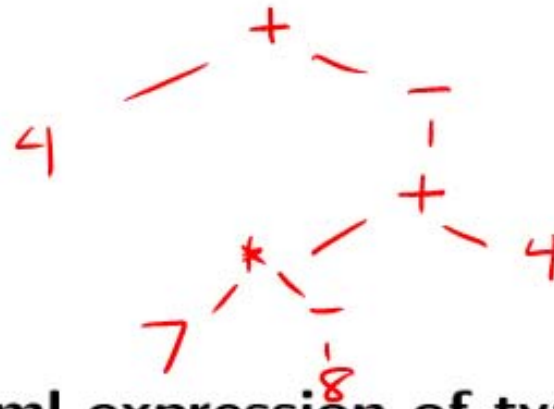


Exercises using `expr`

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



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

*plus (Int 4,
negate (plus (Times (Int 7,
negate (Int 8)),
Int 4)))*

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 -> 0
```

```
  | Plus(e1, e2) -> countPluses e1 + countPluses e2
```

```
  | Times(e1, e2) -> countPluses e1 + countPluses e2
```

```
  | Negate e -> countPluses e
```

Exercises using `expr` (cont.)

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

let rec eval e = match e with
 | Int i → i
 | Plus (e1, e2) → eval e1 + eval e2
 | Times (e1, e2) → eval e1 * eval e2
 | Negate e1 → - (eval e1)

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`:

let rec eval e = match e with
 Int i → i
 | Binop (b, e1, e2) → apply b (eval e1)
 (eval e2)
 | Unop (u, e1) → - (eval e1)
and apply b i1 i2 = match b with
 plus → i1 + i2 | Times → i1 * i2

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
```

*Let ("f", ["a"], Let ("x", [], Int 0),
Binop (Plus, Var "a",
Var "x"))),
App (Var "f", Int 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) ->`

*solution in
Piazza note 4/2*

```
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 $pp : \text{exp} \rightarrow \text{string}$, 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_binop binop = match binop with  
  And -> "&&" | LessThan -> "<" | ...
```

```
end rec pp e = match e with
```

```
  Operation(e1, binop, e2) ->
```

```
  | Array(e1, e2) ->
```

```
  |
```

Handwritten red annotations for the `Operation` case:

`"(" ^ pp e1 ^ pp_binop binop ^ pp e2 ^ ")"`

Handwritten red annotations for the `Array` case:

`"(" ^ pp e1 ^ "[" ^ pp e2 ^ "]" ^ ")"`