0) (Exp.int 2)) (Exp.int 3))))
     (Exp.add (Exp.add (Exp.mul (Exp.int 3) (Exp.int 2)) (Exp.int 1))
        (Exp.add (Exp.int 0)
           (Exp.add (Exp.int 2)
              (Exp.mul (Exp.add (Exp.int 1) (Exp.int 0))
                 (Exp.mul (Exp.int 0) (Exp.int 2)))))))
```

Correct value `"0"` 1 pt

Computing and printing

```
  (Exp.mul
     (Exp.mul (Exp.add (Exp.int 1) (Exp.int 3))
        (Exp.add
           (Exp.add (Exp.mul (Exp.int 0) (Exp.int 1))
              (Exp.add (Exp.int 4) (Exp.int 2)))
           (Exp.mul (Exp.int 4) (Exp.int 3))))
     (Exp.add
        (Exp.mul (Exp.int 2)
           (Exp.add (Exp.int 3)
              (Exp.mul (Exp.int 2) (Exp.add (Exp.int 3) (Exp.int 0)))))
        (Exp.int 4)))
```

Correct value `"(((1 + 3) * ((4 + 2) + (4 * 3))) * ((2 * (3 + (2 * 3))) + 4))"` 1 pt

Computing and printing

```
  (Exp.mul
     (Exp.add
        (Exp.add
           (Exp.add (Exp.int 1)
              (Exp.add (Exp.int 0) (Exp.mul (Exp.int 1) (Exp.int 0))))
           (Exp.mul (Exp.int 3) (Exp.int 2)))
        (Exp.add
           (Exp.mul (Exp.add (Exp.add (Exp.int 2) (Exp.int 2)) (Exp.int 3))
              (Exp.mul (Exp.add (Exp.int 4) (Exp.int 1))
                 (Exp.add (Exp.int 3) (Exp.int 1))))
           (Exp.add (Exp.int 0)
              (Exp.mul (Exp.mul (Exp.int 4) (Exp.int 0)) (Exp.int 1)))))
     (Exp.int 4))
```

Correct value `"(((1 + (3 * 2)) + (((2 + 2) + 3) * ((4 + 1) * (3 + 1)))) * 4)"` 1 pt

Computing and printing

```
  (Exp.add
     (Exp.mul
        (Exp.mul
           (Exp.mul
              (Exp.mul (Exp.mul (Exp.int 3) (Exp.int 1))
                 (Exp.mul (Exp.int 0) (Exp.int 2)))
              (Exp.add (Exp.int 0) (Exp.int 4)))
           (Exp.mul (Exp.add (Exp.int 2) (Exp.add (Exp.int 0) (Exp.int 0)))
              (Exp.add (Exp.mul (Exp.int 2) (Exp.int 1)) (Exp.int 2))))
        (Exp.add
           (Exp.add
              (Exp.add (Exp.add (Exp.int 1) (Exp.int 0))
                 (Exp.mul (Exp.int 1) (Exp.int 1)))
              (Exp.mul (Exp.mul (Exp.int 0) (Exp.int 4))
                 (Exp.mul (Exp.int 4) (Exp.int 3))))
           (Exp.add (Exp.add (Exp.int 2) (Exp.mul (Exp.int 2) (Exp.int 3)))
              (Exp.int 2))))
     (Exp.add
        (Exp.add (Exp.int 3)
           (Exp.add (Exp.int 0) (Exp.mul (Exp.int 2) (Exp.int 4))))
        (Exp.add
           (Exp.mul (Exp.add (Exp.mul (Exp.int 4) (Exp.int 3)) (Exp.int 2))
              (Exp.mul (Exp.int 4) (Exp.int 3)))
           (Exp.add (Exp.mul (Exp.int 0) (Exp.add (Exp.int 1) (Exp.int 3)))
              (Exp.mul (Exp.int 1) (Exp.add (Exp.int 3) (Exp.int 4)))))))
```

```
   (Exp.add (Exp.add (Exp.int 2) (Exp.int 0))
      (Exp.add (Exp.int 0)
         (Exp.mul
            (Exp.mul (Exp.mul (Exp.int 0) (Exp.int 4))
               (Exp.add (Exp.int 0) (Exp.int 3))) (Exp.int 1))))
   (Exp.mul
      (Exp.add (Exp.int 2)
         (Exp.mul (Exp.mul (Exp.int 2) (Exp.int 3))
            (Exp.add (Exp.int 3) (Exp.int 2)))) (Exp.int 1)))
```

Correct value "(2 * (2 + ((2 * 3) * (3 + 2))))" 1 pt

Computing and printing

```
(Exp.int 1)
```

Correct value "1" 1 pt

Computing and printing

```
(Exp.mul
   (Exp.mul
      (Exp.mul (Exp.add (Exp.int 1) (Exp.int 3))
         (Exp.add (Exp.int 1)
            (Exp.mul (Exp.add (Exp.int 1) (Exp.int 3))
               (Exp.mul (Exp.int 2) (Exp.int 0))))) (Exp.int 1))
   (Exp.mul (Exp.add (Exp.int 2) (Exp.int 3)) (Exp.int 0)))
```

Correct value "0" 1 pt