Exercises using `expr`

- **Show the abstract syntax tree for expression `4+-(7*-8+4)`:**

- **Give the OCaml expression of type `expr` for that tree:**

Exercises using `expr` (cont.)

- **Write the function `countPluses: expr` → `int`, which counts the number of `Plus` operations in an `expr`:**

```
let rec countPluses e = match e with
  Int i ->

| Plus(e1, e2) ->

| Times(e1, e2) ->

| Negate e ->
```

Exercises using `expr` (cont.)

- **Write the function** `eval: expr` $\rightarrow$ `int`**, which evaluates its argument, e.g.** `eval (Times(Negate(Int 5), Int 6))` **= -30.**

Exercises using `expr` (cont.)

- **For a little more practice, consider this slightly different definition of type `expr`:**

```
type expr = Int of int | Binop of bop*expr*expr
                       | Unop of uop*expr
 and bop = Plus | Times
 and uop = Negate
```

Define `eval` for this definition of `expr`:

Abstract syntax of OCaml

- **Here's a (partial) abstract syntax for OCaml:**

```
type ocamlexpr = Int of int  |  Binop of bop * ocamlexpr * ocamlexpr
               |  Var of string  |  App of ocamlexpr * ocamlexpr
               |  Let of def * ocamlexpr  |  Letrec of def * ocamlexpr
 and def = string * string list * ocamlexpr
 and bop = Plus  |  Times
```

- **Write the `ocamlexpr` corresponding to**

```
let f a = let x = 0 in a+x in f 7
```

Ex: Abstract syntax of OCaml

`occursin`: string → ocamlexpr → bool checks if a variable or function name is used in an expression (in its scope), e.g.

a occurs in `let x = a in ...`
a occurs in `let x = 0 in a+x`
a does not occur in `let a = 0 in a`
f occurs in `let g x = f (x+1) in g 0`
f does not occur in `let rec f x = f (x+1) in f 0`

```
let rec occursin v e = match e with
   Int i ->
 | Binop(b, e1, e2) ->

 | Var(s) ->

 | App(e1, e2) ->

 | Let(d, e) ->

 | Letrec(d, e) ->
```

```
and exp = Operation of exp * binary_operation * exp
    | Array of exp * exp
    | Length of exp
    | MethodCall of exp * id * (exp list)
    | Integer of int
    | True
    | False
    | Id of id
    | This
    | NewArray of exp_type * exp
    | NewId of id
    | Not of exp
    | Null
    | String of string
    | Float of float

and binary_operation = And
    | Or
    | LessThan
    | Plus
```

Ex: pretty-print expressions

- **Write** `pp : exp` $\rightarrow$ **string, that produces a printed version of its argument, such that, if it were parsed, it would produce the same abstract syntax tree. (**`pp` **stands for "pretty-print".) Use parentheses freely (even though some will be unnecessary).**

```
let rec pp_bop binop = match binop with
   And -> "&&" | LessThan -> "<" | ...

end rec pp e = match e with
   Operation(e1, binop, e2) ->

 | Array(e1, e2) ->

 |
```