Option informatique, cours 4

Structures de données 2/3 : arbres

C. MULLAERT

Lycée Saint-Louis

Année 2024-2025

Plan du cours

1. La structure d'arbre : généralités
2. Algorithmes de parcours d'un arbre et applications
3. Applications : utilisation de la structure d'arbre

La structure d'arbre : généralités

Structure d'arbre

Dans le cours précédent, on a défini la structure de liste de la façon suivante :

```
type 'a liste = Vide | Noeud of 'a*'a liste;;
```

Ce type de donnée a les propriétés suivantes :

- C'est un type **récursif** (car il fait référence à lui-même dans sa définition)
- C'est un type de données **polymorphe** (car on peut utiliser ce type indifféremment pour des listes d'entiers, de flottants... ou même de listes)

C'est également une structure **linéaire** car il n'y a qu'une seule occurrence récursive. En conséquence, le parcours récursif d'une liste est simple.

Structure d'arbre

L'arbre est une structure récursive, mais non linéaire. Un **nœud** peut avoir un ou plusieurs arbres fils. En voici une première définition :

```
type arbre = Vide | Noeud of arbre*arbre;;
```

Dans cette définition, chaque nœud possède exactement deux fils (éventuellement vides). Par rapport aux listes, cette structure **ramifiée** entraine deux conséquences importantes :

- La notion de longueur n'a plus de sens car il peut exister plusieurs chemins à partir d'un nœud donné. On parlera de **hauteur** pour le chemin le plus long qu'il est possible de construire.
- Les fonctions qui manipulent un arbre vont le parcourir de façon récursive, mais il y a en pratique **plusieurs** façon de le faire.

Structure d'arbre : variantes 1/4

En fonction du problème que l'on souhaite traiter, il peut exister plusieurs variantes de la structure d'arbre, plus ou moins adaptée. En voici des exemples :

```
(* Arbre binaire non étiqueté *)
type arbre = Vide | Noeud of arbre*arbre;;
(* Arbre binaire étiqueté avec des entiers *)
type arbre = Vide | Noeud of int*arbre*arbre;;
(* Arbre binaire étiqueté polymorphe *)
type 'a arbre = Vide | Noeud of 'a*'a arbre*'a arbre;;
```

Structure d'arbre : variantes 2/4

On peut également généraliser les arbres binaires en autorisant les à avoir un nombre variable de fils :

```
(* Arbre non étiqueté *)
type arbre = Vide | Noeud of arbre list;;
(* Arbre étiqueté *)
type 'a arbre = Vide | Noeud of 'a*'a arbre list;;
```

Structure d'arbre : variantes 3/4

On peut aussi souhaiter que seules les feuilles (noeuds sans fils) soient étiquetées.

```
type 'a arbre = Feuille of 'a
        | Noeud of 'a arbre * 'a arbre;;
```

Attention, avec cette définition, l'arbre vide n'existe pas et chaque nœud possède exactement deux fils. On parle d'arbre binaire **strict**.

Le constructeur `Vide` servait auparavant à désigner à la fois l'absence d'un ou deux fils d'un nœud et l'arbre vide. C'est maintenant le constructeur `Feuille` qui remplit cette fonction.

Structure d'arbre : variantes 4/4

Enfin, voici une version où les feuilles ont une étiquette d'un type différent de celui des nœuds internes (avec cette définition, l'arbre vide n'existe toujours pas) :

```
type ('a, 'b) arbre = Feuille of 'a
                    | Noeud of 'b * ('a, 'b) arbre list;;
```

La conclusion sur toutes ces variantes est qu'il faut :

- préciser de quoi on parle lorsqu'on manipule un arbre
- utiliser la définition adaptée au problème posé
- adapter les fonctions de parcours en conséquence

Structure d'arbre : vocabulaire

Le nœud initial d'un arbre s'appelle la **racine**. La distance entre un nœud est la racine s'appelle la **profondeur** du nœud. Un nœud sans aucun fils est appelé **feuille**.

La **hauteur** d'un arbre est la profondeur maximale de ses nœuds. Par convention, la hauteur de l'arbre vide est -1.

Un arbre binaire est dit **strict** si tous les nœuds possèdent 0 ou 2 fils.

Un arbre binaire strict est dit **parfait** si toutes les feuilles ont la même profondeur.

Structure d'arbre : propriétés

On note n le nombre de nœuds et f le nombre de feuilles d'un arbre binaire non vide de hauteur h. Les propriétés suivantes sont à connaitre et se montrent par récurrence :

- Si l'arbre est strict, alors $f = n + 1$.
- Si l'arbre est parfait, alors $f = 2^h$ et $n = 2^h - 1$.
- Dans tous les cas, $f \leq 2^h$

Algorithmes de parcours d'un arbre et applications

Considérations générales

On considère le type d'arbre suivant :

```
type 'a arbre = Feuille of 'a
        | Noeud of 'a arbre * 'a arbre;;
```

Comme pour les listes, les fonctions qui manipulent les arbres vont le faire **récursivement**. Le cas de base correspond à la feuille et, dans le cas d'un nœud, la fonction comportera deux appels récursifs pour traiter les deux fils.

Toujours comme pour les listes, on fera un usage intensif du **matching** en utilisant les constructeurs `Feuille` et `Noeud` pour distinguer les cas.

Premier exemple

```
type 'a arbre = Feuille of 'a
        | Noeud of 'a arbre * 'a arbre;;
let rec hauteur = function
        | Feuille (_) -> 0
        | Noeud (g,d) -> 1+ max (hauteur g) (hauteur d);;
```

Sur ce premier exemple, l'ordre des appels récursifs gauche et droite n'a pas d'importance.

Autres exemples

```
type 'a arbre = Feuille of 'a
        | Noeud of 'a arbre * 'a arbre;;
let rec nb_noeuds = function
        | Feuille (_) -> 0
        | Noeud (g,d) -> 1+ nb_noeuds g + nb_noeuds d;;
let rec nb_feuilles = function
        | Feuille (_) -> 1
        | Noeud (g,d) -> nb_feuilles g + nb_feuilles d;;
```

Ici encore, l'ordre des appels récursifs gauche et droite n'a pas d'importance.

Exercice

Recoder les fonctions hauteur et nombre de nœuds lorsque l'arbre est défini par les types suivants :

```
(* Arbre binaire non étiqueté *)
type arbre = Vide | Noeud of arbre*arbre;;

(* Arbre général non étiqueté *)
type arbre = Vide | Noeud of arbre list;;
```

Parcours préfixe, infixe et postfixe

Dans certains cas, on réalise une action sur chaque nœud d'un arbre et l'ordre de ces actions a une importance. Considérons par exemple la définition suivante d'arbre binaire étiqueté :

```
type arbre = Vide | Noeud of int*arbre*arbre;;
```

Si on souhaite afficher toutes les étiquettes, ce qui est affiché dépend de l'ordre dans lequel on place les appels récursifs par rapport à l'affichage.

Parcours préfixe

Dans le parcours **préfixe**, on commence par afficher la valeur de l'étiquette, puis on appelle la fonction affiche récursivement sur le fils gauche et le fils droit :

```
(* version préfixe *)
let rec affiche = function
  | Vide -> ()
  | Noeud (a,g,d) -> print_int a; affiche g; affiche d;;
```

Cette fonction est du type `arbre -> unit` et agit par **effet de bord**. L'opérateur `;` permet de réaliser plusieurs opérations en séquence. Il n'a d'intérêt que lorsque l'évaluation des expressions entraine un effet de bord.

Parcours infixe et postfixe

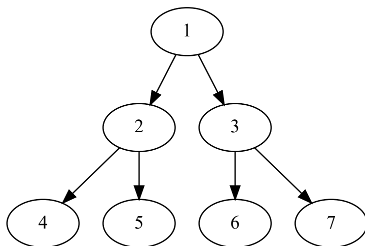
Dans les parcours **infixe** et **postfixe**, l'affichage se fait entre les deux appels récursifs, ou bien à la fin :

```
(* version infixe *)
let rec affiche = function
 | Vide -> ()
 | Noeud (a,g,d) -> affiche g; print_int a; affiche d;;

(* version postfixe *)
let rec affiche = function
 | Vide -> ()
 | Noeud (a,g,d) -> affiche g; affiche d; print_int a;;
```

Par exemple, avec l'arbre ci-dessous, les sorties des trois parcours sont :

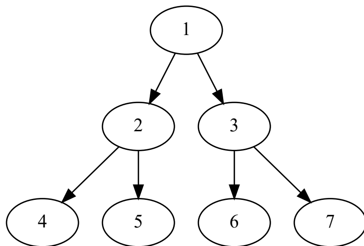
- 1245367 pour le parcours préfixe
- 4251637 pour le parcours infixe
- 4526731 pour le parcours postfixe



Parcours préfixe, infixe et postfixe

On a illustré les parcours préfixe, infixe et postfixe avec des fonction d'affichage en utilisant des effets de bord.

Écrire trois fonctions qui renvoient la liste des étiquettes d'un arbre en le parcourant de façon préfixe, infixe et postfixe. Ces fonctions seront du type `int arbre -> int list`.



Applications : utilisation de la structure d'arbre

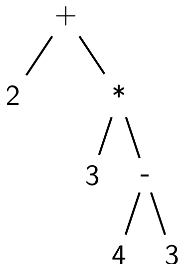
utilisation de la structure d'arbre

Dans cette partie, on va voir des exemples où la structure d'arbre permet naturellement de traiter un problème algorithmique :

- Le codage et la manipulation d'expressions algébriques
- Les arbres de recherche

Expressions algébriques

La structure d'arbre permet naturellement de représenter des expressions faisant intervenir des nombres et les quatre opérations $+$, $-$, $*$ et $/$.
Par exemple, l'expression $2 + (3 * (4 - 3))$ peut se représenter par l'arbre suivant, dans lequel les feuilles contiennent les nombres et les nœuds contiennent les opérateurs :



Expressions algébriques

On remarque que l'expression $2 + (3 * (4 - 3))$ ne contient que des opérateurs binaires. Elle est donc représentée par un arbre binaire :

```
type op = Somme | Prod | Diff | Div;;
type exp = Feuille of float | Noeud of op*exp*exp;;
let e = Noeud (Somme,Feuille 2.,
               Noeud(Prod,Feuille 3.,
               Noeud(Diff, Feuille 4.,Feuille 3.)));;
```

Cette représentation est utile pour effectuer des opérations formelles sur ces expressions.

Expressions algébriques

Il est également possible de simplifier l'écriture en codant l'opérateur en considérant plusieurs types de Noeud plutôt que dans une étiquette :

```
type exp = Const of float
        | Somme of exp*exp
        | Diff of exp*exp
        | Prod of exp*exp
        | Div of exp*exp ;;
let e = Somme (Const 2.,
               Prod (Const 3.,
               Diff (Const 4.,Const 3.)));;
```

Cette nouvelle définition permet éventuellement d'étendre de façon simple les expressions qui comportent des opérateur uniaires (ex. la racine carrée).

On peut alors écrire une fonction qui évalue une expression, en parcourant la structure de façon récursive :

```
let rec eval = function
        | Const a -> a
        | Somme (a,b) -> (eval a) +. (eval b)
        | Diff (a,b) -> (eval a) -. (eval b)
        | Prod (a,b) -> (eval a) *. (eval b)
        | Div (a,b) -> (eval a) /. (eval b) ;;
```

Expressions algébriques

On peut aussi formater une expression avec des parenthèses :

```
let rec format = function
  | Const a -> string_of_float a
  | Somme (a,b) -> "("^(format a)^"+"^(format b)^")"
  | Diff (a,b) -> "("^(format a)^"-"^(format b)^")"
  | Prod (a,b) -> "("^(format a)^"*"^(format b)^")"
  | Div (a,b) -> "("^(format a)^"/"^(format b)^")";;
```

On remarque que l'opérateur est placé en position infixe. D'autres types de notations sont possibles : la notation **polonaise** correspond à un parcours préfixe et la notation **polonaise inversée ou NPI** correspond à un parcours postfixe. Ces dernières ne nécessitent pas de parenthèses.

Notations polonaises (préfixe)

```
let rec np = function
 | Const a -> string_of_float a
 | Somme (a,b) -> "+"^(np a)^(np b)
 | Diff (a,b) -> "-"^(np a)^(np b)
 | Prod (a,b) -> "*"^(np a)^(np b)
 | Div (a,b) -> "/"^(np a)^(np b);;
```

Avec l'expression $2 + (3 * (4 - 3))$, l'évaluation de `np` renvoit "+2.*3.-4.3.".

Notation polonaise inversée (postfixe)

```
let rec npi = function
  | Const a -> string_of_float a
  | Somme (a,b) -> (npi a)^(npi b)^"+"
  | Diff (a,b) -> (npi a)^ (npi b)^"-"
  | Prod (a,b) -> (npi a)^(npi b)^"*"
  | Div (a,b) -> (npi a)^(npi b)^"/";;
```

Avec l'expression $2 + (3 * (4 - 3))$, l'évaluation de `npi` renvoit "2.3.4.3.-*+".

Arbres binaires de recherche

On rappelle que la complexité de la recherche d'un élément dans une liste est **linéaire** en la longueur de la liste. On réalise pour cela un parcours récursif de la liste.

En python, dans une liste triée, la recherche d'un élément est d'une complexité **logarithmique** à l'aide d'un algorithme itératif par dichotomie.

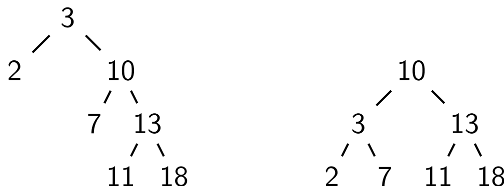
Pour faire de même en Ocaml, il faut utiliser un autre type de donnée que la structure de liste (que l'on ne peut pas couper en deux en temps constant). C'est la structure d'**arbre binaire de recherche**.

Arbres binaires de recherche

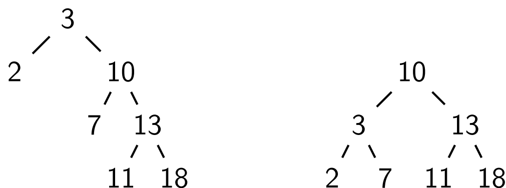
Un **arbre binaire de recherche** est un arbre binaire étiqueté tel que :

- Les étiquettes sont à valeur dans un ensemble totalement ordonné
- L'étiquette de tout nœud interne est supérieure à celle de ses descendants du sous-arbre gauche
- L'étiquette de tout nœud interne est inférieure à celle de ses descendants du sous-arbre droit

Par exemple, les arbres suivants sont des arbres binaires de recherche :



Considérons les deux arbres binaires de recherche suivants :



Il est important de remarquer que :

- La recherche d'un élément par parcours récursif va nécessiter au plus $h+1$ étapes, où h est la hauteur de l'arbre. En ce sens, l'arbre de gauche est "meilleur" car plus équilibré.
- Le parcours infixe d'un tel arbre réalise l'opération de tri de l'ensemble des étiquettes avec une complexité **linéaire** en n, nombre de nœuds.

On utilise la définition suivante d'arbre binaire :

```
type abin = Vide | Noeud of int*abin*abin;;
```

On codera en TP les fonctions qui testent si un arbre binaire est un arbre de recherche, qui manipulent ces arbres.