

(5) Arbres binaires – Applications

Programmation fonctionnelle (LU2IN019)

Licence d'informatique
2022/2023

Mathieu Jaume – Adrien Koutsos



Expressions arithmétiques

- **les mêmes que dans l'UE de mathématiques discrètes**

- ▶ l'ensemble des expressions arithmétiques défini sur un ensemble de symboles de variable est défini inductivement par :

(base) tout symbole de variable est une expression arithmétique

(base) tout entier est une expression arithmétique

(induction) si e est une expression arithmétique, alors $-e$ est une expression arithmétique

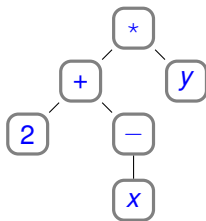
si e_1 et e_2 sont des expressions arithmétiques, alors $e_1 + e_2$, $e_1 * e_2$ et e_1 / e_2 sont des expressions arithmétiques

Expressions arithmétiques : syntaxe

- **syntaxe concrète** : utilisation de parenthèses pour indiquer la portée des opérateurs (ou règles implicites de priorité)
- **syntaxe abstraite** : représentation d'une expression par un arbre
 - ▶ la structure d'arbre indique la portée des opérateurs (plus besoin de parenthèses ni de règles de priorité)

$(2 + (-x)) * y$

syntaxe concrète

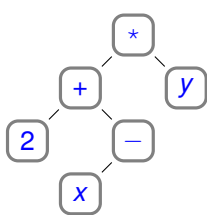


AST : abstract syntax tree

Expressions arithmétiques et arbres binaires

● 1ère possibilité :

représentation des expressions arithmétiques par des *arbres binaires*



- les étiquettes des nœuds de l'arbre sont des symboles de variable, des entiers ou des opérateurs
- les opérateurs arithmétiques $+$, $*$ et $/$ sont des opérateurs binaires
- l'opérateur arithmétique $-$ est unaire : le fils gauche du nœud étiqueté par $-$ est l'opérande de $-$, son fils droit est l'arbre vide

● difficulté : tout les arbres binaires ne correspondent pas à une expressions arithmétique

- ▶ l'arbre binaire vide ne correspond à aucune expression
- ▶ les variables et les entiers étiquettent seulement les feuilles
- ▶ les opérateurs arithmétiques étiquettent seulement les « nœuds internes »
 - ★ ces nœuds ont deux fils non vides pour les opérateurs $+$, $*$ et $/$
 - ★ ces nœuds ont un fils gauche non vide et un fils droit vide pour l'opérateur $-$

Expressions arithmétiques et arbres binaires

- type des étiquettes ('a désigne le type des symboles de variable)

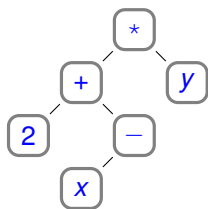
```
type 'a label_exparith = L_var of 'a | L_Cste of int  
                        | L_Plus | L_Mult | L_Div | L_opp
```

- type des arbres binaires étiquetés par des valeurs de type 'a label_exparith:

```
type 'a btree =  
  | Empty  
  | Node of 'a * ('a btree) * ('a btree)
```

```
type 'a exparith_tree = ('a label_exparith) btree
```

- exemple



```
let e =  
  Node(L_Mult,  
    Node(L_Plus,  
      Node(L_Cste 2, Empty, Empty),  
      Node(L_opp,  
        Node(L_var "x", Empty, Empty),  
        Empty)),  
    Node(L_var "y", Empty, Empty))
```

Expressions arithmétiques et arbres binaires

- **rappel** : tout les arbres binaires étiquetés par des valeurs de type 'a label_exparith ne correspondent pas à une expression arithmétique
 - ▶ l'arbre binaire vide ne correspond à aucune expression
 - ▶ les variables et les entiers étiquettent seulement les feuilles
 - ▶ les opérateurs arithmétiques étiquettent seulement les « nœuds internes, et doivent avoir autant de fils que leur arité
- vérification qu'un arbre binaire représente une expression arithmétique :

```
let rec wellformed (e: 'a exparith_tree) : bool =  
  match e with  
  | Empty -> false  
  | Node(e, g, d) ->  
    match e with  
    | L_var(x) -> g = Empty && d = Empty  
    | L_Cste(k) -> g = Empty && d = Empty  
    | L_opp -> wellformed g && d = Empty  
    | _ -> wellformed g && wellformed d
```

Expressions arithmétiques

- comment exprimer ces contraintes directement dans la définition du type des arbres représentant des expressions arithmétiques ?

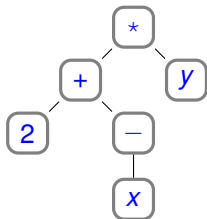
↪ *définition d'un type spécifique pour les AST d'expressions arithmétiques*

```
type 'a exparith =  
  | Var   of 'a  
  | Cste  of int  
  | Opp   of 'a exparith  
  | Plus  of 'a exparith * 'a exparith  
  | Mult  of 'a exparith * 'a exparith  
  | Div   of 'a exparith * 'a exparith
```

- **Opp**, **Plus**, **Mult** et **Div** sont des constructeurs d'expressions arithmétiques à partir d'autres expressions arithmétiques

Expressions arithmétiques

- Exemple :



```
# let ex =  
    Mult (Plus (Cste 2, Opp(Var "x")),  
          Var "y");;  
  
val ex : string exparith = ...
```

- **exemple de fonction** : nombre d'occurrences de symboles de variable dans une expression arithmétique

```
let rec nb_var (e : 'a exparith) : int =  
  match e with  
  | Var x   -> 1  
  | Cste n  -> 0  
  | Opp e0  -> nb_var e0  
  | Plus (e1,e2) -> nb_var e1 + nb_var e2  
  | Mult (e1,e2) -> nb_var e1 + nb_var e2  
  | Div  (e1,e2) -> nb_var e1 + nb_var e2  
  
# nb_var ex;;  
- : int = 2
```

Évaluation d'une expression arithmétique

- pour évaluer une expression arithmétique il faut connaître la valeur des variables de l'expression
 - ▶ **environnement d'évaluation** : liste d'association composée de paires (x, k) où x est un symbole de variable et k est la valeur associée à x

exemple :

```
# let envxy = [ ("x", 2); ("y", 5) ];;  
val envxy : (string * int) list = ...
```

Expressions arithmétiques : évaluation

```
let rec eval_e (env : ('a*int) list) (e : 'a exparith) : int =
  match e with
  | Var x   -> List.assoc x env
  | Cste n  -> n
  | Opp e0  -> - (eval_e env e0)
  | Plus (e1, e2) -> (eval_e env e1) + (eval_e env e2)
  | Mult (e1, e2) -> (eval_e env e1) * (eval_e env e2)
  | Div  (e1, e2) -> (eval_e env e1) / (eval_e env e2)

# eval_e envxy ex;;
- : int = 0

# eval_e envxy (Plus (Var("x"), Var("z")));;
Exception: Not_found.
```

Expressions arithmétiques : évaluation

Version alternative avec message d'erreur amélioré.

```
let eval_var (env : (string*int) list) (x : string) : int =
  try List.assoc x env with
  | Not_found -> raise (Invalid_argument (x ^ " indefini"))

let rec eval_e (env : ('a*int) list) (e : 'a exparith) : int =
  match e with
  | Var x -> eval_var env x
  | Cste n -> n
  | Opp e0 -> - (eval_e env e0)
  | Plus (e1,e2) -> (eval_e env e1) + (eval_e env e2)
  | Mult (e1,e2) -> (eval_e env e1) * (eval_e env e2)
  | Div (e1,e2) -> (eval_e env e1) / (eval_e env e2)

# eval_e envxy ex;;
- : int = 0

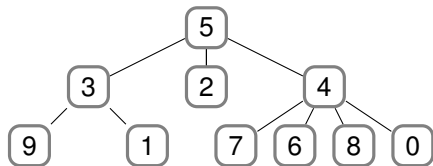
# (eval_e envxy (Plus (Var("x"), Var("z"))));;
Exception: Invalid_argument "z indefini".
```

- tous les opérateurs ne sont pas binaires
 - ▶ certains langages d'expressions utilisent des opérateurs unaires, binaires, ternaires, etc.
 - ★ `not` : opérateur booléen unaire
 - ★ `if e_1 then e_2 else e_3` : `if-then-else` est un opérateur ternaire
- notion de termes du premier ordre
(à venir dans l'UE de logique du 2ème semestre)*
- tous les arbres ne sont pas binaires
 - ▶ arbres généraux

Arbres généraux

Arbres généraux

- arbre général : arbre non vide dans lequel chaque nœud a un nombre quelconque de sous-arbres



à chaque nœud est associé

- une étiquette
- la liste de ses sous-arbres

- définition OCaml du type des arbres généraux :

```
type 'a gtree = GNode of 'a * (('a gtree) list)
```

```
let gt =
```

```
  GNode(5, [GNode(3, [GNode(9, []); GNode(1, [])]);  
            GNode(2, []);  
            GNode(4, [GNode(7, []); GNode(6, []);  
                      GNode(8, []); GNode(0, [])])])])
```


- une **forêt** est une liste d'arbres
- un **arbre** est défini par :
 - ▶ l'étiquette de sa racine et la **forêt** qui constitue la liste des sous-arbres de sa racine

définitions mutuellement récursives

- exemple : recherche d'un élément dans un arbre général

```
let rec gtree_mem (x : 'a) (t : 'a gtree) : bool =  
  match t with  
  | GNode (e, lt) -> e = x || forest_mem e lt  
  
and forest_mem (x : 'a) (f : ('a gtree) list) : bool =  
  match f with  
  | [] -> false  
  | hf :: tf -> gtree_mem x hf || forest_mem x tf  
  
val gtree_mem : 'a -> 'a gtree -> bool = <fun>  
val forest_mem : 'a -> 'a gtree list -> bool = <fun>
```

Formules de la logique propositionnelle

Formules de la logique propositionnelle

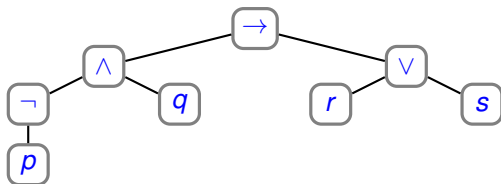
- (à venir) dans l'UE de mathématiques discrètes
 - ▶ ensemble de *variables propositionnelles* $\mathcal{P} = \{p, q, r, \dots\}$
 - ★ désignent des énoncés atomiques (*a priori* indépendants les uns des autres) qui peuvent être vrais ou faux
 - ▶ ensemble \mathbb{F} des formules de la logique des propositions défini inductivement à partir de $\mathcal{P} \cup \{\text{true}, \text{false}, \neg, \wedge, \vee, \rightarrow\}$:
 - ★ toute variable propositionnelle de \mathcal{P} est une formule de \mathbb{F} ,
 - ★ les constantes **true** et **false** sont des formules de \mathbb{F} ,
 - ★ si $F \in \mathbb{F}$ alors $\neg F \in \mathbb{F}$ (négation),
 - ★ si $F_1, F_2 \in \mathbb{F}$ alors $(F_1 \wedge F_2) \in \mathbb{F}$ (conjonction : et),
 - ★ si $F_1, F_2 \in \mathbb{F}$ alors $(F_1 \vee F_2) \in \mathbb{F}$ (disjonction : ou),
 - ★ si $F_1, F_2 \in \mathbb{F}$ alors $(F_1 \rightarrow F_2) \in \mathbb{F}$ (implication).
- exemple
 - ▶ $\mathcal{P} = \left\{ \begin{array}{ll} p : \text{il pleut,} & q : \text{je suis en vacances,} \\ r : \text{je vais à la plage,} & s : \text{je fais de la logique} \end{array} \right\}$
 - ▶ formule :
Si il ne pleut pas et que je suis en vacances,
alors je vais à la plage ou je fais de la logique. $(\neg p \wedge q) \rightarrow (r \vee s)$

Formules de la logique propositionnelle

- formule :

$$(\neg p \wedge q) \rightarrow (r \vee s)$$

- arbre de syntaxe abstraite



- ▶ c'est un **arbre général** mais
 - ★ les feuilles sont étiquetées par des éléments de $\mathcal{P} \cup \{\text{true}, \text{false}\}$
 - ★ les nœuds étiquetés par \neg ont exactement un sous-arbre
 - ★ les nœuds étiquetés par \vee , \wedge et \rightarrow ont exactement deux sous-arbres

Formules de la logique propositionnelle

- ensemble 'a formul des formules défini inductivement à partir de l'ensemble des variables de 'a :
 - ▶ si p est une variable, alors **Prop**(p) est une formule
 - ▶ les constantes **Vrai** et **Faux** sont des formules
 - ▶ si F est une formule alors **Non**(F) est une formule
 - ▶ si F1 et F2 sont des formules alors **Et**(F1, F2) est une formule
 - ▶ si F1 et F2 sont des formules alors **Ou**(F1, F2) est une formule
 - ▶ si F1 et F2 sont des formules alors **Impl**(F1, F2) est une formule
- en Ocaml

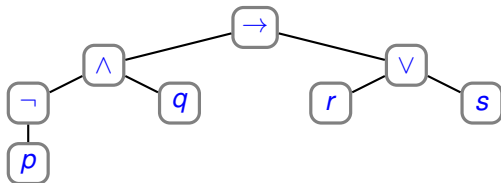
```
type 'a formul =  
| Prop of 'a  
| Vrai  
| Faux  
| Non  of 'a formul  
| Et   of 'a formul * 'a formul  
| Ou   of 'a formul * 'a formul  
| Impl of 'a formul * 'a formul
```

Formules de la logique propositionnelle

- formule :

$$(\neg p \wedge q) \rightarrow (r \vee s)$$

- arbre de syntaxe abstraite



- valeur de type **char** formul :

```
# let f =  
  Impl (Et (Non (Prop 'p'), Prop 'q'),  
        Ou (Prop 'r', Prop 's'));;  
  
val f : char formul = ...
```

Évaluation d'une formule de la logique propositionnelle

- pour évaluer une formule il faut connaître la « valeur de vérité » des variables de la formule
 - ▶ **valuation/interprétation** : liste d'association composée de paires (p, b) où p est un symbole de variable et b est la valeur booléenne associée à p

exemple :

```
# let envpqrs = [('p', true); ('q', false);  
                ('r', true); ('s', false)];;  
  
val envpqrs : (char * bool) list = ...
```

Formules de la logique propositionnelle : évaluation

évaluation d'une formule de la logique propositionnelle

```
let rec eval_f (env : ('a*bool) list) (f : 'a formul) : bool =
  match f with
  | Prop p -> List.assoc p env
  | Vrai   -> true
  | Faux   -> false
  | Non f0 -> not (eval_f env f0)
  | Et    (f1,f2) -> eval_f env f1 && eval_f env f2
  | Ou    (f1,f2) -> eval_f env f1 || eval_f env f2
  | Impl (f1,f2) -> (not (eval_f env f1)) || eval_f env f2

# eval_f envpqrs f;;
- : bool = true
```


Formules de la logique propositionnelle

- une formule f de type `'a formul` est une formule valide si `eval_f env f = true` pour toute interprétation `env` de type `('a * bool) list`
- programmation de la fonction de signature :

```
is_valid_f (f : 'a formul) : bool
```

- ▶ utilisation des fonctions du TME sur les ensembles finis représentés par des listes

en direct au tableau ...