Chapitre 1
langage symbolique

Programmation Fonctionnelle, Licence d'informatique, 2021

Expressions, évaluation

- Habituellement, une expression s'évalue en une valeur

```
# 2 + 3 * 4;;

- : int = 14

# let x : int = 3 in 2 + x * 4;;

- : int = 14

# let b1 : bool = true and b2 : bool = false in not (b1 && b2);;

- : bool = true
```

Expressions, manipulation

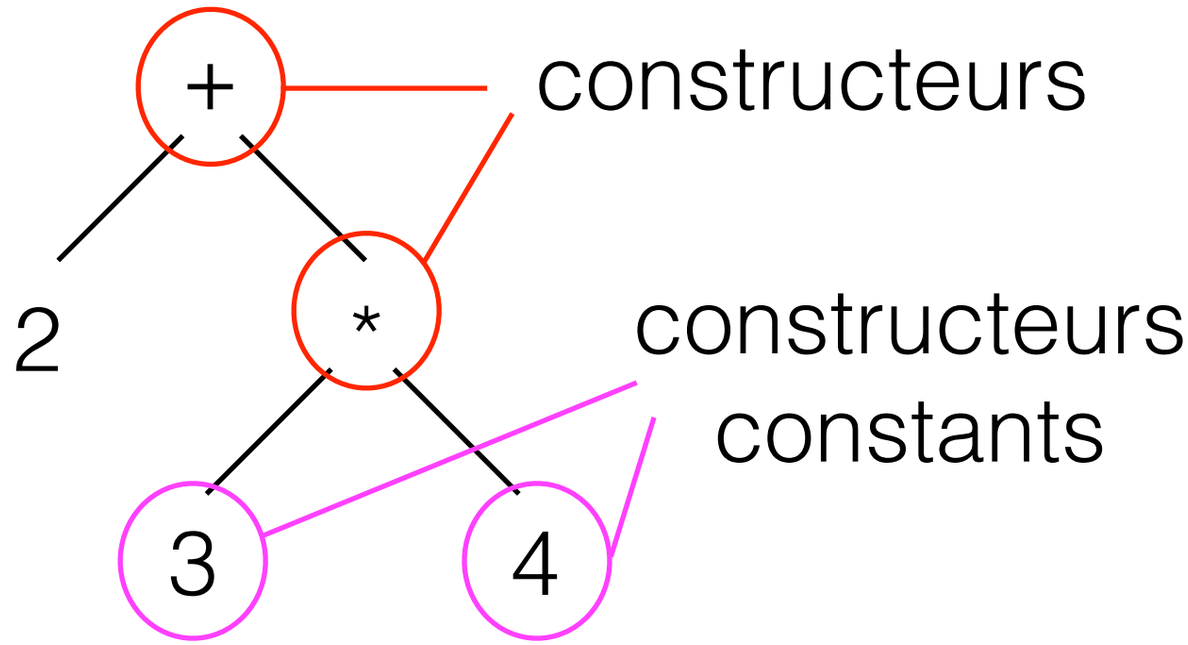
- Mais on peut aussi les transformer
  - développement de 2 * x * (3 + y) en…
    - évaluation partielle de (true && b1) || (b2 && false) en…
- Les expressions deviennent des données, appelées termes

Expressions, manipulation

- Mais on peut aussi les transformer
  - le développement de 2 * x * (3 + y) est (2 * x * 3) + (2 * x * y)
  - L'évaluation partielle de (true && b1) || (b2 && false) est b1
- Les expressions deviennent des données, appelées termes

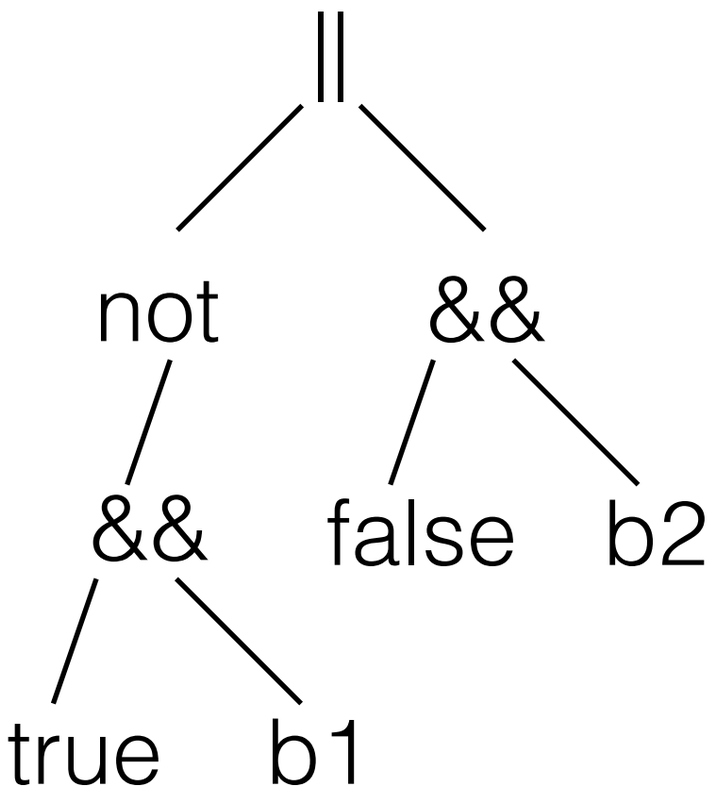
Termes

- 2 + 3 * 4
- Exercice, mettre sous forme arborescente les termes suivants :
  - not(true && b1) || (false && b2)
  - $-4x + 3y^2 - 2$

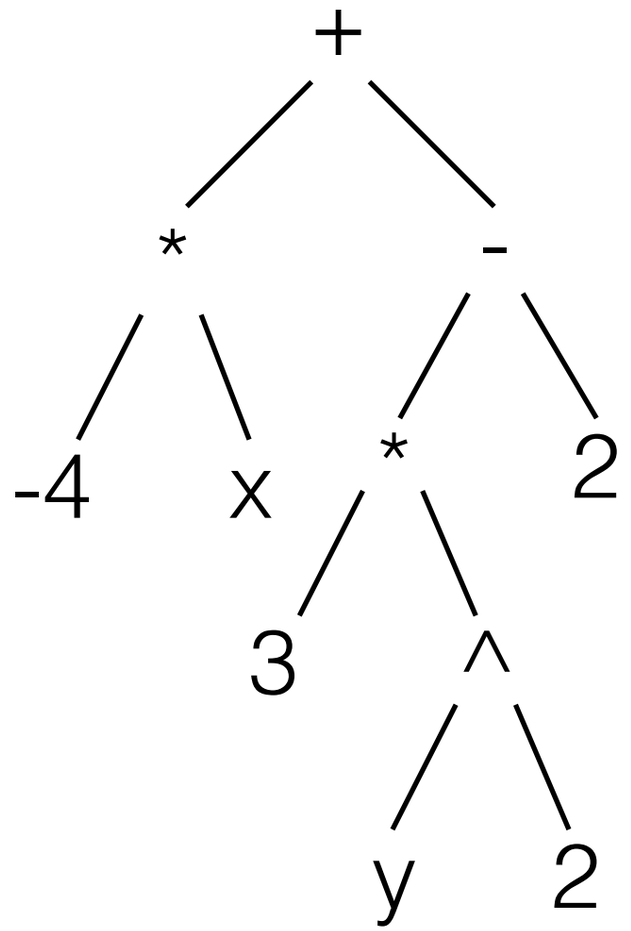


Termes

- not(true && b1) || (false && b2)



- $-4x + 3y^2 - 2$



Types sommes

- **type** nom_type = Constructeur1 [**of** nom_type1] | … | ConstructeurN [**of** nom_typeN]
- Remarque : les constructeurs commencent pas une majuscule
- Exemple
  - type ‘a list = [] | _::_ of ‘a * ‘a list
  - type int_exp =

```
  Valeur of int
| Plus of int_exp * int_exp
| Moins of int_exp * int_exp
| Minus of int_exp
| Mult of int_exp * int_exp;;

# Minus (Plus (Valeur 3, Valeur 5));;
- : int_exp = Minus (Plus (Valeur 3, Valeur 5))
```

Exercices

- Définissez les types sommes suivants :
  - Expressions booléennes avec les opérateurs « et », « ou », « non », des variables et les 2 constantes booléennes
  - Arbres généalogique, où chaque personne possède un père et une mère qui sont soit eux-mêmes des personnes aillant un arbre généalogique, soit inconnus

Correction

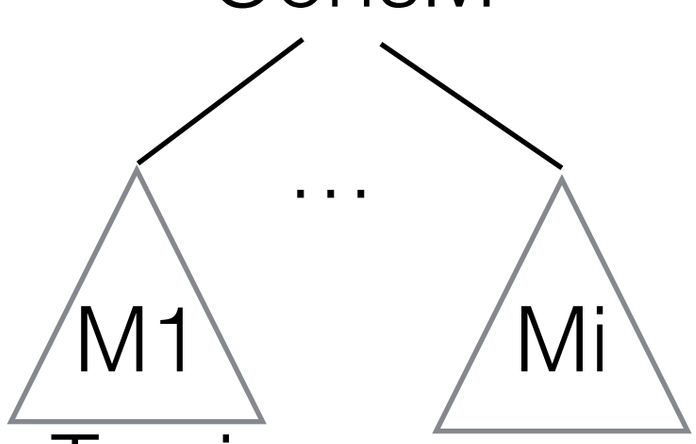
```
type bool_exp =
Variabe of string
| Et of bool_exp * bool_exp
| Ou of bool_exp * bool_exp
| Non of bool_exp
| Vrai
| Faux;;

type genealogie =
    Personne of string * genealogie * genealogie
  | Inconnu;;
```

Filtrage

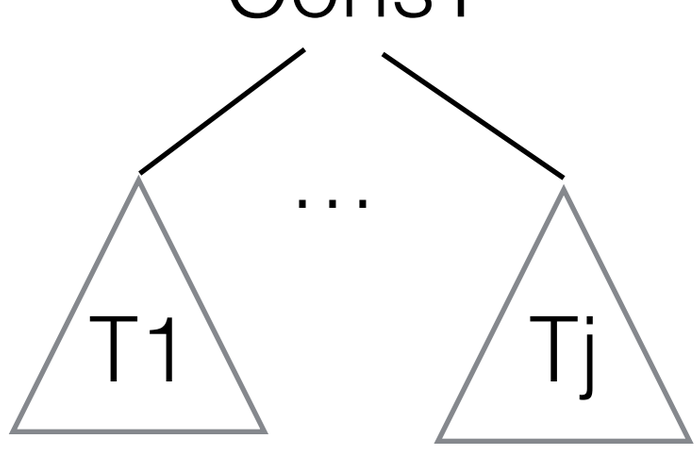
Motif de filtrage M =
terme avec variable

ConsM



Terme filtré T

ConsT



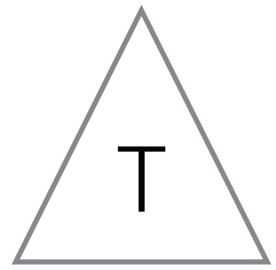
M filtre T ssi
si ConsM = ConsT (et donc i=j),
alors M1 filtre T1 et … et Mi filtre Tj
sinon, échec du filtrage

Filtrage

Motif de filtrage M =
terme avec variable

M

Terme filtré T



Le filtrage est en succès et M <- T

Remarque :
en OCaml, les variables commencent par une minuscule

Filtrage

Exemple :

- Motif = x + _, Terme = 2 + 3 * 4, succès du filtrage avec x = 2
- Motif = x * y, Terme = 2 + 3 * 4, échec du filtrage

Remarque : « _ » est une variable anonyme, elle n'est pas instanciée.

Exercice, réalisez les filtrages suivants :

- Motif = (x + y) * z, Terme = 2 + 3 * 4
- Motif = a et b, Terme = (Vrai et c1) ou (Faux et c2)
- Motif = x + y, Terme = 5 * 6 + 25

Correction

- Motif = (x + y) * z, Terme = 2 + 3 * 4

Échec, car les constructeurs sont « * » et « + »

- Motif = a et b, Terme = (Vrai et c1) ou (Faux et c2)

Échec, car les constructeurs sont différents « et » et « ou »

- Motif = x + y, Terme = 5 * 6 + 25

Succès, avec x = 5 * 6 et y = 25

Match

- **match** expression **with**

| motif1 -> expression1

…

| motifN -> expressionN

- Exemple :

```
# let developpe (exp : int_exp) : int_exp =
match exp with
| Mult(x, Plus(y, z)) -> Plus(Mult(x, y), Mult(x, z))
| Mult(Plus(x, y), z) -> Plus(Mult(x, z), Mult(y, z))
| _ -> exp
;;
# developpe (Mult (Plus (Valeur 2 , Valeur 3), Minus (Valeur 4)));;
- : int_exp =
Plus (Mult (Valeur 2, Minus (Valeur 4)), Mult (Valeur 3, Minus (Valeur 4)))
```

Exercices

- Écrire une fonction qui retourne la liste des noms des parents d'un arbre généalogique.
- Écrire les fonctions sur les listes qui :
  - compte les éléments d'une liste,
  - construit la sous-liste des entiers pairs d'une liste d'entiers.

Corrections

```
let parents(exp : genealogie) : string list =
match exp with
| Inconnu -> []
| Personne (nom, Inconnu, Inconnu) -> []
| Personne (nom, Personne (pere, _, _), Inconnu) ->
[pere]
| Personne (nom, Inconnu, Personne (mere, _, _)) ->
[mere]
| Personne (nom, Personne (pere, _, _), Personne
(mere, _, _)) -> [pere ; mere]
;;
```

```
let rec compte (la_list : ‘a list) : int =
match la_list with
| [] -> 0
| _ :: suite -> (compte suite) + 1
;;

let rec sous_liste (la_list : int list) : int list=
match la_list with
| [] -> []
| elem :: suite ->
  if elem mod 2 = 0
  then elem :: (sous_liste suite)
  else sous_liste suite
;;
```