Exercices « Programmation récursive »
Deuxième saison
UPMC Cycle L
Revision: 1.21

Anne Brygoo, Maryse Pelletier, Christian Queinnec, Michèle Soria
Université Paris 6 — Pierre et Marie Curie

septembre 2005 — janvier 2006

Ce lot d'exercices correspond aux quatre dernières semaines d'enseignement. Il est divisé en deux parties

1. **Arbres Binaires**

   Les arbres binaires sont manipulés à travers une barrière d'abstraction. L'objectif principal des exercices de cette partie est de faire de la récursion binaire. Mais il faut y voir aussi une introduction à l'algorithmique, par exemple avec les arbres binaires de recherche.
   - Les exercices « Arbres Binaires » (page 2) et « Liste des nœuds » (page 4), applications directes du cours, sont indispensables pour apprendre à manipuler les arbres binaires.
   - Ensuite on étudie des arbres binaires qui représentent des expressions arithmétiques. Les exercices « Représentation d'une expression arithmétique » (page 5), « Evaluation d'une expression arithmétique » (page 6) et « Dérivation formelle » (page 7) sont assez simples. L'exercice « Dérivation formelle paramétrée » (page 8) est plus complexe et marie la manipulation des arbres et celle des listes.
   - Enfin on travaille sur des arbres binaires de recherche. L'exercice « Arbres Binaires de Recherche » (page 10), application directe du cours, implante plusieurs opérations de base de cette structure de données. L'exercice « Recherche d'un point dans un intervalle » (page 13) utilise cette structure de données pour traiter un problème de géométrie.

2. **Arbres Généraux**

   Les arbres généraux et forêts (listes d'arbres généraux) sont manipulés à travers une barrière d'abstraction. L'objectif principal des exercices de cette partie est de faire de la récursion croisée, en suivant les définitions mutuellement récursives des arbres et des forêts ; on insiste aussi sur l'utilisation de `map` et `reduce` pour traiter ces arbres.
   - Les exercices « Arbres Généraux » (page 14), « Profondeurs » (page 16), « Nombre de Strahler » (page 16) et « Listes des nœuds » (page 17) sont des applications directes du cours, qui sont nécessaires pour apprendre à manipuler les arbres généraux et les forêts.
   - L'exercice « Représentation des arbres généraux par des arbres binaires » (page 18) est plus difficile : il travaille sur l'implantation de la barrière d'abstraction des arbres généraux à l'aide de celle des arbres binaires.
   - Enfin l'exercice « Arbres cardinaux » (page 20) présente une autre famille d'arbres, sur laquelle on peut revoir synthétiquement toutes les notions étudiées sur les arbres, et implanter une barrière d'abstraction. Ces arbres cardinaux, qui sont largement utilisés en géométrie algorithmique pour représenter des images « bitmap », nous permettent aussi de refaire une incursion dans le graphique de Scheme.

1 Arbres Binaires

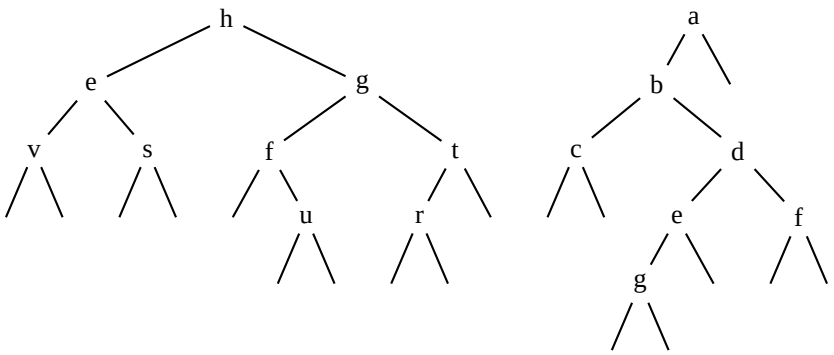
1.1 Exercices de base

Exercice 1 – Arbres binaires

Les arbres binaires sont des arbres dans lesquels chaque nœud a zéro ou deux fils. On peut donner la définition récursive suivante : un *arbre binaire* est

- soit vide,
- soit formé :
  - d'un nœud racine, portant une étiquette,
  - d'un sous-arbre gauche qui est un arbre binaire,
  - d'un sous-arbre droit qui est un arbre binaire.

La figure suivante montre deux arbres binaires, que l'on nommera par la suite B1 (celui de gauche) et B2 :



Le but de cet exercice est de calculer différentes caractéristiques des arbres binaires : nombre de nœuds, nombre de nœuds à profondeur *k*, nombre de Stralher ..., en manipulant les arbres binaires à travers une barrière d'abstraction.

Le type `ArbreBinaire[alpha]` représente les arbres binaires dont les étiquettes sont de type `alpha` . La barrière d'abstraction permettant de manipuler des arbres binaires est formée

- des constructeurs `(ab-vide)` pour construire l'arbre vide et `(ab-noeud e B1 B2)` pour construire un arbre à partir d'une étiquette `e`, et de deux arbres binaires `B1` et `B2` ;
- un reconnaisseur `ab-noeud?` pour reconnaître si un arbre est non vide ;
- des accesseurs sur les arbres non vide :
  - `(ab-etiquette B)` pour accéder à l'étiquette de la racine de `B` ;
  - `(ab-gauche B)` pour accéder au sous-arbre gauche de `B` ;
  - `(ab-droit B)` pour accéder au sous-arbre droit de `B`.

Voici les spécifications des fonctions de la barrière d'abstraction

```
;;; ab-noeud : alpha * ArbreBinaire[alpha] * ArbreBinaire[alpha] -> ArbreBinaire[alpha]
;;; (ab-noeud e B1 B2) rend l'arbre binaire formé de la racine d'étiquette e,
;;; du sous-arbre gauche B1 et du sous-arbre droit B2

;;; ab-vide :  -> ArbreBinaire[alpha]
;;; (ab-vide) rend l'arbre binaire vide

;;; ab-noeud? : ArbreBinaire[alpha] -> bool
;;; (ab-noeud? B) rend vrai ssi B n'est pas l'arbre vide

;;; ab-etiquette : ArbreBinaire[alpha]/non vide/-> alpha
;;; (ab-etiquette B) rend l'étiquette de la racine de l'arbre B
```

```
;;; ab-gauche : ArbreBinaire[alpha]/non vide/ -> ArbreBinaire[alpha]
;;; (ab-gauche B) rend le sous-arbre gauche de B

;;; ab-droit : ArbreBinaire[alpha]/non vide/ -> ArbreBinaire[alpha]
;;; (ab-droit B) rend le sous-arbre droit de B
```

Les valeurs de type `ArbreBinaire[alpha]`, c'est-à-dire construites avec `ab-vide` et `ab-noeud`, sont des valeurs abstraites. Elles ne sont pas affichées telles qu'elles ont été construites, mais sont manifestées à la façon des procédures : `#<object>`. Pour permettre néanmoins la visualisation du contenu d'un arbre binaire, nous avons ajouté aux fonctions de la barrière d'abstraction une fonctions calculant une expression affichable à partir d'un arbre binaire. Voici sa spécification :

```
;;; ab-expression: ArbreBinaire[alpha] -> Sexpression
;;; (ab-expression B) rend une Sexpression reflétant la construction
;;; de l'arbre binaire B.
```

Remarque : Dans toute la suite de l'exercice, vous manipulerez les arbres binaires **uniquement** à travers les fonctions de la barrière d'abstraction.

Question 1 :

Écrire les fonctions `(ab-B1)` et `(ab-B2)`, qui construisent les arbres `B1` et `B2` de la figure précédente.

Question 2 :

Écrire une fonction `ab-feuille` qui prend en argument une étiquette et renvoie l'arbre binaire contenant cette seule étiquette. Exemple

`(ab-expression (ab-feuille 1))` → `(ab-noeud 1 (ab-vide) (ab-vide))`

Vous utiliserez cette fonction pour définir la fonction `(ab-B2-bis)` construisant l'arbre B2.

Question 3 :

Écrire le prédicat `ab-vide?`, qui reconnaît si un arbre binaire est vide.

```
;;; ab-vide? : ArbreBinaire[alpha] -> bool
;;; (ab-vide? B) rend vrai ssi B est un arbre vide et faux sinon
```

Question 4 :

Écrire le prédicat `ab-feuille?`, qui reconnaît si un arbre binaire est une feuille

```
;;; ab-feuille? : ArbreBinaire[alpha] -> bool
;;; (ab-feuille? B) rend vrai ssi B est une feuille et faux sinon
```

Question 5 :

On définit récursivement le nombre de Strahler d'un arbre binaire comme suit :

- si l'arbre est vide son nombre de Strahler vaut 0,
- sinon le calcul du nombre de Strahler dépend des valeurs des nombres de Strahler des sous-arbres gauche et droit :
  - si ses sous-arbres gauche et droit ont même nombre de Strahler S, alors le nombre de Strahler de l'arbre est $S+1$,
  - sinon le nombre de Strahler de l'arbre est égal au maximum des nombres de Strahler de ses sous-arbres gauche et droit.

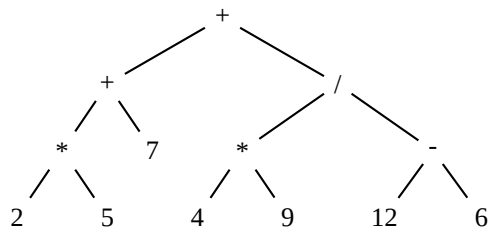
Écrire une fonction récursive `ab-strahler` qui, étant donné un arbre binaire, retourne son nombre de Strahler. Par exemple

```
(ab-strahler (ab-B1))  → 3
(ab-strahler (ab-B2))  → 2
```

À quoi correspond le nombre de Strahler d'un arbre binaire ? Dans le cas d'un arbre binaire représentant une expression arithmétique, c'est le nombre minimal de registres nécessaires pour évaluer cet arbre au moyen d'une calculatrice avec laquelle on peut faire trois types d'opérations :

- mettre une valeur dans un registre
- mettre le résultat de `reg1` *op* `reg2` dans `reg3` (un même registre pouvant apparaître plusieurs fois)
- faire afficher le contenu d'un registre.

L'exemple ci-dessous montre un arbre E dont le nombre de Strahler est 3, et une évaluation de E qui utilise trois registres :




```
reg1 <- 4
reg2 <- 9
reg1 <- reg1 * reg2
reg2 <- 12
reg3 <- 6
reg2 <- reg2 - reg3
reg1 <- reg1 / reg2
reg2 <- 2
reg3 <- 5
reg2 <- reg2 * reg3
reg3 <- 7
reg2 <- reg2 + reg3
reg1 <- reg1 + reg2
affiche reg1
```

Remarque : l'évaluation peut se faire de plusieurs manières, en utilisant plus ou moins de registres. Par exemple, si on commence l'évaluation de E par `reg1 <- 2` et `reg2 <- 5` alors il faudra utiliser au moins quatre registres.

Question 6 : Écrire la fonction `ab-nombre-noeuds` qui rend le nombre de nœuds d'un arbre binaire, par exemple :

```
(ab-nombre-noeuds (ab-B1)) → 9
(ab-nombre-noeuds (ab-B2)) → 7
```

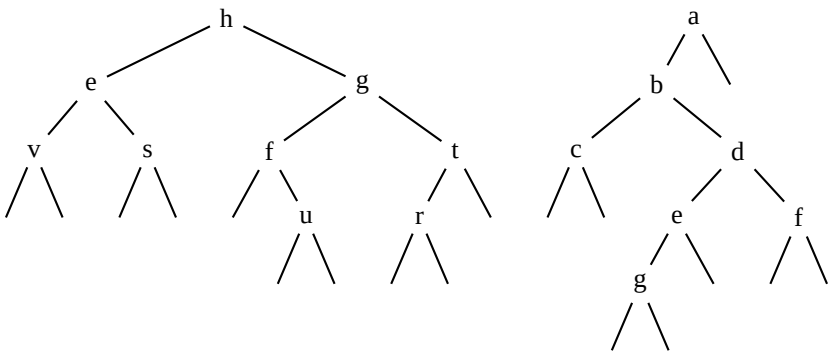
Question 7 : Écrire la fonction `ab-nombre-noeuds-nive au` qui, étant donnés un entier positif k et un arbre binaire B rend le nombre de nœuds étiquetés à niveau k dans B. La racine d'un arbre non vide est à niveau 1, et le niveau d'un nœud est égal au niveau de son père augmenté de 1. Par exemple

```
(ab-nombre-noeuds-nive au 3 (ab-B1)) → 4
(ab-nombre-noeuds-nive au 4 (ab-B2)) → 2
```

Exercice 2 – Listes des nœuds

Le but de cet exercice est de lister de différentes façons les nœuds d'un arbre binaire.

La figure suivante reproduit les deux arbres binaires de l'exercice « Arbres binaires » (page 2), qui sont construits par les fonctions `(ab-B1)` et `(ab-B2)` .



Remarque : Dans toute la suite de l'exercice, vous manipulerez les arbres binaires **uniquement** à travers les fonctions de la barrière d'abstraction.

Question 1 :

On définit récursivement la liste préfixe d'un arbre binaire comme suit :

– si `B` est l'arbre vide alors sa liste préfixe est vide,

– sinon la liste préfixe de `B` est égale à la concaténation de l'étiquette de la racine de `B` et des listes préfixes des sous-arbres gauche et droit de `B`.

Écrire une fonction `ab-prefixe` qui, étant donné un arbre binaire, retourne la liste de ses étiquettes en ordre préfixe. Par exemple

```
(ab-prefixe (ab-B1)) → (h e v s g f u t r)
(ab-prefixe (ab-B2)) → (a b c d e g f)
```

Question 2 :

On définit récursivement la liste suffixe d'un arbre binaire comme suit :

– si `B` est l'arbre vide alors sa liste suffixe est vide,

– sinon la liste suffixe de `B` est égale à la concaténation de la liste suffixe du sous-arbre gauche de `B`, de la liste suffixe du sous-arbre droit de `B`, et de l'étiquette de la racine de `B`.

Écrire une fonction `ab-suffixe` qui, étant donné un arbre binaire, retourne la liste de ses étiquettes en ordre suffixe . Par exemple

```
(ab-suffixe (ab-B1)) → (v s e u f r t g h)
(ab-suffixe (ab-B2)) → (c g e f d b a)
```

Question 3 : On définit récursivement la branche gauche d'un arbre binaire comme suit :

– si `B` est l'arbre vide alors sa branche gauche est la liste vide,

– sinon la branche gauche de `B` est égale à la concaténation de l'étiquette de la racine de `B` et de la branche gauche du sous-arbre gauche de `B`.

Écrire une fonction récursive `ab-branche-gauche` qui, étant donné un arbre binaire, retourne sa branche gauche. Par exemple

```
(ab-branche-gauche (ab-B1)) → (h e v)
(ab-branche-gauche (ab-B2)) → (a b c)
```

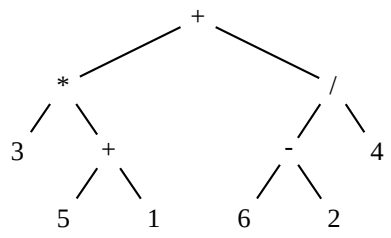
Question 4 : Écrire une fonction récursive `ab-branche-droite` qui, étant donné un arbre binaire, retourne sa branche droite. Par exemple

```
(ab-branche-droite (ab-B1)) → (h g t)
(ab-branche-droite (ab-B2)) → (a)
```

1.2 Arbres d'expression

Exercice 3 – Arbre binaire représentant une expression arithmétique

Le but de cet exercice est de représenter une expression arithmétique binaire par un arbre binaire. Par exemple l'expression `(+ (* 3 (+ 5 1)) (/ (- 6 2) 4))` sera représentée par l'arbre



Quel arbre représente l'expression `(+ 3 5)` ?

Remarque : Dans toute la suite de l'exercice, vous manipulerez les arbres binaires **uniquement** à travers les fonctions de la barrière d'abstraction.

Il sera aussi utile de faire appel aux fonctions `ab-feuille` et `ab-feuille?` , définies dans l'exercice « Arbres Binaires » (page 2).

Question 1 : Écrire la fonction, nommée `expr->arbre`, calculant l'arbre associé à une expression. On supposera ici que l'expression de départ est une expression arithmétique bien formée, composée uniquement d'opérateurs binaires et de constantes numériques.

On pourra verifier le résultat de la fonction en utilisant la fonction `ab-expression` de la barrière d'abstraction des arbres binaires. Par exemple

```
(ab-expression (expr->arbre '(+ (* 3 (+ 5 1))(/(- 6 2) 4)))) →
(ab-noeud
  '+
  (ab-noeud
    '*
    (ab-noeud 3 (ab-vide) (ab-vide))
    (ab-noeud
      '+
      (ab-noeud 5 (ab-vide) (ab-vide))
      (ab-noeud 1 (ab-vide) (ab-vide))))
  (ab-noeud
    '/
    (ab-noeud
      '-
      (ab-noeud 6 (ab-vide) (ab-vide))
      (ab-noeud 2 (ab-vide) (ab-vide)))
    (ab-noeud 4 (ab-vide) (ab-vide))))
```

Question 2 : La fonction `expr->arbre` a pour hypothèse que l'expression est bien formée. Son comportement dans le cas d'une expression mal formée n'est donc pas défini.

On demande ici d'écrire une fonction, nommée `expr->arbreV`, qui calcule l'arbre associé à une expression, et signale une erreur lorsque l'expression n'est pas purement binaire (mais ne vérifie pas la nature des opérateurs et opérandes). Ainsi les deux exemples ci-dessus devront être traités en erreur par la fonction `expr->arbreV` :

```
(expr->arbreV '(* (+ 2) 3)) →
expr->arbrev: ERREUR: mauvais nombre d'opérandes dans (+ 2)

(expr->arbreV '(* (+ 2 5) 3 4 5)) →
expr->arbrev: ERREUR: mauvais nombre d'opérandes dans (* (+ 2 5) 3 4 5)
```

Exercice 4 – Évaluation d'une expression arithmétique

Remarque : avant de traiter cet exercice, nous vous conseillons de faire les exercices d'auto-évaluation de la barrière d'abstraction des arbres binaires.

On se propose ici d'évaluer les expressions arithmétiques représentées par des arbres binaires (cf. exercice « Arbre binaire représentant une expression arithmétique » (page 5)). On évaluera ici uniquement des arbres binaires représentant des expressions arithmétiques bien formées, à partir des quatre opérateurs `+`, `-`, `*` , `/` et de constantes numériques.

Remarque : Dans toute la suite de l'exercice, vous manipulerez les arbres binaires **uniquement** à travers les fonctions de la barrière d'abstraction.

Il sera aussi utile de faire appel aux fonctions `ab-feuille` et `ab-feuille?`, définies dans l'exercice « Arbres Binaires » (page 2).

Dans cet exercice, `ArbreExprConst` est le type des arbres binaires non vides dont les feuilles sont étiquetées par des nombres et les autres nœuds sont étiquetés par les opérateurs (symboles) `+`, `*`, `/` et `-`.

Question 1 : Écrire une fonction `evaluation` de spécification

```
;;; evaluation : ArbreExprConst -> Nombre
;;; ERREUR lorsque l'arbre B contient des opérateurs autres que +, -, * et /
;;; ou lorsque les feuille ne contiennent pas des nombres
;;; (evaluation B) rend la valeur de l'expression arithmétique représentée par B
```

Par exemple
`(evaluation (expr->arbre '(* (+ 2 5) 3))) -> 21`
Remarquer que cette évaluation est équivalente à celle réalisée par l'interprète Scheme sur l'expression `(* (+ 2 5) 3)`.
Il est conseillé, pour une première approche, de traiter les opérateurs cas par cas.

Question 2 : Pour aller plus loin

On voudrait maintenant factoriser le traitement des opérateurs et non plus les traiter cas par cas. On pense à écrire le programme suivant qui est incorrect.

```
(define (evaluationFAUX B) ;; PROGRAMME INCORRECT
  (if (ab-feuille? B)
      (if (number? (ab-etiquette B))
          (ab-etiquette B)
          (erreur 'evaluation "les feuilles doivent contenir des nombres"))
      ((ab-etiquette B)
       (evaluationFAUX (ab-gauche B))
       (evaluationFAUX (ab-droit B)))))
```

Par exemple pour `(evaluationFAUX (expr->arbre '(* (+ 2 5) 3)))`, l'interprète Scheme signale l'erreur suivante
`. procedure application: expected procedure, given: +; arguments were: 2 5`
Pouvez-vous corriger ce programme ? Ecrivez alors la fonction `evaluationJUSTE`
Pour vous aider, vous pouvez méditer sur les exemples suivants, qui font appel à la fonction `eval` de Scheme.

```
(symbol?(ab-etiquette(expr->arbre '(* (+ 2 5) 3)))) → #t
(procedure?(ab-etiquette(expr->arbre '(* (+ 2 5) 3)))) → #f
(symbol?(eval(ab-etiquette(expr->arbre '(* (+ 2 5) 3))))) → #f
(procedure?(eval(ab-etiquette(expr->arbre '(* (+ 2 5) 3))))) → #t
```

1.3 Dérivation formelle

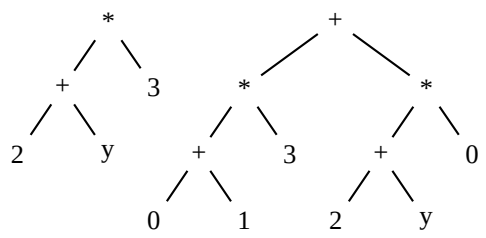
Exercice 5 – Dérivation formelle

On considère ici des arbres binaires représentant des expressions arithmétiques composées d'opérateurs d'addition et de multiplication, ainsi que de constantes et de variables. On pourra construire automatiquement de tels arbres en utilisant la fonction de l'exercice « Arbre représentant une expression arithmétique » (page 5)

Le but de cet exercice est de calculer l'arbre représentant l'expression dérivée, en appliquant les règles de dérivation classiques :

- la dérivée d'une somme est égale à la somme des dérivées : $(e_1 + e_2)' = e_1' + e_2'$,
- la dérivée d'un produit est égale à la somme des produits "mixtes" : $(e_1 * e_2)' = e_1' * e_2 + e_1 * e_2'$,
- la dérivée d'une variable vaut 1 et la dérivée d'une constante vaut 0.

On ne fera aucune simplification sur l'expression dérivée, par exemple la dérivée, par rapport à y, de $(2+y)*3)$ est $((0+1)*3)+((2+y)*0)$. La figure suivante montre l'arbre de l'expression initiale (à gauche), et l'arbre de l'expression dérivée (à droite).



Remarque : Dans la suite de l'exercice, vous manipulerez les arbres binaires **uniquement** à travers les fonctions de la barrière d'abstraction.

Il sera aussi utile de faire appel aux fonctions `ab-feuille` et `ab-feuille?`, définies dans l'exercice « Arbres Binaires » (page 2).

Dans cet exercice, les arbres binaires considérés représentent des expressions aritmétiques composées d'opérateurs d'addition et de multiplication, ainsi que de de constantes et de variables. Ainsi dans ce qui suit, le type `ArbreExpr` est le type des arbres binaires non vides dont les feuilles sont étiquetées par des nombres ou des symboles représentant des variables et les autres nœuds sont étiquetés par les opérateurs (symboles) `+` et `*`.

Question 1 : Écrire une définition de la fonction `derivation` qui, étant donné un arbre binaire *B* et une variable *x*, renvoie l'arbre représentant la dérivée par rapport à *x* de l'arbre initial. La fonction renverra une erreur lorsque l'arbre initial contient des opérateurs autres que + et *.

```
;;; derivation : ArbreExpr * Symbol -> ArbreExpr
;;; ERREUR lorsque l'arbre B contient des opérateurs autres que + et *
;;; (derivation B var) rend l'arbre résultant de l'application des règles
;;; de dérivation de la somme et du produit, selon la variable var
```

Exercice 6 – Dérivation formelle paramétrée

L'exercice « Dérivation formelle » (page 7) implante la dérivation formelle pour des arbres d'expressions ne contenant que des opérateurs d'addition et de multiplication. Si l'on veut pouvoir traiter d'autres opérateurs, il faut non seulement connaître leurs règles de dérivation, mais en plus modifier le code de la fonction `dérivation` en rajoutant une règle de filtrage pour chaque nouvel opérateur :

```
(define (derivation B var)
  (if (ab-feuille? B)
      ...
      (cond
        ((equal? '+ op) ... )
        ((equal? '* op) ... )
        ((equal? '/ op) ... )
        ((equal? '- op) ... )
        ...
        (else (erreur ...)))))
```

On se propose ici de fabriquer un mécanisme plus général de récriture qui produit l'arbre dérivé en fonction d'un ensemble de règles de dérivation, *passées elles aussi en paramètre*.

A - Les règles

Une règle de dérivation se présentera sous la forme `(MG -> MD)`, ce qui se lit "le membre gauche `MG` se récrit en le membre droit `MD`".

Le membre gauche `MG` est une expression `(op a b)`, formée d'un opérateur binaire `op` et de deux opérandes `a` et `b`. Le membre droit `MD` est une expression contenant des opérateurs binaires et des dérivations, notées par l'opérateur `D`. Voici par exemple quelques règles classiques :

```
((+ a b) -> (+ (D a) (D b)))
((* a b) -> (+ (* (D a) b) (* a (D b))))
((/ a b) -> (/(- (* (D a) b) (* a (D b))) (* b b)))
```

Dans cet exercice, `ArbreExpr` est le type des arbres binaires non vides dont les feuilles sont étiquetées par des nombres ou des symboles représentant des variables et les autres nœuds sont étiquetés par des opérateurs.

Remarque : Dans toute la suite de l'exercice, on supposera que les règles de dérivation sont bien formées, et que les membres droits sont des listes représentants des expressions formées à l'aide d'opérateurs binaires et du seul opérateur unaire D, représentant l'opérateur de dérivation.

Question 1 : On commence par définir quelques fonctions utilitaires pour manipuler les règles :

– les fonctions, nommées `membre-gauche` et `membre-droit`, qui renvoient respectivement le membre gauche et le membre droit d'une règle (on supposera que la règle est conforme à la syntaxe proposée ci-dessus).
– les fonctions, nommées `var1` et `var2`, qui renvoient respectivement le premier et le second opérande d'un membre gauche de règle (que l'on supposera encore une fois conforme à la syntaxe).

Par exemple

```
(var1  '(/ a b))   → a
(var2  '(/ a b))   → b

(membre-gauche    '(( * a b) -> (+ (* (D a) b) (* a (D b)))))   → (* a b)
(membre-droit    '(( * a b) -> (+ (* (D a) b) (* a (D b)))))
→ (+ (* (D a) b) (* a (D b)))
```

Question 2 : La fonction de dérivation travaillera sur une liste de règles, parmi lesquelles il lui faudra trouver la bonne règle à appliquer, selon l'opérateur à traiter.

On demande ici de définir une fonction `rechRegle`, qui prend en paramètres un opérateur `op` et une liste de règles `LA`, et renvoie la règle de dérivation associée l'opérateur `op` dans la liste `LA`. Cette fonction renverra une erreur si la liste `LA` ne contient pas de règle pour l'opérateur `op`. Par exemple

```
(rechRegle   '/
             '(((+  a b) ->  (+ (D a) (D b)))
               (( * a b) -> (+ (* (D a) b) (* a (D b))))
               ((/ a b) -> (/(-  (* (D a) b) (* a (D b)))  (* b b)))
               ((exp  a b) -> (* b (* (D a) (exp  a (- b 1)))))))
→ ((/ a b) -> (/(-  (* (D a) b) (* a (D b)))  (* b b)))
```

Question 3 : Pour nommer les variables apparaissant dans les règles, il faut assurer une cohérence entre les noms choisis dans le le membre gauche et le membre droit, mais les noms eux-mêmes peuvent être quelconques. On peut par exemple préférer donner la règle de dérivation du produit sous la forme

```
(( * u v) -> (+ (* (D u) b) (* u (D v))))
```

On définira ici une fonction générale de substitution de symboles, nommée `substitution`, qui étant donnés deux symboles `s1` et `s2` et une expression `E`, renvoie l'expression dans laquelle toutes les occurrences de `s1` sont remplacées par `s2`. Par exemple

```
(substitution    'a 'u '(a  b r a c a d a b r a))  →
    (u  b r u c u d u b r u)
(substitution    'a 'u  '((a  b r a)  c a (d  a (b  r)a)))  →
((u  b r u)  c u (d  u (b  r)u))
```

Et aussi, bien sûr :

```
(substitution    'b 'v
                 (substitution    'a 'u
                                  '(( * a b) -> (+ (* (D a) b) (* a (D b))))))
→ (( * u v) -> (+ (* (D u) v) (* u (D v))))))
```

B - La fonction de dérivation

Il s'agit donc de définir la fonction de dérivation qui produit l'arbre de l'expression dérivée, lorsqu'on fournit un arbre d'expression et l'ensemble des règles de dérivation nécessaires.

On note `Regle` le type des règles de dérivation bien formées (elles sont de la forme `((op a b) -> (...))`).

La fonction de dérivation a pour spécification

```
;;; derivation : ArbreExpr * Symbol * LISTE[Regle] -> ArbreExpr
;;; ERREUR lorsque l'arbre B contient un opérateur dont la règle de dérivation n'est pas
;;; dans la liste LA
;;; (derivation B var LA) rend l'arbre résultant de la dérivation de l'arbre B
;;; par rapport à la variable var, selon les règles de la liste LA.
```

Question 4 : Il s'agit de définir la fonction `derivation`. Dans la suite on vous guide pour écrire une définition (mais vous pouvez aussi ne pas tenir compte de ces conseils ;-).

Voici le programme de départ, suivi de commentaires sur sa structure :

```
(define (derivation B var LA)
  (if (ab-feuille? B)
      (ab-feuille (if (not (equal? var (ab-etiquette B))) 0 1))
      (let * (( R (rechRegle (ab-etiquette B) LA))
              ;si l'on ne trouve pas de regle, on sort en erreur par rechRegle
              (MG (membre-gauche R))
              (MD (membre-droit R)))
        (application (ab-gauche B)
                     (ab-droit B)
                     (substitution (var1 MG) 'u
                                   (substitution (var2 MG) 'v MD))))))
```

Le cas où l'arbre est réduit à une feuille se traite simplement – les fonctions `ab-feuille` et `ab-feuille?`, pour construire et reconnaître un arbre qui est une feuille sont définies à l'exercice « Arbres Binaires » (page 2).

Sinon il faut extraire de la liste la règle de dérivation correspondant à l'opérateur situé à la racine de l'arbre `B` (si l'on ne trouve pas de règle, on sort en erreur par la fonction `rechRegle`). Lorsque l'on trouve une telle règle `((op a b) -> (...))`, il reste à l'appliquer en prenant pour premier (resp. second) opérande le sous-arbre gauche (resp. droit) de `B`. Et l'application d'une règle peut entraîner le calcul d'une dérivation . . ., d'où une récursion croisée entre la fonction `derivation` et sa fonction interne `application`.

Il reste donc à écrire la définition de la fonction interne `application`, dont la spécification est :

```
;; application : ArbreExpr * ArbreExpr * Expression -> ArbreExpr
;; HYPOTHESE : expr est une expression formée à l'aide d'opérateurs binaires
;; et du seul opérateur unaire D, représentant l'opérateur de dérivation
;; dans les membres droits des règles.
;; (application ABG ABD expr) rend l'arbre correspondant au calcul de
;; l'expression expr en prenant ABG(resp. ABD) comme opérande gauche(resp. droit)
```

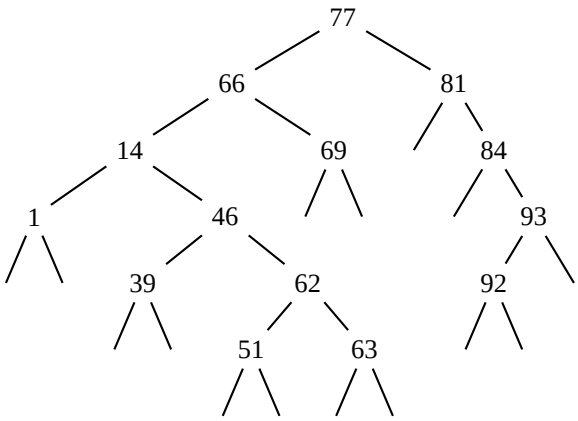
1.4 Arbres binaires de recherche

Exercice 7 – Arbres binaires de recherche

Un arbre binaire de recherche numérique est un arbre binaire, étiqueté par des nombres, tel que tout arbre binaire de recherche non vide possède la propriété suivante :

- l'étiquette de sa racine est
  - supérieure à toutes les étiquettes de son sous-arbre gauche,
  - inférieure à toutes les étiquettes de son sous-arbre droit,
- ses sous-arbres gauche et droit sont aussi des arbres binaires de recherche.

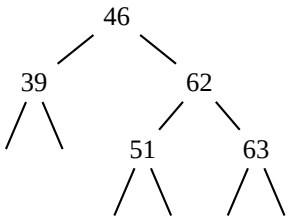
Cette propriété permet «d'aiguiller» la recherche d'un élément, d'où le nom «d'arbre binaire de *recherche*». La figure suivante montre un exemple d'arbres binaires de recherche, que l'on nommera par la suite `mon-ABR` :



Le but de cet exercice est de rechercher et d'ajouter des éléments (qui sont des nombres) dans un arbre binaire de recherche. On supposera par la suite que les étiquettes d'un arbre binaire de recherche sont toujours **deux à deux distinctes**.

Le type `ArbreBinRecherche` est représenté par le type `ArbreBinaire[Nombre]`, et l'on écrira toutes les fonctions demandées sur les arbre binaire de recherche en utilisant la barrière d'abstraction des arbres binaires « Arbres binaires » (page 2).

Question 1 : Écrire le semi-prédicat `abr-recherche` qui, étant donnés un nombre `x` et un arbre binaire de recherche `ABR` renvoie `#f` si `x` n'apparaît pas dans `ABR`, et sinon renvoie le sous-arbre de racine `x` dans `ABR`. Par exemple la recherche de l'élément `3` dans `mon-ABR` renvoie `#f`, et la recherche de l'élément `46` dans `mon-ABR` renvoie l'arbre

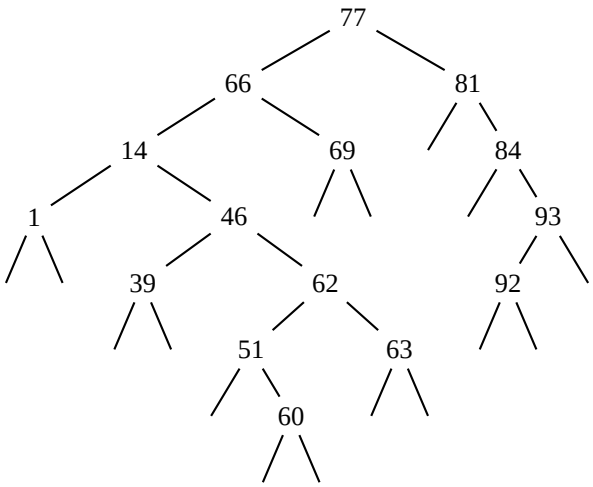


Pour rechercher un élément `x` dans un arbre binaire de recherche `ABR` non vide, on compare cet élément à la racine de `ABR` :

- si `x` est plus petit que la racine on recommence récursivement la recherche sur le sous-arbre gauche de `ABR`
- si `x` est plus grand on recommence récursivement la recherche sur le sous-arbre droit de `ABR`
- si `x` est égal à la racine, on renvoie l'arbre `ABR`

Et la recherche d'un élément dans un arbre binaire de recherche vide renvoie la valeur `#f`.

Question 2 : Pour ajouter un élément "au niveau des feuilles" d'un arbre binaire de recherche, on procède récursivement comme pour une recherche. Et l'ajout de `x` dans l'arbre vide rend l'arbre de racine `x`, et dont les sous-arbres gauche et droit sont vides. Par exemple l'ajout aux feuilles de l'élément `60` dans `mon-ABR` renvoie l'arbre suivant que l'on nommera `mon-ABR-bis` :



Écrire la fonction `abr-ajout-feuille` qui, étant donnés un nombre `x` et un arbre binaire de recherche `ABR` renvoie

- `ABR` si `x` apparaît initialement dans l'arbre
- sinon l'arbre binaire de recherche dans lequel `x` a été ajouté au niveau des feuilles de `ABR` .

Question 3 : Écrire la fonction `abr-aleatoire` qui, étant donné un entier `n`, renvoie un arbre binaire de recherche résultant de l'adjonction successive de `n` nombres aléatoires (l'arbre n'est pas nécessairement de taille `n` car un même nombre peut être tiré plusieurs fois). On pourra utiliser la fonction `(random k)`, qui retourne un entier aléatoire entre 0 et $k-1$.

Écrire aussi la fonction `abr-construction` qui, étant donné une liste de nombres `L` renvoie un arbre binaire de recherche résultant de l'adjonction successive des éléments de `L`. Les éléments de `L` seront ajoutés en ordre inverse de leur ordre d'apparition dans la liste. Par exemple l'arbre de la question 1 peut être obtenu par

```
(abr-construction '(63 39 51 62 46)) ,
```

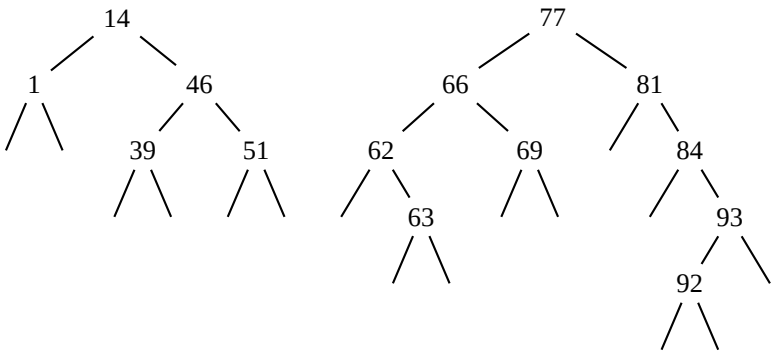
mais aussi par

```
(abr-construction '(51 39 63 62 46)) .
```

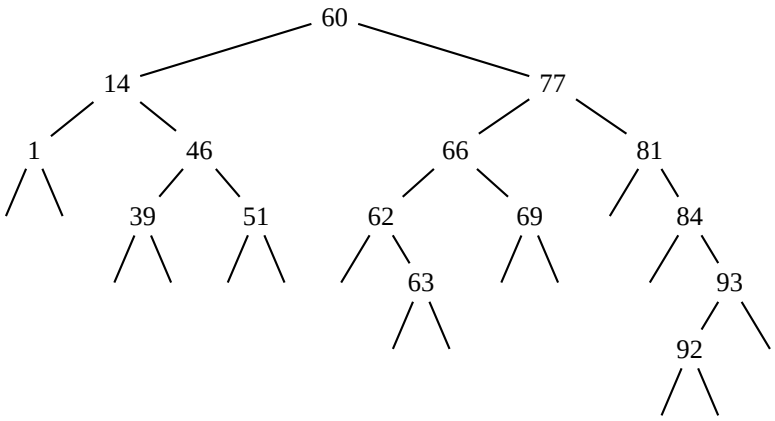
Question 4 : Écrire la fonction `abr-coupure` qui, étant donnés un nombre `x` et un arbre binaire de recherche `ABR` , renvoie une liste de deux arbres binaires de recherche :

- le premier est composé des éléments de `ABR` strictement inférieurs à x,
- et le second est composé des éléments de `ABR` strictement supérieurs à x.

Par exemple la coupure de `mon-ABR` selon l'élément `60` renvoie le couple d'arbres binaires de recherche :



Question 5 : En déduire une définition de la fonction `abr-ajout-racine` qui, étant donnés un nombre `x` et un arbre binaire de recherche `ABR` renvoie l'arbre dans lequel `x` a été ajouté à la racine de `ABR`. Attention, lorsque `x` apparaît initialement dans `ABR`, le résultat de l'ajout de `x` est en général un arbre différent de `ABR`. Par exemple l'ajout à la racine de l'élément `60` dans `mon-ABR-bis` renvoie l'arbre :



Exercice 8 – Arbre de recherche pour un point dans un intervalle

Comme dans l'exercice « Y-a-t-il un point dans l'intervalle ? » (dans le polycopié d'exercices de la saison1), la donnée de cet exercice est un ensemble de points et un intervalle de l'axe réel, et le but de l'exercice est de rechercher un point qui appartienne à l'intervalle.

La différence est que l'on va représenter ici l'ensemble des points non par une liste, mais par un arbre binaire de recherche, ce qui rend la recherche beaucoup plus rapide.

Pour la définition des arbres binaires de recherche se reporter à l'exercice « Arbres binaires de recherche » (page 10). Les arbres binaires de recherche sont manipulés à travers les fonctions de la barrière d'abstraction des arbres binaires.

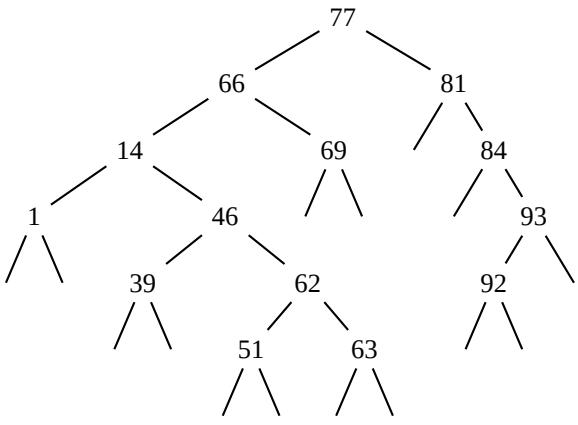
Question 1 : On considère les types

- `Intervalle = NUPLET[Nombre Nombre]`, et
- `ArbreBinRecherche = ArbreBinaire[Nombre]`, avec la condition de croissance pour la liste des étiquettes en ordre infixe.

Ecrire la fonction de recherche répondant à la spécification suivante :

```
;;; rechercheAbr : Intervalle * ArbreBinRecherche -> Nombre + #f
;;; (rechercheAbr interv abr) renvoie un élément de l'arbre abr (le plus près de la
;;; racine) qui est dans l'intervalle interv
;;; et renvoie #f si abr ne contient pas d'élément dans l'intervalle interv
```

Par exemple la recherche d'un point de l'intervalle `(20 60)` dans l'arbre binaire de recherche ci-dessous renvoie le point `46`.



Question 2 : Cette question se propose de faire réfléchir sur la notion de complexité des algorithmes. L'exécution d'un programme sur une donnée consomme des ressources : temps et mémoire. Dans le cas des fonctions récursives, ces ressources sont proportionnelles au nombre d'appels récursifs. Le langage Scheme propose une fonction `trace` qui permet de visualiser les appels récursifs lors de l'exécution d'une fonction sur une donnée. On se propose de mettre en évidence la complexité d'une fonction en "traçant" son exécution.

Il s'agit de comparer l'efficacité de différentes méthodes pour traiter un même problème. Ici le problème à résoudre est de rechercher, parmi un ensemble de points, un point qui appartienne à un certain intervalle. On a étudié deux représentations différentes pour l'ensemble des points : une liste dans l'exercice « Y-a-t-il un point dans l'intervalle ? » (dans le polycopié d'exercices de la saison1), et un arbre binaire de recherche dans le présent exercice. Chacune de ces représentations induit un algorithme de recherche différent :

- recherche séquentielle dans le cas des listes,
- recherche dichotomique (du grec "couper en deux") dans le cas des arbres.

Pour une recherche séquentielle sur une liste de taille n, c'est-à-dire contenant n points, le temps de recherche est proportionnel à n dans le pire des cas : si le point recherché se trouve en fin de liste, la fonction se rappelle récursivement sur *tous les points de la liste* avant de produire son résultat. (En moyenne le temps de recherche est de l'ordre de $n/2$.)

En revanche dans le cas d'un arbre binaire de recherche, le principe d'aiguillage fait que chaque examen d'un point (pour savoir s'il appartient à l'intervalle) permet de "laisser tomber" un sous-arbre entier (gauche ou droit). Si chaque sous-arbre contient à peu près la moitié de son arbre (on dit alors que l'arbre est *équilibré*, le nombre de points restant à examiner est *divisé par deux* à chaque fois. Ainsi, si l'ensemble de départ contient n points, le nombre d'appels récursifs de la fonction de recherche dichotomique sera au pire $\log_2 n$, et le temps de recherche sera donc proportionnel à $\log_2 n$. (Rappel : si $n = 2^p$, alors $p = \log_2 n$ est le nombre de fois qu'il faut répéter la division de n par 2 pour arriver à 1.)

Ainsi s'il y a de l'ordre d'un million de points ($10^6 \equiv 2^{20}$), le nombre d'appels récursifs lors de l'exécution d'une recherche sera au pire de

- un million pour la fonction de recherche séquentielle `rechercheListe` ,
- une vingtaine pour la fonction de recherche dichotomique `rechercheAbr` , si l'arbre est équilibré.

Expérimentation

Il s'agit de tracer l'exécution des fonctions `rechercheListe` et `rechercheAbr` sur différentes données. Pour utiliser la fonction `trace` , il faut la charger et indiquer quelles sont les fonctions que l'on veut tracer :

- invoquer d'abord `(require-library "trace.ss")`
- puis `(trace rechercheListe)` et `(trace rechercheAbr)` .

Attention `trace` ne visualise les appels récursifs que des fonctions top-level (pas des fonctions internes).

Voici deux exemples à comparer :

```
(rechercheListe '(14 5 7 20 13 34 30 16 2 1 15 3 18 40 6) '( 8 12))
(rechercheAbr '( 8 12)
              (abr-construction
                 (reverse '(14 5 7 20 13 34 30 16 2 1 15 3 18 40 6))))
```

Pour la fonction `abr-construction` , se reporter à l'exercice « Arbres binaires de recherche » (page 10).

On peut aussi faire différents tests sur des listes aléatoires de points, et leur arbre binaire associé. Pour engendrer une liste de nombres aléatoires, on pourra utiliser la fonction `(random k)`, qui retourne un entier aléatoire entre 0 et $k-1$.

2 Autres arbres

2.1 Arbres Généraux et Forêts

Exercice 9 – Arbres généraux

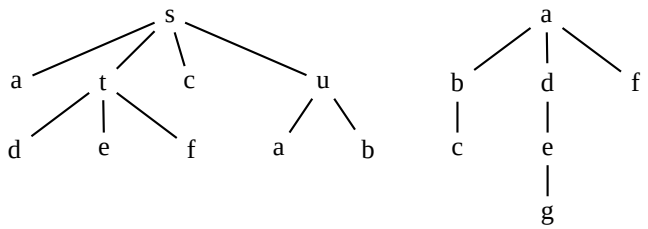
Les arbres généraux sont des arbres dans lesquels le nombre de fils de chaque nœud n'est pas borné, et les fils d'un nœud forment une suite – ordonnée– d'arbres. Ces arbres portent des étiquettes sur les nœuds.

On peut donner la définition récursive suivante : un *arbre général* est formé

– d'un nœud portant une étiquette

– d'une liste d'arbres généraux (appelée *forêt*), éventuellement vide.

La figure suivante montre deux arbres généraux, que l'on nommera par la suite A1 (celui de gauche) et A2 :



Le but de cet exercice est de manipuler les arbres généraux à travers leur barrière d'abstraction.

Le type `ArbreGeneral[alpha]` représente les arbres généraux dont les étiquettes sont de type `alpha`. Le type `Foret[alpha]` représente les suites d'arbres de type `ArbreGeneral[alpha]` ; ainsi `Foret[alpha] = LISTE[ArbreGeneral[alpha]]`.

La barrière d'abstraction permettant de manipuler des arbres généraux est formée

– du constructeur `(ag-noeud e F)` pour construire un arbre à partir d'une étiquette et d'une forêt,

– des accesseurs `(ag-etiquette A)` et `(ag-foret A)` pour accéder d'une part à l'étiquette de la racine, et d'autre part à la forêt d'un arbre.

Voici les spécifications des fonctions de de la barrière d'abstraction

```
;;; ag-noeud : alpha * Foret[alpha] -> ArbreGeneral[alpha]
;;; (ag-noeud e F) rend l'arbre dont la racine a pour étiquette e et de forêt F

;;; ag-etiquette : ArbreGeneral[alpha] -> alpha
;;; (ag-etiquette A) rend l'étiquette de la racine de l'arbre A

;;; ag-foret : ArbreGeneral[alpha] -> Foret[alpha]
;;; (ag-foret A) rend la forêt sous la racine de A.
```

Remarque : Dans toute la suite de l'exercice, vous manipulerez les arbres **uniquement** à travers les fonctions de la barrière d'abstraction. Les forêts étant des listes d'arbres, on utilisera les fonctions sur les listes (`car`, `cdr`, `pair?`, `map` ...) pour manipuler les forêts.

Question 1 :

Écrire les fonctions `(ag-A1)` et `(ag-A2)`, qui construisent les arbres `A1` et `A2` de la figure précédente.

Question 2 : Un arbre réduit à un nœud est appelé *feuille*. Écrire le prédicat `ag-feuille?` qui reconnaît si un arbre est une feuille, et la fonction `ag-feuille` qui construit une feuille à partir de la donnée d'une étiquette.

Question 3 : Écrire la fonction `ag-nombre-noeuds` qui rend le nombre de nœuds d'un arbre général. Par exemple

```
(ag-nombre-noeuds (ag-A1))  → 10
(ag-nombre-noeuds (ag-A2))  → 7
```

Remarque : du fait des définitions mutuellement récursives des arbres et des forêts, la définition d'une fonction sur les arbres s'accompagne de la définition d'une fonction sur les forêts, ces deux fonctions s'appelant récursivement. Par exemple pour calculer le nombre de nœuds d'un arbre il faut calculer le

nombre de nœuds de sa forêt sous-jacente, et pour calculer le nombre de nœuds d'une forêt il faut calculer le nombre de nœuds de chacun de ses arbres.

Question 4 : Écrire la fonction `ag-nombre-noeuds-niveau` qui, étant donnés un entier k et un arbre général A, rend le nombre de nœuds à niveau k dans A. La racine est à niveau 1, et un nœud dont le père à niveau k, est à niveau $k+1$. Par exemple

```
(ag-nombre-noeuds-niveau 3 (ag-A1))  → 5
(ag-nombre-noeuds-niveau 3 (ag-A2))  → 2
```

Question 5 : Écrire la fonction `ag-nombre-feuilles-niveau` qui, étant donnés un entier k et un arbre général A, rend le nombre de *feuilles* à niveau k dans A. Par exemple

```
(ag-nombre-feuilles-niveau 3 (ag-A1))  → 5
(ag-nombre-feuilles-niveau 3 (ag-A2))  → 1
```

Exercice 10 – Profondeurs des arbres généraux

Le but de cet exercice est de calculer la profondeur des arbres généraux, en manipulant les arbres à travers leur barrière d'abstraction.

On reprend les arbres servant d'exemples dans l'exercice « Arbres généraux » (page 14), qui sont construits par `ag-A1` et `ag-A2` .

Question 1 :

On définit récursivement la profondeur d'un arbre comme suit :

– si `A` est une feuille alors sa profondeur est 1,

– sinon la profondeur de `A` est égale au maximum des profondeurs de ses sous-arbres, augmenté de 1.

Écrire une fonction `ag-profondeur` qui, étant donné un arbre général, retourne sa profondeur. Par exemple

```
(ag-profondeur (ag-A1))  → 3
(ag-profondeur (ag-A2))  → 4
```

Dans cette question vous ne devez pas utiliser les fonctionnelles `map` et `reduce` . Attendez pour cela la question suivante !

Remarque : du fait des définitions mutuellement récursives des arbres et des forêts, la définition d'une fonction sur les arbres s'accompagne de la définition d'une fonction sur les forêts, ces deux fonctions s'appelant récursivement. Par exemple pour calculer la profondeur d'un arbre il faut calculer la profondeur de sa forêt, et pour calculer la profondeur d'une forêt il faut calculer la profondeur de chacun de ses arbres.

Question 2 :

Écrire une fonction `ag-profondeur2` qui calcule la profondeur d'un arbre en utilisant les fonctionnelles `map` et `reduce` .

Indication : ne pas oublier qu'une forêt est une liste d'arbres.

Exercice 11 – Strahler d'un arbre général

On définit récursivement le nombre de Strahler d'un arbre général comme suit :

– si l'arbre est réduit à une feuille son nombre de Strahler vaut 0,

– sinon le calcul du nombre de Strahler dépend des valeurs des nombres de Strahler de tous les arbres de la forêt sous-jacente :

  – si tous les arbres de la forêt sous-jacente ont même nombre de Strahler S, alors le nombre de Strahler de l'arbre général est $S+1$,

  – sinon le nombre de Strahler de l'arbre général est égal au maximum des nombres de Strahler des arbres de la forêt sous-jacente.

Le but de cet exercice est de calculer le le nombre de Strahler d'un arbre général. En reprenant les arbres servant d'exemples dans l'exercice « Arbres généraux » (page 14), qui sont construits par `ag-A1` et `ag-A2` :

```
(ag-strahler (ag-A1)) → 1
(ag-strahler (ag-A2)) → 2
```

Question 1 : Écrire un prédicat `tous-egaux?` qui, étant donnée une liste non vide d'éléments, renvoie vrai ssi tous les éléments sont égaux.

```
(tous-egaux? '(a a a a)) → #T
(tous-egaux? '(20 21 20 20 20 20)) → #F
```

Question 2 : Écrire une fonction `le-max` qui, étant donnée une liste non vide de nombres, renvoie le maximum.

Si les nombres sont positifs, définir aussi cette fonction par un simple appel à la fonction `reduce` .

Question 3 : En utilisant les fonctions précédentes, écrire une fonction `ag-strahler` qui, étant donné un arbre général, retourne son nombre de Strahler.

Exercice 12 – Listes des nœuds des arbres généraux

Le but de cet exercice est de lister les étiquettes d'un arbre général, selon différents critères : liste des feuilles, liste préfixe et suffixe, liste des étiquettes de la branche gauche et de la branche droite.

On reprend les arbres servant d'exemples dans l'exercice « Arbres généraux » (page 14), qui sont construits par `ag-A1` et `ag-A2`

Question 1 : Écrire une fonction `ag-liste-feuilles` qui, étant donné un arbre général, retourne la liste -de gauche à droite- de ses feuilles (sans utiliser `map` et `reduce`). Par exemple

```
(ag-liste-feuilles (ag-A1)) → (a d e f c a b)
(ag-liste-feuilles (ag-A2)) → (c g f)
```

Question 2 : Écrire une fonction `ag-liste-feuilles2` qui calcule la liste des feuilles d'un arbre en utilisant les fonctionnelles `map` et `reduce` .

Question 3 :

On définit récursivement la liste préfixe d'un arbre comme suit :

- si `A` est une feuille alors sa liste préfixe contient un unique élément : l'étiquette,
- sinon la liste préfixe de `A` est égale à la concaténation de l'étiquette de la racine de `A` et des listes préfixes des sous-arbres de `A`.

Écrire une fonction `ag-prefixe` qui, étant donné un arbre général, retourne la liste de ses étiquettes en ordre préfixe (sans utiliser `map` et `reduce`). Par exemple

```
(ag-prefixe (ag-A1)) → (s a t d e f c u a b)
(ag-prefixe (ag-A2)) → (a b c d e g f)
```

Question 4 :

Écrire une fonction `ag-prefixe2` qui calcule la liste préfixe d'un arbre en utilisant les fonctionnelles `map` et `reduce` .

Question 5 :

On définit récursivement la liste suffixe d'un arbre comme suit :

- si `A` est une feuille alors sa liste suffixe contient un unique élément : son étiquette,
- sinon la liste suffixe de `A` est égale à la concaténation des listes suffixes des sous-arbres de `A`, et enfin de l'étiquette de la racine de `A` .

Écrire une fonction `ag-suffixe` qui, étant donné un arbre général, retourne la liste de ses étiquettes en ordre suffixe (sans utiliser `map` et `reduce`). Par exemple

```
(ag-suffixe (ag-A1)) → (a d e f t c a b u s)
(ag-suffixe (ag-A2)) → (c b g e d f a)
```

Question 6 : Écrire une fonction `ag-suffixe2` qui calcule la liste suffixe d'un arbre en utilisant les fonctionnelles `map` et `reduce` .

Question 7 : On définit récursivement la branche gauche d'un arbre comme suit :

- si `A` est une feuille alors sa branche gauche est la liste contenant son étiquette,
- sinon la branche gauche de `A` est égale à la concaténation de l'étiquette de la racine de `A` et de la branche gauche du premier sous-arbre de `A`.

Écrire une fonction `ag-branche-gauche` qui, étant donné un arbre général, retourne sa branche gauche. Par exemple

```
(ag-branche-gauche (ag-A1)) → (s a)
(ag-branche-gauche (ag-A2)) → (a b c)
```

Question 8 : On définit récursivement la branche droite d'un arbre :

- si `A` est une feuille alors sa branche droite est la liste contenant son étiquette,
- sinon la branche droite de `A` est égale à la concaténation de l'étiquette de la racine de `A` et de la branche droite du *dernier* sous-arbre de `A`.

Écrire une fonction `ag-branche-droite` qui, étant donné un arbre général, retourne sa branche droite. Par exemple

```
(ag-branche-droite (ag-A1)) → (s u b)
(ag-branche-droite (ag-A2)) → (a f)
```

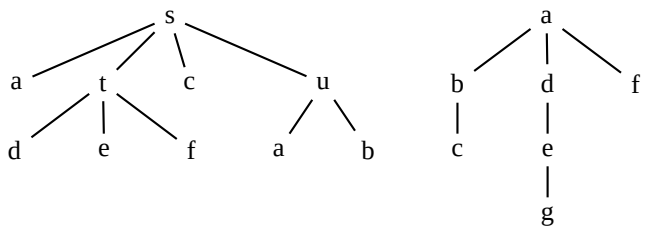
Exercice 13 – Représentation des arbres généraux par des arbres binaires

Cet exercice travaille sur la représentation des arbres généraux par des arbres binaires. Pour le traiter il est bon d'avoir fait tous les exercices concernant les arbres binaires et les arbres généraux.

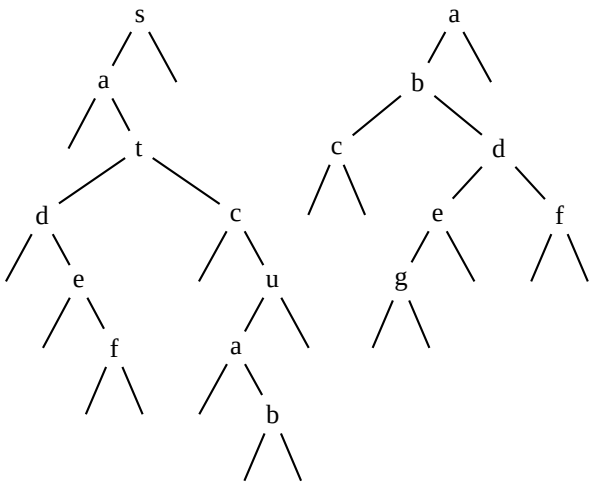
Le but de cet exercice est d'implanter les fonctions de la barrière d'abstraction des arbres généraux en utilisant les fonctions de la barrière d'abstraction des arbres binaires.

Il faut d'abord comprendre et visualiser la représentation d'un arbre général par un arbre binaire. Cette transformation est souvent appelée *rotation fils-aîné-frère droit*. On transforme chaque nœud *v* (d'arité quelconque) de l'arbre général en un nœud *binaire b* : le fils gauche de *b* (dans l'arbre binaire) est le premier fils de *v* (dans l'arbre général) ; et le fils droit de *b* (dans l'arbre binaire) est le frère-droit de *v* (dans l'arbre général). Ainsi le nœud racine de l'arbre général devient la racine de l'arbre binaire, et le sous -arbre droit à la racine de l'arbre binaire est toujours vide.

Par exemple les deux arbres généraux suivants :



sont respectivement représentés par les deux arbres binaires suivants :



Du fait des définitions mutuellement récursives des arbres généraux et des forêts, la représentation par arbre binaire doit traiter les arbres et les forêts, ces deux représentations s'épaulant mutuellement.

Tout arbre général peut être représenté par un arbre binaire comme suit :

- si l'arbre est réduit à une feuille d'étiquette `x`, on le représente par l'arbre binaire de racine `x` et dont les sous-arbre gauche et droit sont vides,
- sinon un arbre constitué d'une racine `x` et d'une forêt `f` est représenté par l'arbre binaire
  - de racine `x`,
  - dont le sous-arbre gauche est l'arbre binaire représentant la forêt `f`
  - et dont le sous-arbre droit est l'arbre vide.

Et toute forêt d'arbre généraux `(A1 A2 ... An)` peut être représentée par un arbre binaire dont :

- la racine est la racine du premier arbre `A1`,
- le sous-arbre gauche est l'arbre binaire représentant la forêt des sous-arbres de `A1`,
- le sous-arbre droit est l'arbre binaire représentant la forêt `(A2 ... An)`.

Puisque l'on peut représenter les arbres généraux par des arbres binaires, on peut donc décrire les fonctions permettant de manipuler les arbres généraux en termes de fonctions sur les arbres binaires. La majeure partie de cet exercice, consiste à **implanter les fonctions de la barrière d'abstraction des arbres généraux à l'aide des fonctions de la barrière d'abstraction des arbres binaires**. La dernière question étudie la transformation des caractéristiques : par exemple la liste infixe des nœuds de l'arbre binaire B représentant A est la même que la liste suffixe des nœuds de l'arbre général A,

Question 1 : Écrire le prédicat `ag-feuille?` (qui ne fait pas partie de la barrière d'abstraction des arbres généraux, mais c'est un bon exercice pour s'échauffer :).

```
;;; ag-feuille? : ArbreGeneral[alpha] -> bool
;;; (ag-feuille? A) rend vrai ssi A est une feuille (c-a-d un arbre réduit à une racine)
```

Remarquez que le paramètre `A` de la fonction `ag-feuille?` est un arbre général, mais on considère sa représentation en arbre binaire, donc le corps de la définition utilise les fonctions de la barrière d'abstraction des arbres binaires.

Question 2 : Écrire l'accesseur `ag-etiquette`

```
;;; ag-etiquette : ArbreGeneral[alpha] -> alpha
;;; (ag-etiquette A) rend l'étiquette de la racine de l'arbre A (ou l'étiquette de A si A est une feuille)
```

Question 3 : Écrire l'accesseur `ag-foret`

```
;;; ag-foret : ArbreGeneral[alpha] -> Foret[alpha]
```

```
;;; (ag-foret A) rend la forêt sous la racine de A, éventuellement vide si A est une feuille.
```

Ne pas oublier que les arbres généraux et les forêts sont manipulés à travers leurs représentations par des arbres binaires.

Question 4 : Écrire le constructeur `ag-noeud`

```
;;; ag-noeud : alpha * Foret[alpha] -> ArbreGeneral[alpha]
;;; (ag-noeud e F) rend l'arbre formé de la racine d'étiquette e et des
;;; sous-arbres F ; si F est vide, rend la feuille d'étiquette e
```

Ne pas oublier que les arbres généraux et les forêts sont manipulés à travers leurs représentations par des arbres binaires.

Question 5 : Faut-il modifier les fonctions `ag-profondeur` , `ag-liste-feuilles` , `ag-prefixe` , `ag-suffixe` , `ag-branche-gauche` définis dans les exercices sur les arbres généraux pour calculer ces différentes caractéristiques des arbres généraux ?

Question 6 :

La représentation des arbres généraux par des arbres binaires préserve de nombreuses caractéristiques. Prouvez par exemple que pour tout arbre général A, représenté par l'arbre binaire B :

1. la liste préfixe de l'arbre binaire B est la même que la liste préfixe de l'arbre général A,
2. la liste infixe de l'arbre binaire B est la même que la liste suffixe de l'arbre général A,
3. la branche gauche de l'arbre binaire B est la même que la la branche gauche de l'arbre général A.
4. la profondeur gauche de l'arbre binaire B est égale à la la profondeur de l'arbre général A.

Écrire le prédicat `verifie-proprietes` qui, étant donné un arbre général, vérifie les propriétés ci-dessus.

2.2 Arbres Cardinaux

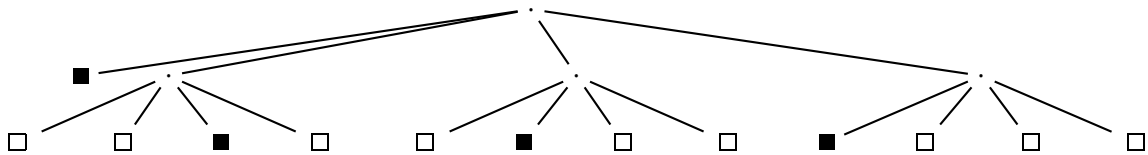
Exercice 14 – Arbres cardinaux

Un *arbre cardinal* est

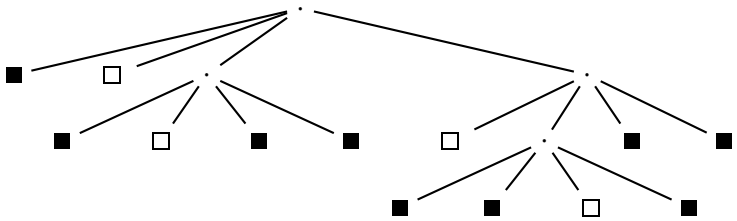
- soit réduit à une feuille, pleine ou vide
- soit formé d'un nœud racine (sans étiquette) et de quatre sous-arbres : nord-ouest, nord-est, sud-ouest et sud-est, qui sont eux-mêmes des arbres cardinaux.

La figure suivante montre deux arbres cardinaux, que l'on nommera par la suite C1 et C2.

Arbre cardinal C1 :



Arbre cardinal C2 :



Remarque : les arbres cardinaux sont largement utilisés en géométrie algorithmique pour représenter des images « bitmap ».

Le but de cet exercice est de construire des arbres cardinaux et d'en calculer différentes caractéristiques : hauteur, liste des feuilles, en manipulant les arbres cardinaux à travers une barrière d'abstraction.

Le type `ArbreCard` représente les arbres cardinaux. La barrière d'abstraction permettant de manipuler des arbres cardinaux est formée

- des constructeurs (`ac-f-pleine`) (resp. (`ac-f-vide`)), pour construire un arbre cardinal réduit à une feuille pleine (resp. vide) et (`ac-noeud no ne so se`) pour construire un arbre à partir de quatre arbres cardinaux
- des reconnaisseurs (`ac-f-pleine? a`) , resp. (`ac-f-vide? a`) pour reconnaître si un arbre cardinal est une feuille pleine, resp. vide
- des accesseurs (`ac-n-o a`) , resp. (`ac-n-e a`) , resp. (`ac-s-o a`) , resp. (`ac-s-e a`) , pour accéder au sous-arbre nord-ouest, resp. nord-est, resp. sud-ouest, resp. sud-est, de `a`.

Par exemple C1 est l'arbre obtenu par l'application (`cardEx1`)

```
;;; cardEx1 :  -> ArbreCard
;;; (cardEx1) rend l'arbre C1
(define (cardEx1)
  (let ((no (ac-f-pleine))
        (ne (ac-noeud (ac-f-vide) (ac-f-vide)
                      (ac-f-pleine)(ac-f-vid e) ))
        (so (ac-noeud (ac-f-vide) (ac-f-pleine)
                      (ac-f-vide) (ac-f-vide)))
        (se (ac-noeud (ac-f-pleine) (ac-f-vide)
                      (ac-f-vide) (ac-f-vide))))
    (ac-noeud no ne so se)))
```

Voici les spécifications des fonctions de la barrière d'abstraction

```
;;; ac-f-pleine : -> ArbreCard
;;; (ac-f-pleine) rend un arbre cardinal réduit à une feuille pleine

;;; ac-f-vide : -> ArbreCard
;;; (ac-f-vide) rend un arbre cardinal réduit à une feuille vide

;;; ac-noeud : ArbreCard * ArbreCard * ArbreCard * ArbreCard  -> ArbreCard
;;; (ac-noeud no ne so se) rend un arbre cardinal dont les quatre sous-arbres
;;; nord-ouest, nord-est, sud-ouest et sud-est sont no, ne, so et se

;;; ac-f-pleine? : ArbreCard -> bool
;;; (ac-f-pleine? a) rend vrai ssi l'arbre cardinal a est une feuille pleine

;;; ac-f-vide? : ArbreCard -> bool
;;; (ac-f-vide? a) rend vrai ssi l'arbre cardinal a est une feuille vide

;;; ac-n-o : ArbreCard -> ArbreCard
;;; (ac-n-o a) rend le sous-arbre nord-ouest de l'arbre cardinal a

;;; ac-n-e : ArbreCard -> ArbreCard
;;; (ac-n-e a) rend le sous-arbre nord-est de l'arbre cardinal a

;;; ac-s-o : ArbreCard -> ArbreCard
;;; (ac-s-o a) rend le sous-arbre sud-ouest de l'arbre cardinal a

;;; ac-s-e : ArbreCard -> ArbreCard
;;; (ac-s-e a) rend le sous-arbre sud-est de l'arbre cardinal a
```

Question 1 :

Écrire la fonction (`cardEx2`) qui construit l'arbre cardinal `C2` de la figure précédente.

Question 2 :

Écrire la fonction `(ac-feuille? a)` qui reconnaît si un arbre cardinal est une feuille. Par exemple

```
(ac-feuille? (ac-f-vide))  → #T
(ac-feuille? (cardEx1))    → #F
```

Question 3 :

On définit récursivement la hauteur d'un arbre cardinal comme suit :

- si l'arbre est une feuille sa hauteur est 0,
- sinon sa hauteur est égale au maximum des hauteurs de ses sous-arbres nord-ouest, nord-est, sud-ouest et sud-est, augmenté de 1.

Écrire une fonction récursive `ac-hauteur` qui, étant donné un arbre cardinal, retourne sa hauteur. Par exemple

```
(ac-hauteur (cardEx1))  → 2
(ac-hauteur (cardEx2))  → 3
```

Question 4 :

On définit récursivement la liste des feuilles d'un arbre cardinal comme suit :

- si `a` est une feuille, alors la liste de ses feuilles a un seul élément : 1 si la feuille est pleine et 0 si la feuille est vide
- sinon la liste des feuilles de `a` est égale à la concaténation des listes des feuilles des sous-arbres nord-ouest, nord-est, sud-ouest et sud-est de `a`.

Écrire une fonction `(ac-liste-feuilles a)` qui, étant donné un arbre cardinal, retourne la liste de ses feuilles. Par exemple

```
(ac-liste-feuilles (cardEx1))  → (1 0 0 1 0 0 1 0 0 1 0 0 0)
(ac-liste-feuilles (cardEx2))  → (1 0 1 0 1 1 0 1 1 0 1 1 1)
```

Remarque : La liste de ses feuilles ne suffit pas à caractériser un arbre cardinal. Donner un exemple d'arbre cardinal ayant la même liste de feuilles que C1.

Question 5 :

On définit récursivement le *codage préfixe* d'un arbre cardinal comme suit :

- si l'arbre est réduit à une feuille pleine (resp. vide), son codage préfixe est la chaîne "1" (resp. "0")
- sinon le codage préfixe de l'arbre est égal à la concaténation de la chaîne "n" et des codages préfixes des sous-arbres nord-ouest, nord-est, sud-ouest et sud-est, dans cet ordre.

Écrire une fonction `(ac-affiche1 a)` qui, étant donné un arbre cardinal, retourne son codage préfixe. Par exemple

```
(ac-affiche1 (cardEx1))  → "n1n0010n0100n1000"
(ac-affiche1 (cardEx2))  → "n10n1011n0n110111"
```

Question 6 :

On peut aussi représenter un arbre cardinal par un paragraphe :

- si l'arbre est réduit à une feuille pleine (resp. vide), il est représenté par le paragraphe dont la seule ligne est "1" (resp. "0")
- sinon l'arbre est représenté par le paragraphe obtenu en faisant précéder d'un tiret chaque ligne de chacun des paragraphes représentant les sous-arbres nord-ouest, nord-est, sud-ouest et sud-est.

Écrire une fonction `(ac-affiche2 a)` qui, étant donné un arbre cardinal, retourne le paragraphe le représentant. Par exemple, `(ac-affiche2 (cardEx2))` retourne le paragraphe

```
"
-1
-0
--1
--0
--1
--1
--0
```

```
---1
---1
---0
---1
--1
--1
"
```

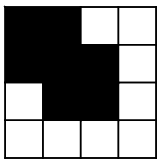
Remarquons que chaque ligne représente une feuille, en en donnant la nature (1 pour pleine et 0 pour vide) et la profondeur (le nombre de tirets).

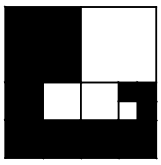
Question 7 :

Enfin, on peut afficher les arbres cardinaux au moyen d'images. On affiche un arbre cardinal dans un carré de côté c de la façon suivante :

- si l'arbre est réduit à une feuille pleine, on le représente par un carré plein de côté c
- si l'arbre est réduit à une feuille vide, on le représente par un carré vide de côté c
- si l'arbre est formé des quatre sous-arbres `no`, `ne`, `so` et `se` :
  - on partage le carré initial en quatre carrés de côté $c/2$: un carré nord-ouest (en haut à gauche), un carré nord-est (en haut à droite), un carré sud-ouest (en haut à gauche) et un carré sud-est (en bas à droite)
  - on affiche les sous-arbres `ne`, `ne`, `so` et `se` dans les carrés nord-ouest, nord-est, sud-ouest et sud-est, respectivement.

Écrire une fonction `ac-affiche3` qui réalise l'affichage d'un arbre cardinal dans un carré de côté 2. Par exemple, `(ac-affiche3 (cardEx1))` et `(ac-affiche3 (cardEx2))` produisent les affichages

cardEx1 : 

cardEx2 : 

Question 8 : Il s'agit ici d'implanter, de deux façons différentes, la barrière d'abstraction des arbres cardinaux

- Donner une implantation des arbres cardinaux à l'aide de S-expressions,
- Donner une implantation des arbres cardinaux à l'aide de vecteurs.

Table des matières