Seconde saison

Version 1.22

Sommaire

1. Notion de barrière d'abstraction

C'est une notion que vous connaissez déjà !

En effet, pour utiliser une fonction, nous avons dit qu'il fallait regarder sa spécification – et non sa définition –, spécification qui est une description, une « abstraction » de ce que rend la fonction.

```
;;; fn: Donnee -> Resultat
;;; (fn d) rend ...
++++++++++++++++++++++++
(define (fn d) ...)
```

Ainsi, on peut considérer qu'il y a une barrière entre la fin de la spécification et sa définition proprement dite et, lors d'une utilisation, on n'a pas besoin (et il faudrait même se l'interdire) de franchir cette barrière.

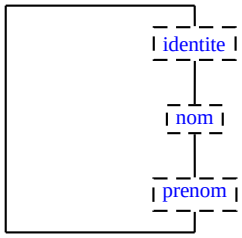
Nous avons vu aussi que, bien entendu, du côté de celui qui écrit la définition de la fonction – on dit qu'il l'implante –, il faut que cette implantation réponde à la spécification.

1.1. Barrière d'abstraction

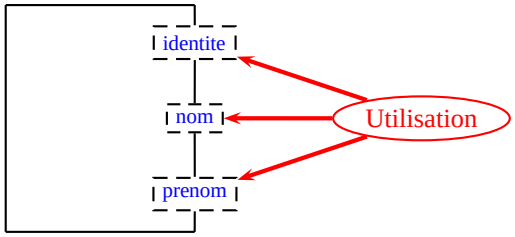
Souvent, on ne peut pas implanter une fonction toute seule sans tenir compte de l'implantation d'autres fonctions, pour avoir un ensemble cohérent.

Aussi, regroupe-t-on toutes ces fonctions dans ce qu'on appelle une **barrière d'abstraction**. Comme pour les fonctions, une barrière d'abstraction a un aspect abstrait (c'est l'ensemble des spécifications des fonctions) et un aspect concret (c'est l'implantation des fonctions).

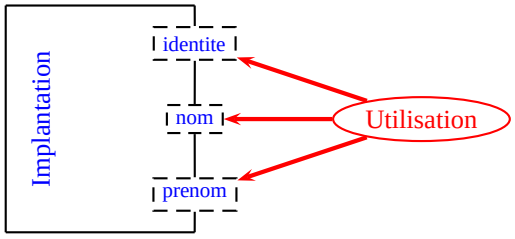
Prenons un exemple (uniquement didactique) : nous voudrions manipuler l'identité des individus, identité qui est composé d'un nom et d'un prénom. Les fonctions utiles sont alors `identite` (qui crée une identité à partir d'un nom et d'un prénom), `prenom` (qui rend le prénom d'une identité donnée) et `nom` (qui rend le nom d'une identité donnée).



Ensuite, lorsque l'on veut définir une fonction dont une donnée ou le résultat est un élément du domaine de la barrière d'abstraction, on n'utilise que les fonctions de la barrière d'abstraction (on s'interdit d'utiliser la connaissance que l'on pourrait avoir de l'implantation de cette barrière d'abstraction) :



Mais, bien entendu, il faut aussi implanter cette abstraction.



L'intérêt (fondamental en génie logiciel) est que l'on peut ensuite changer facilement l'implantation de la barrière d'abstraction (en général pour avoir une implantation plus performante) : il suffit de modifier cette implantation et,

Comme les fonctions utilisatrices de la barrière d'abstraction ne sont définies qu'avec les fonctions de cette barrière, sans utiliser la connaissance des détails de l'implantation, elles restent parfaitement valides.

1.2. Exemple de différentes implantations pour une même barrrière d'abstraction

Prenons l'exemple de l'identité des individus. La spécification de la barrière d'abstraction est :

```
;;; identite : string * string -> Identite
;;; (identite nom prenom) rend l'identité dont le nom est «nom» et le
;;; prénom est «prenom»

;;; nom : Identite -> string
;;; (nom id) rend le nom de l'identité «id»

;;; prenom : Identite -> string
;;; (prenom id) rend le prénom de l'identité «id»
```

Écrivons une fonction qui, étant donnée une identité, rend la chaîne de caractères composée du prénom de l'individu, suivi d'un espace et terminée par le nom de l'individu :

```
;;; ident->string : Identite -> string
;;; (ident->string id) rend la chaîne de caractères composée du prénom de l'individu «id»,
;;; suivi d'un espace et terminée par le nom de l'individu
(define  (ident->string id)
  (string-append (prenom id) " " (nom id)))
```

Il ne reste plus qu'à implanter la barrière d'abstraction. Vous pensez peut-être qu'il n'y a qu'une solution : un n-uplet qui comporte deux chaînes de caractères. Mais, tout de suite, on voit qu'il y en a deux selon que l'on mette le nom avant ou après le prénom. Ces deux solutions étant très proches, on ne voit pas pourquoi on changerait (mais, lorsque l'on écrit des fonctions sur cette structure de données, il faut savoir dans quel cas on est : il est alors beaucoup plus parlant d'utiliser les fonctions `nom` et `prenom` que les fonctions `car` et `cadr`). Voici cette implantation :

```
;;; identite : string * string -> Identite
;;; (identite nom prenom) rend l'identité dont le nom est «nom» et le
;;; prénom est «prenom»
(define (identite nom prenom)
  (list nom prenom))

;;; nom : Identite -> string
;;; (nom id) rend le nom de l'identité «id»
(define (nom id)
  (car id))

;;; prenom : Identite -> string
;;; (prenom id) rend le prénom de l'identité «id»
(define (prenom id)
  (cadr id))
```

Mais on pourrait aussi utiliser le type `Paragraphe`, en mémorisant le prénom dans la première ligne et le nom dans la seconde ligne :

```
;;; identite : string * string -> Identite
;;; (identite nom prenom) rend l'identité dont le nom est «nom» et le
;;; prénom est «prenom»
(define (identite nom prenom)
  (paragraphe (list prenom nom)))

;;; nom : Identite -> string
;;; (nom id) rend le nom de l'identité «id»
(define (nom id)
```

```
  (car (lignes id)))
```

```
;;; prenom : Identite -> string
;;; (prenom id) rend le prénom de l'identité «id»
(define (prenom id)
  (car (lignes id)))
```

Et il y en a (au moins) une quatrième ! L'identité des individus intervient dans tous les fichiers qui comportent des informations sur des personnes, par exemple le fichier de la scolarité qui contient votre cursus, vos notes... Or, dans ce fichier, les identités sont mémorisées sous forme d'une chaîne de caractères obtenue en concaténant le nom, puis une virgule et enfin le prénom. Ainsi, n'est-il pas aberrant d'avoir une autre implantation de cette barrière d'abstraction (nous ne donnons pas les définitions des fonctions `string-avant-virgule` et `string-apres-virgule` car elles sont hors programme) :

```
;;; identite : string * string -> Identite
;;; (identite nom prenom) rend l'identité dont le nom est «nom» et le prénom est «prenom»
(define (identite nom prenom)
  (string-append nom "," prenom))
```

```
;;; nom : Identite -> string
;;; (nom id) rend le nom de l'identité «id»
(define (nom id)
  (string-avant-virgule id))
```

```
;;; prenom : Identite -> string
;;; (prenom id) rend le prénom de l'identité «id»
(define (prenom id)
  (string-apres-virgule id))
```

avec

```
;;; string-avant-virgule : string -> string
;;; (string-avant-virgule s) rend le préfixe de «s» avant la première occurrence du caractère virgule
;;; HYPOTHÈSE: il y a une occurrrence  du caractère virgule dans «s»

;;; string-apres-virgule : string -> string
;;; (string-apres-virgule s) rend le suffixe de «s» après la première occurrence du caractère virgule
;;; HYPOTHÈSE: il y a une occurrrence  du caractère virgule dans «s»
```

1.3. Mise en œuvre en DEUG MIAS

Par la suite, nous voudrons souvent avoir plusieurs implantations d'une même barrière d'abstraction et tester des fonctions utilisatrices de cette barrière d'abstraction avec chaque implantation de la barrière. Afin de minimiser les « copier–coller », nous utiliserons une architecture logicielle particulière que nous décrivons ci-dessous sur notre exemple.

Ainsi, les définitions Scheme des différentes fonctions de l'exemple ci-dessus sont écrites dans deux fichiers :

- un fichier, nommé `identite1.scm`, contient une première version de la barrière d'abstraction (spécification et implantation),
- un fichier, nommé `identToString.scm`, contient une définition, utilisatrice de cette barrière d"abstraction, de la fonction `ident->string`.

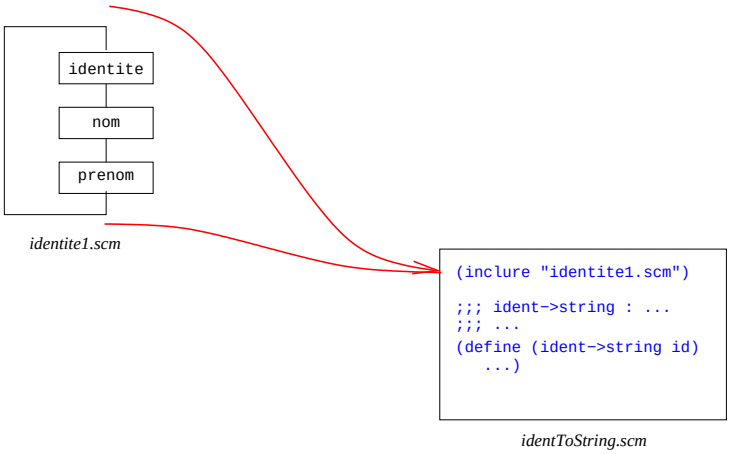
La définition de cette dernière fonction utilisant la barrière d'abstraction, nous demandons que le fichier `identite1.scm` soit inclus en début de fichier grâce à la fonction `inclure` qui a comme spécification :

```
;;; inclure : string ->
;;; (inclure s) inclue le fichier de nom «s». Tout se passe comme si le texte
;;; du fichier de nom «s» était écrit à la place de cette application.
```

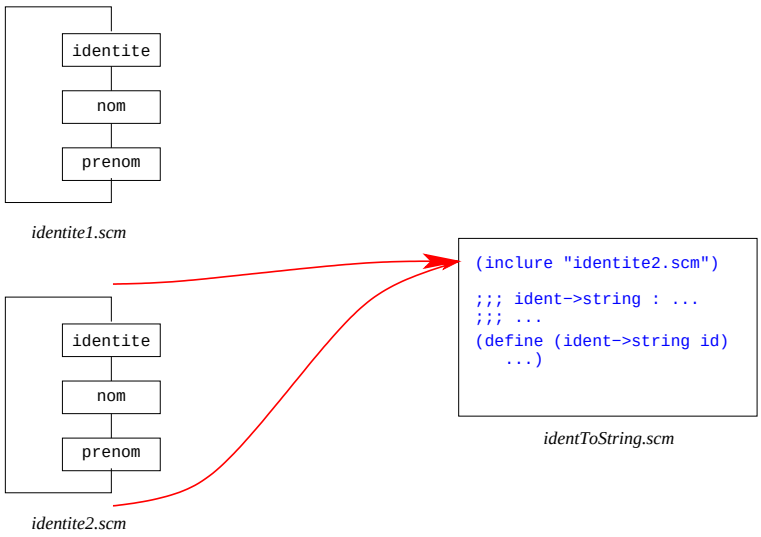
Ainsi, le fichier `identToString.scm` commence par :

```
;;;; Utilise identite
(inclure "identite1.scm")
```

On peut schématiser cette architecture par :



Si l'on veut une autre implantation de la barrière d'abstraction, il suffi t
- de l'écrire dans un fi chier nommé `identite2.scm` et,
- dans le fi chier `identToString.scm` de remplacer `(inclure "identite1.scm")` par `(inclure "identite2.scm")` :



Remarque : dans la pratique,
- pour que le test affi che la version de la barrière d'abstraction utilisée, dans le fi chier `identToString.scm`,
  - nous avons défi ni une fonction qui rend le nom du fi chier où se trouve la barrière d'abstraction et
  - nous appliquons cette fonction dans l'application de `inclure` et dans un « display » qui affi che ce nom lors de l'exécution ;
- cette défi nition est écrite plusieurs fois, avec les noms des fi chiers des différentes versions, et nous les commentons systématiquement sauf celle que nous voulons tester.

Ainsi, notre fi chier `identToString.scm` (qui permet de tester trois versions de la barrière d'abstraction) est :

```
;;;; Id : identToString.scm,v1.12002/09/2015 : 18 : 07titouExp
;;;; Utilise identite :
; (define (version-identite) "identite1.scm")
(define (version-identite) "identite2.scm")
```

```
; (define (version-identite) "identite3.scm")
(inclure (version-identite))

;;; ident->string : Identite -> string
;;; (ident->string id) rend la chaîne de caractères composée du prénom de l'individu «id»,
;;; suivi d'un espace et terminée par le nom de l'individu
(define  (ident->string id)
  (string-append (prenom id) " " (nom id)))

;;; essai de ident->string :
(display  (string-append "Essai avec " (version-identite)))
(verifier ident->string
  (ident->string (identite "Curie" "Pierre")) == "Pierre Curie")
```

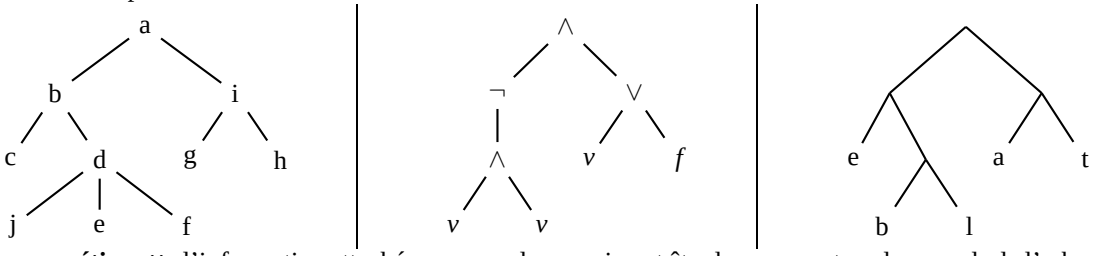
Pour s'auto-évaluer
Exercices d'assouplissement[1]

2. Notion d'arbre

2.1. Notion d'arbre

Un **arbre** est un ensemble de **nœuds** organisés de façon hiérarchique à partir d'un nœud distingué qui est la **racine**, les autres nœuds étant eux-mêmes structurés sous forme d'arbres, les **sous-arbres immédiats** de l'arbre.

Voici trois exemples d'arbres :

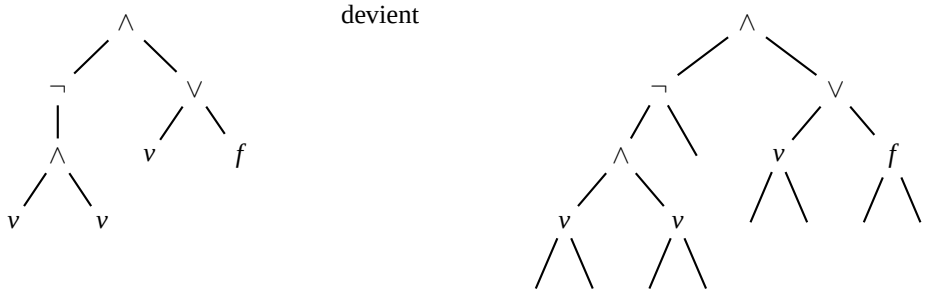


On nomme **étiquette** l'information attachée aux nœuds, ce qui peut être le cas pour tous les nœuds de l'arbre (deux premiers exemples ci-dessous) ou uniquement aux feuilles (dernier exemple ci-dessous).

Intuitivement, les **feuilles** sont les nœuds qui « n'ont rien au-dessous ».

L'arbre dont l'ensemble des nœuds est vide est nommé l'**arbre vide**. Selon l'usage que l'on veut faire des arbres, l'arbre vide peut exister ou non dans l'ensemble des arbres considérés. Ainsi, on peut voir les arbres de deux façons :

- exactement comme le suggère les dessins ci-dessus : il n'y a rien au dessous des feuilles (et une feuille est alors un arbre qui n'a pas de sous-arbre immédiat),
- en considérant que les sous-arbres des feuilles existent et sont (tous) égaux à l'arbre vide. Pour indiquer l'existence de ces arbres vides, dans les dessins, nous ajouterons les traits au-dessous des feuilles :



Un arbre est un **arbre binaire** lorsque c'est l'arbre vide ou lorsqu'il a exactement deux sous-arbres immédiats et que ceux-ci sont eux-mêmes des arbres binaires.

[1] `http://127.0.0.1:20022/q-ab-barriere-1.quizz`

2.2. Différentes structures de données « arbre »

La notion d'arbre peut aider pour représenter toute situation hiérarchique :
- sommaire d'un livre (une partie comportant des chapitres qui comportent des sections qui comportent des sous-sections...),
- analyse grammaticale d'une phrase (une phrase est composée d'un sujet, d'un verbe et d'un complément d'objet, le sujet étant un groupe nominale composé d'un article, d'un adjectif et d'un nom, le verbe...),
- classifi cation des animaux,
- arbre généalogique,
- ...

Notons qu'il existe plusieurs « sortes » d'arbres comme structures de données, selon l'endroit où l'on attache les informations (tous les nœuds ou uniquement aux feuilles), selon le nombre de sous-arbres immédiats de chaque nœud (fi xe, borné ou non borné) et selon l'existence ou non de l'arbre vide.

Nous verrons par la suite les « arbres binaires » (information attachée à chaque nœud, existence de l'arbre vide, pour chaque nœud exactement deux sous-arbres immédiats) et les « arbres généraux » (information attachée à chaque nœud, pas d'arbre vide, pour chaque nœud nombre quelconque de sous-arbres immédiats).

3. Barrière d'abstraction des arbres binaires

3.1. Caractéristiques des arbres binaires

Parmi la famille des arbres, les arbres binaires sont caractérisés par :

- information attachée à chaque nœud,
- existence de l'arbre vide,
- chaque nœud a exactement deux sous-arbres immédiats.

3.2. Barrière d'abstraction des arbres binaires

Notation : lorsque le type des étiquettes attachées aux nœuds est α, on notera `ArbreBinaire[`α`]` le type des arbres binaires.

Dans ce paragraphe, nous donnons la spécifi cation de la barrière d'abstraction des arbres binaires. Nous vous montrerons une implantation de cette barrière plus tard. Pour que vous puissiez l'utiliser avant de voir cette implantation, nous vous fournissons une autre implantation que vous pouvez utiliser, en TME ou chez vous, sous réserve d'avoir chargé « miastools.plt » (*cf.* installation du céderom).

Dans la bibiothèque MIAS de DrScheme, la barrière d'abstraction des arbres binaires comporte l'ensemble des fonctions (constructeurs, reconnaisseurs et accesseurs) suivantes :

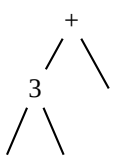
3.2.1. Constructeurs

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

;;; ab-noeud : α * ArbreBinaire[α] * ArbreBinaire[α] -> ArbreBinaire[α]
;;; (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».
```

Exemples

Pour construire l'arbre suivant :



on peut écrire :

```
(ab-noeud '+
          (ab-noeud 3 (ab-vide) (ab-vide))
          (ab-vide))
```

On ne sait rien de la représentation des arbres ainsi construits. Autrement dit, si vous utilisez l'implantation fournie et si vous tapez une telle expression – toute seule, sans être un argument d'une application –, son évaluation affiche `#<abstract:AB>` qui indique que le résultat est un arbre binaire mais qu'il est manipulé au niveau abstrait, c'est-à-dire que l'on ne veut pas montrer son implantation.

Ainsi, on peut tester les fonctions dont la donnée est un arbre binaire. Pour tester des fonctions dont le résultat est un arbre binaire, la barrière d'abstraction contient une fonction supplémentaire, la fonction `ab-expression`, qui a comme spécification :

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

Autrement dit, cette fonction rend une expression Scheme qui permet de construire (en utilisant les deux constructeurs) l'arbre donné. Par exemple l'application :

```
(ab-expression
 (ab-noeud '+
           (ab-noeud 3 (ab-vide) (ab-vide))
           (ab-vide)))
```

a comme valeur

```
(ab-noeud '+ (ab-noeud 3 (ab-vide) (ab-vide)) (ab-vide))
```

(le résultat est bien l'expression donnée comme argument à la fonction `ab-expression`).

Pour construire des arbres binaires, il est très pratique d'avoir aussi à disposition la fonction `ab-feuille` de spécification :

`;;;` *ab-feuille : α -> ArbreBinaire[α]*
`;;;` *(ab-feuille e) rend l'arbre binaire constitué d'une feuille, ayant «e» comme étiquette*

Sa définition peut être :

```
(define (ab-feuille e)
  (ab-noeud e (ab-vide) (ab-vide)))
```

Voici un exemple d'application :

```
(ab-expression
 (ab-noeud '+
           (ab-feuille 3)
           (ab-vide)))
```

a comme valeur

```
(ab-noeud '+ (ab-noeud 3 (ab-vide) (ab-vide)) (ab-vide))
```

La barrière d'abstraction comporte aussi des reconnaisseurs et des accesseurs :

3.2.2. Reconnaisseurs

`;;;` *ab-noeud? : ArbreBinaire[α] -> bool*
`;;;` *(ab-noeud? B) rend vrai ssi «B» n'est pas l'arbre vide.*

`;;;` *ab-vide? : ArbreBinaire[α] -> bool*
`;;;` *(ab-vide? B) rend vrai ssi «B» est l'arbre vide.*

3.2.3. Accesseurs

```
;;; ab-etiquette : ArbreBinaire[α] -> α
;;; (ab-etiquette B) rend l'étiquette de la racine de l'arbre «B»
;;; ERREUR lorsque «B» ne satisfait pas «ab-noeud?»

;;; ab-gauche : ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-gauche B) rend le sous-arbre gauche de «B»
;;; ERREUR lorsque «B» ne satisfait pas «ab-noeud?»

;;; ab-droit : ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-droit B) rend le sous-arbre droit de «B»
;;; ERREUR lorsque «B» ne satisfait pas «ab-noeud?»
```

3.2.4. Propriétés remarquables

Les fonctions de la barrière d'abstraction des arbres binaires vérifient les propriétés suivantes :

Pour tout couple d'arbres binaires, `G` et `D`, et toute valeur, `v` :

```
(ab-etiquette (ab-noeud v G D))  →  v
(ab-gauche (ab-noeud v G D))  →  G
(ab-droit (ab-noeud v G D))  →  D
```

Pour tout arbre binaire **non vide** `B`

```
(ab-noeud (ab-etiquette B)
          (ab-gauche B)
          (ab-gauche B))     →  B
```

3.3. Exemples d'utilisations de la barrière d'abstraction

Pour commencer, on peut définir la fonction `ab-feuille?` dont la spécification est :

```
;;; ab-feuille? : ArbreBinaire[α] -> bool
;;; (ab-feuille? B) rend #t ssi «B» est un arbre binaire réduit à une feuille
;;; ERREUR lorsque l'arbre est vide
```

Sa définition peut être :

```
(define (ab-feuille? B)
  (and (ab-vide? (ab-gauche B))
       (ab-vide? (ab-droit B))))
```

et alors

```
(ab-feuille? (ab-feuille 3))
```

a comme valeur `#t`
et

```
(ab-feuille? (ab-noeud '+
                       (ab-feuille 3)
                       (ab-vide)))
```

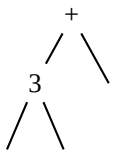
a comme valeur `#f`

3.3.1. Profondeur d'un arbre

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

- la profondeur de l'arbre vide est 0,
- la profondeur d'un arbre non vide est égale au maximum des profondeurs de ses sous-arbres immédiats, augmenté de 1.

Par exemple, la profondeur de l'arbre



est égale à 2.

Pour écrire une définition de la fonction `ab-profondeur` :

```
;;; ab-profondeur : ArbreBinaire[α] -> nat
;;; (ab-profondeur B) rend la profondeur de l'arbre «B»
```

il suffit de suivre la définition mathématique de la fonction :

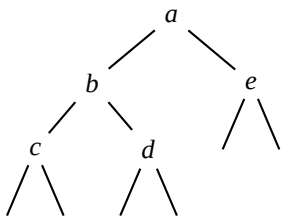
```
(define (ab-profondeur B)
  (if (ab-vide? B)
      0
      (+ 1 (max (ab-profondeur (ab-gauche B))
                (ab-profondeur (ab-droit B))))))
```

3.3.2. Liste infixe des étiquettes d'un arbre

La liste infixe des étiquettes d'un arbre binaire est définie comme suit :

- si `B` est l'arbre vide alors sa liste infixe est vide,
- sinon, la liste infixe de `B` est égale à la concaténation de la liste infixe du sous-arbre gauche de `B` suivi de l'étiquette de la racine de `B` et de la liste infixe du sous-arbre droit de `B`.

Par exemple, la liste infixe de l'arbre



est égale à `(c b d a e)`.

Pour écrire une définition de la fonction

```
;;; ab-liste-infixe : ArbreBinaire[α] -> LISTE[α]
;;; (ab-liste-infixe B) rend la liste infixe des étiquettes de l'arbre «B»
```

il suffit de suivre la définition mathématique de la fonction :

```
(define (ab-liste-infixe B)
  (if (ab-vide? B)
      '()
      (append (ab-liste-infixe (ab-gauche B))
              (cons (ab-etiquette B)
                    (ab-liste-infixe (ab-droit B))))))
```

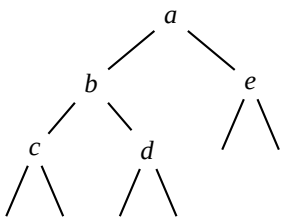
3.3.3. Affichage d'un arbre

Dans ce paragraphe, nous étudions la fonction `ab-affichage` qui permet un affichage plus visuel que la fonction `ab-expression` : `B` étant un arbre binaire, `(ab-affichage B)` rend un paragraphe tel que

- si `B` est l'arbre vide, le paragraphe résultat est composé d'une seule ligne, la ligne `vide` ;
- sinon, le paragraphe résultat est composé

– d'une ligne qui contient l'étiquette de l'arbre,
– les lignes suivantes contiennent la représentation présentement défi nie du sous-arbre gauche, chaque ligne étant préfi xé par le caractère tiret (« - »),
– les lignes suivantes contiennent la représentation présentement défi nie du sous-arbre droit, chaque ligne étant également préfi xé par le caractère tiret (« - »).

Ainsi, si `ex-ab-2` est l'arbre



`(ab-affichage ex-ab-2)` a comme valeur

```
"
a
-b
--c
---vide
---vide
--d
---vide
---vide
-e
--vide
--vide
"
```

Spécification

```
;;; ab-affichage : ArbreBinaire[α] -> Paragraphe
;;; (ab-affichage B) rend, lorsque «B» est l'arbre vide, le paragraphe constitué de la
;;; seule ligne constituée par le mot qui s'affiche "vide" et, lorsque l'arbre «B» n'est pas
;;; vide, le paragraphe dont la première ligne est l'étiquette de la racine de l'arbre «B»
;;; et dont les lignes suivantes sont égales à cette représentation des deux
;;; sous-arbres de «B», toutes les lignes étant préfixées par un tiret
```

Première implantation

Idée

Considérons l'exemple précédent :

```
a
-b
--c
---vide
---vide
--d
---vide
---vide
-e
--vide
--vide
```

La première ligne est la représentation de l'étiquette de la racine et les lignes suivantes correspondent à la représentation du sous-arbre gauche et du sous-arbre droit, chaque ligne étant précédée d'un tiret :

```
-a
--b
--c
---vide
---vide
--d
---vide
---vide
-e
--vide
--vide
```

(ab−affichage (ab−gauche B))

(ab−affichage (ab−droit B))

Définition de la fonction

Ainsi, pour implanter la fonction, il suffit :

- à partir de la représentation des sous-arbres gauche et droit (appel récursif),
- de « fabriquer » la liste des lignes de ces deux représentations (en utilisant les fonctions `lignes` et `append`),
- de « maper » une fonction qui ajoute un tiret en tête de ligne sur cette liste de lignes,
- de concaténer la représentation de l'étiquette de la racine de l'arbre donné devant le paragraphe correspondant à la liste obtenue.

Cette expression n'étant pas définie lorsque l'arbre est vide, la définition de la fonction est :

```
(define (ab-affichage B)
  ;; add-tiret-prefixe : Ligne -> Ligne
  ;; (add-tiret-prefixe ligne) rend la ligne obtenue en ajoutant un tiret devant «ligne»
  (define (add-tiret-prefixe ligne)
    (string-append "-" ligne))

  ;; expression de (ab-affichage B):
  (if (ab-vide? B)
      (paragraphe '("vide"))
      (paragraphe-cons
       (->string (ab-etiquette B))
       (paragraphe
        (map add-tiret-prefixe
             (append (lignes (ab-affichage (ab-gauche B)))
                     (lignes (ab-affichage (ab-droit B)))))))))
```

Seconde implantation

Avec la définition précédente, l'évaluation de la fonction `ab-affichage` passe son temps à fabriquer un paragraphe à partir d'une liste de lignes (en utilisant la fonction `paragraphe`) et à refabriquer la liste de lignes à partir de ce paragraphe (en utilisant la fonction `lignes`. Pour éviter cela, on peut définir une fonction auxiliaire, `liste-lignes-affichage`, qui rend la liste des lignes du paragraphe recherché :

```
;;; ab-affichage : ArbreBinaire[α] -> Paragraphe
;;; (ab-affichage B) rend, lorsque «B» est l'arbre vide, le paragraphe constitué de la
;;; seule ligne constituée par le mot qui s'affiche "vide" et, lorsque l'arbre «B» n'est pas
;;; vide, le paragraphe dont la première ligne est l'étiquette de la racine de l'arbre «B»
;;; et dont les lignes suivantes sont égales à cette représentation des deux
;;; sous-arbres de «B», toutes les lignes étant préfixées par un tiret
(define (ab-affichage B)
  ;; add-tiret-prefixe : Ligne -> Ligne
  ;; (add-tiret-prefixe ligne) rend la ligne obtenue en ajoutant un tiret devant «ligne»
  (define (add-tiret-prefixe ligne)
    (string-append "-" ligne))

  ;; liste-lignes-affichage : ArbreBinaire[α] -> LISTE[Ligne]
  ;; (liste-lignes-affichage B) rend (lignes (ab-affichage B))
  (define (liste-lignes-affichage B)
```

```
    (if (ab-vide? B)
        '("vide")
        (cons
         (->string (ab-etiquette B))
         (map add-tiret-prefixe
              (append (liste-lignes-affichage (ab-gauche B))
                      (liste-lignes-affichage (ab-droit B))))) ) )

  ;; expression de (ab-affichage B):
  (paragraphe (liste-lignes-affichage B)) )
```

Troisième implantation

Pour avoir une définition plus efficace[2] de cette fonction, on peut aussi utiliser la technique que nous avons expliquée lorsque nous avons écrit la définition de la fonction `triangle2` : on définit une fonction interne qui rend un paragraphe obtenu en préfixant l'affichage d'un arbre donné par un préfixe donné et on applique cette fonction avec un préfixe vide et l'arbre donné :

```
(define (ab-affichage B)
  ;; aff-Aux : Ligne * ArbreBinaire[α] -> LISTE[Ligne]
  ;; (aff-Aux pref B) rend la liste de lignes obtenue en préfixant chaque ligne de
  ;; (lignes (ab-affichage B)) par la chaîne «pref»
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire

  ;; expression de (ab-affichage B) :
  (paragraphe (aff-Aux "" B)) )
```

Pour la définition de `aff-Aux`, il suffit de suivre la spécification de `ab-affichage` (le préfixe pour les appels récursifs étant égal à la concaténation du préfixe donné et d'un tiret) :

```
(define (ab-affichage B)
  ;; aff-Aux : Ligne * ArbreBinaire[α] -> Paragraphe
  ;; (aff-Aux pref B) rend le paragraphe obtenu en préfixant chaque ligne du paragraphe
  ;; « (ab-affichage B) » par la chaîne «pref»
  (define (aff-Aux pref B)
    (if (ab-vide? B)
        (paragraphe (list (string-append pref "vide")))
        (let ((pref2 (string-append pref "-")))
          (paragraphe-cons (string-append pref (->string (ab-etiquette B)))
                           (paragraphe-append (aff-Aux pref2 (ab-gauche B))
                                              (aff-Aux pref2 (ab-droit B)))) ) ) )

  ;; expression de (ab-affichage B) :
  (aff-Aux "" B) )
```

Remarque (efficacité) : la structure des deux définitions que nous avons données pour cette fonction sont similaires sauf que la première définition exécute systématiquement en plus, pour chaque appel récursif, un « map » sur les lignes de la représentration (rappelons que le temps de l'exécution d'un map est de l'ordre de la longueur de la liste donnée). Ainsi, la seconde définition est bien plus performante que la première.

Pour s'auto-évaluer

[2] Voir la remarque à la fin de la présente section

4. Arbres binaires de recherche

Les arbres binaires sont des structures de données très utiles en informatique. Dans cette section, comme exemple d'utilisation des arbres binaires, nous étudions les « arbres binaires de recherche ».

4.1. Introduction et définition

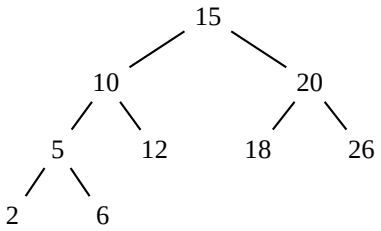
En informatique, il est fréquent d'avoir besoin de gérer un ensemble d'informations (comme l'ensemble des étudiants de DEUG-MIAS). Dans cette gestion, l'une des opérations fondamentales est de savoir si un individu donné (connu par son nom ou par son numéro de carte d'étudiant) fait, ou ne fait pas, partie de la base. Pour ce faire, il existe une structure de données efficace, la structure d'arbre binaire de recherche.

Un arbre binaire de recherche est un arbre binaire vide ou un arbre binaire qui possède les propriétés suivantes :

- 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.

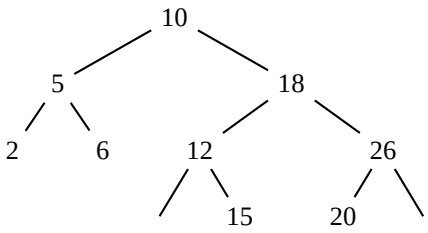
Par la suite, pour que les opérations de comparaison soient définies (et soient < et >), nous considèrerons que les étiquettes sont des nombres.

Voici un exemple d'arbre binaire de recherche :



Les propriétés caractéristiques de la notion d'arbre binaire de recherche sont intéressantes pour l'efficacité de la recherche car elles permettent de ne chercher un élément que dans un des deux sous-arbres, recherche qui s'effectue en ne recherchant que dans un des deux sous-sous-arbres...

Notons que, dans un arbre binaire de recherche, c'est l'ensemble de ses étiquettes qui nous intéresse : deux arbres binaires de recherche peuvent donc être « équivalents ». Par exemple, en tant qu'arbres binaires de recherche, l'arbre précédent est « équivalent » à l'arbre suivant :



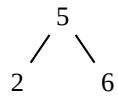
4.2. Spécification

Nous nommerons `ArbreBinRecherche` le type des arbres binaires de recherche, et, par la suite, nous voudrions définir les fonctions suivantes :

```
;;; abr-recherche : Nombre * ArbreBinRecherche -> ArbreBinRecherche + #f
;;; (abr-recherche x ABR) rend l'arbre de racine «x», lorsque «x» apparait dans «ABR»
;;; et renvoie #f si «x» n'apparait pas dans «ABR»
```

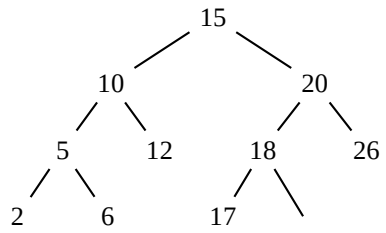
[3] `http://127.0.0.1:20022/q-ab-barriere-arbre-bin-1.quizz`

Par exemple, dans notre implantation, si `abr1` est l'arbre binaire de recherche dessiné ci-dessus, `(abr-recherche 5 abr1)` a comme valeur



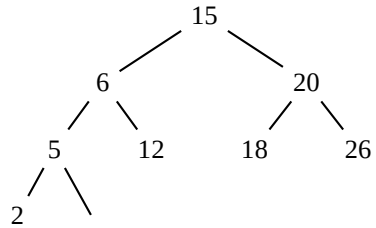

```
;;; abr-ajout : Nombre * ArbreBinRecherche -> ArbreBinRecherche
;;; (abr-ajout x ABR) rend l'arbre «ABR» lorque «x» apparait dans «ABR» et, lorsque
;;; «x» n'apparait pas dans «ABR», rend un arbre binaire de recherche qui contient «x»
;;; et toutes les étiquettes qui apparaissent dans «ABR»
```

Par exemple, dans notre implantation – et une autre implantation peut rendre un arbre binaire de recherche équivalent –, `(abr-ajout 17 abr1)` a comme valeur




```
;;; abr-moins : Nombre * ArbreBinRecherche -> ArbreBinRecherche
;;; (abr-moins x ABR) rend l'arbre «ABR» lorque «x» n'apparait pas dans «ABR» et,
;;; lorsque «x» apparait dans «ABR», rend un arbre binaire de recherche qui
;;; contient toutes les étiquettes qui apparaissent dans «ABR» hormis «x».
```

Par exemple, dans notre implantation – et une autre implantation peut rendre un arbre binaire de recherche équivalent –, `(abr-moins 10 abr1)` a comme valeur



Noter que lorsque l'élément à supprimer est l'étiquette d'un nœud interne, obligatoirement, l'arbre binaire est profondemment modifié.

4.3. Implantation

Naturellement, nous implantons les arbres binaires de recherche à l'aide de la barrière d'abstraction `ArbreBinaire`.

4.3.1. Fonction `abr-recherche`

Rappelons la spécification :

```
;;; abr-recherche : Nombre * ArbreBinRecherche -> ArbreBinRecherche + #f
;;; (abr-recherche x ABR) rend l'arbre de racine «x», lorsque «x» apparait dans «ABR»
;;; et renvoie #f si «x» n'apparait pas dans «ABR»
```

L'idée (récursive) de l'implantation est très simple :

- lorsque l'élément recherché est égal à l'étiquette de la racine, on l'a trouvé et le résultat est l'arbre lui-même ;
- sinon, on doit rechercher l'élément dans le sous-arbre gauche ou dans le sous-arbre droit suivant qu'il est inférieur ou supérieur à l'étiquette de la racine.

D'où la définition :

```
(define (abr-recherche x ABR)
  (if (ab-vide? ABR)
      #f
      (let ((e (ab-etiquette  ABR)))
        (cond ((= x e) ABR)
              ((< x e) (abr-recherche x (ab-gauche ABR)))
              (else (abr-recherche x (ab-droit ABR)))))))
```

4.3.2. Fonction `abr-ajout`

Rappelons la spécifi cation :

```
;;; abr-ajout : Nombre * ArbreBinRecherche -> ArbreBinRecherche
;;; (abr-ajout x ABR) rend l'arbre «ABR» lorque «x» apparait dans «ABR» et, lorsque
;;; «x»  n'apparait pas dans «ABR», rend un arbre binaire de recherche qui contient «x»
;;; et toutes les étiquettes qui apparaissent dans «ABR»
```

L'idée (récursive) de l'implantation est également très simple :

- lorsque l'élément à ajouter est égal à l'étiquette de la racine, le résultat est l'arbre donné ;
- lorsque l'élément à ajouter est inférieur à l'étiquette de la racine, il doit être dans le sous-arbre gauche, aussi le résultat est l'arbre
  - dont l'étiquette est l'étiquette de l'arbre donné,
  - dont le sous-arbre gauche est l'arbre obtenu en ajoutant l'élément dans le sous-arbre gauche de l'arbre donné,
  - dont le sous-arbre droit est le sous-arbre droit de l'arbre donné ;
- lorsque l'élément à ajouter est supérieur à l'étiquette de la racine, il doit être dans le sous-arbre droit et on raisonne comme ci-dessus en inversant sous-arbre gauche et sous-arbre droit.

D'où la défi nition :

```
(define (abr-ajout x ABR)
  (if (ab-vide? ABR)
      (ab-noeud x (ab-vide) (ab-vide))
      (let ((e (ab-etiquette  ABR)))
        (cond ((= x e) ABR)
              ((< x e) (ab-noeud e
                                 (abr-ajout x (ab-gauche ABR))
                                 (ab-droit ABR)))
              (else (ab-noeud e
                              (ab-gauche ABR)
                              (abr-ajout x (ab-droit ABR)))))))))
```

Remarque : le recueil d'exercices contient une autre implantation de cette fonction.

4.3.3. Fonction `abr-moins`

Rappelons la spécifi cation :

```
;;; abr-moins : Nombre * ArbreBinRecherche -> ArbreBinRecherche
;;; (abr-moins x ABR) rend l'arbre «ABR» lorque «x» n'apparait pas dans «ABR» et,
;;; lorsque «x»  apparait dans «ABR», rend un arbre binaire de recherche qui
;;; contient toutes les étiquettes qui apparaissent dans «ABR» hormis «x».
```

L'idée initiale est encore très simple (mais ça se complique par la suite...) :

- lorsque l'élément à supprimer est inférieur à l'étiquette de la racine, cet élément ne peut être que dans le sous-arbre gauche, et le résultat est donc l'arbre
  - dont l'étiquette est l'étiquette de l'arbre donné,
  - dont le sous-arbre gauche est l'arbre obtenu en supprimant l'élément du sous-arbre gauche de l'arbre donné,
  - dont le sous-arbre droit est le sous-arbre droit de l'arbre donné ;
- lorsque l'élément à supprimer est supérieur à l'étiquette de la racine, cet élément ne peut être que dans le sous-arbre droit, et on raisonne comme ci-dessus en inversant sous-arbre gauche et sous-arbre droit ;

— reste le cas où l'élément à supprimer est égal à l'étiquette de la racine ; il faut le supprimer, mais on doit alors complètement reconstruire l'arbre, et de tel sorte que cet arbre vérifie la propriété « être un arbre binaire de recherche » ; compliqué... comme d'habitude, dans ce cas, on utilise et spécifie une nouvelle fonction :

```
(define (abr-moins x ABR)
  (if (ab-vide? ABR)
      (ab-vide)
      (let ((e (ab-etiquette  ABR)))
        (cond ((= x e) (moins-racine ABR))
              ((< x e) (ab-noeud e
                                 (abr-moins x (ab-gauche ABR))
                                 (ab-droit ABR)))
              (else (ab-noeud e
                              (ab-gauche ABR)
                              (abr-moins x (ab-droit ABR))))))))
```

La spécification de `moins-racine` étant :

```
;;; moins-racine : ArbreBinRecherche -> ArbreBinRecherche
;;; (moins-racine ABR) rend l'arbre binaire de recherche qui contient toutes
;;; les étiquettes qui apparaissent dans «ABR» hormis l'étiquette de sa racine.
;;; ERREUR lorsque l'arbre «ABR» est vide
```

Comment implanter cette fonction ?

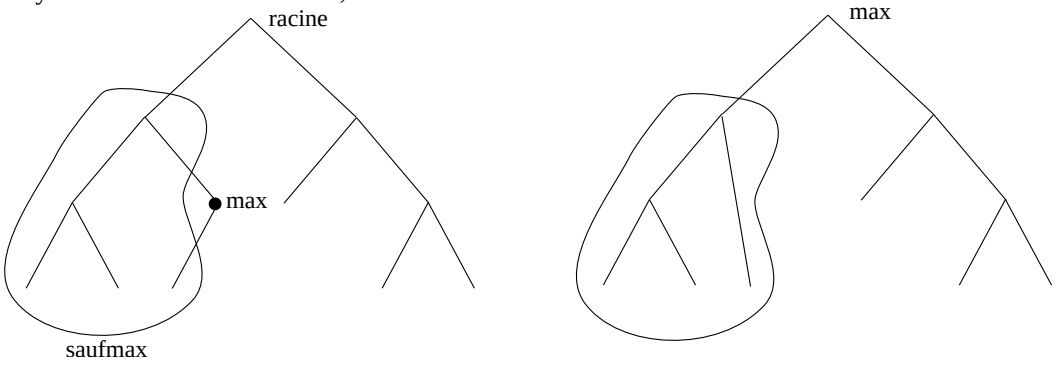
Première idée :

```
(define (moins-racine0 ABR)
  (reduce abr-ajout
          (ab-gauche ABR)
          (ab-liste-infixe (ab-droit ABR))))
```

Seconde idée :

Deux situations entraînent une solution évidente : lorsque l'un des deux sous-arbres est vide, le résultat est l'autre sous-arbre. Dans le cas contraire, on peut essayer de « conserver » un des deux sous-arbres, le droit par exemple : pour obtenir un arbre qui vérifie la propriété « arbre binaire de recherche », il faut alors que la racine de l'arbre résultat soit le plus grand élément du sous-arbre gauche. Ainsi, le résultat est un arbre

- ayant comme racine le plus grand élément du sous-arbre gauche de l'arbre donné (noter que c'est celui qui est au bout de la branche la plus à droite de ce sous-arbre gauche),
- ayant comme sous-arbre gauche, le sous-arbre gauche de l'arbre donné privé de son plus grand élément,
- ayant comme sous-arbre droit, le sous-arbre droit de l'arbre donné :



Ainsi, pour implanter la fonction `moins-racine`, on a besoin de la fonction `max-sauf-max` de spécification :

```
;;; max-sauf-max : ArbreBinRecherche -> NUPLET[Nombre ArbreBinRecherche]
;;; (max-sauf-max ABR) rend le couple formé de la plus grande étiquette présente
;;; dans l'arbre «ABR» et d'un arbre binaire de recherche qui contient toutes les
;;; étiquettes qui apparaissent dans «ABR» hormis ce maximum
```

;;; *ERREUR lorsque l'arbre «ABR» est vide*

La définition de la fonction `moins-racine` étant alors :

```
(define (moins-racine ABR)
  (cond ((ab-vide? (ab-gauche ABR))
         (ab-droit ABR))
        ((ab-vide? (ab-droit ABR))
         (ab-gauche ABR))
        (else
         (let ((m-sm-ss-ab-g (max-sauf-max (ab-gauche ABR))))
           (ab-noeud (car m-sm-ss-ab-g)
                     (cadr m-sm-ss-ab-g)
                     (ab-droit ABR))))))
```

Ne reste plus qu'à implanter la fonction `max-sauf-max`. Comme nous l'avons noté, le plus grand des éléments présents dans un arbre binaire de recherche se trouve au bout de la branche la plus à droite de cet arbre. Ainsi, la recherche du maximum et la constitution de l'arbre privé de son maximum s'effectuent par l'exploration successive des sous-arbres droits :

```
;;; max-sauf-max : ArbreBinRecherche -> NUPLET[Nombre ArbreBinRecherche]
;;; (max-sauf-max ABR) rend le couple formé de la plus grande étiquette présente
;;; dans l'arbre «ABR» et d'un arbre binaire de recherche qui contient toutes les
;;; étiquettes qui apparaissent dans «ABR» hormis ce maximum
;;; ERREUR lorsque l'arbre «ABR» est vide
(define (max-sauf-max ABR)
  (if (ab-vide? (ab-droit ABR))
      (list (ab-etiquette ABR) (ab-gauche ABR))
      (let ((m-sm-ss-ab-d (max-sauf-max (ab-droit ABR))))
        (list (car m-sm-ss-ab-d)
              (ab-noeud (ab-etiquette ABR)
                        (ab-gauche ABR)
                        (cadr m-sm-ss-ab-d))))))
```

5. Implantations des arbres binaires

Dans cette section, pour implanter les arbres binaires, nous étudions les notions Scheme de Sexpressions et de vecteurs.

Ainsi, nous pourrons implanter les arbres binaires de deux façons différentes. Noter que nous ne comparerons pas les performances de ces implantations, une telle étude étant hors du programme de ce cours.

5.1. Notion de Sexpression

Considérons l'expression Scheme, `(- (+ (- 95) 5) (+ 10 3))`. Cela ressemble à une liste (de trois éléments) de type `LISTE[`α`]`, le premier élément étant `-`, le second élément étant `(+ (- 95) 5)` et le dernier élément étant `(+ 10 3)`; à nouveau, `(+ (- 95) 5)` ressemble à une liste de type `LISTE[`α`]` (de trois éléments), le premier élément étant `+`, le second élément étant `(- 95)` et le dernier élément étant `5`; à nouveau, `(- 95)` ressemble à une liste (de deux éléments)... Mais ce ne sont pas des listes de type `LISTE[`α`]` puisque les éléments ne sont pas de même type. Ce sont des **Sexpressions**.

5.1.1. Définition

Formellement, on peut définir les Sexpressions par

<*Sexpression*> → <*atome*>
`LISTE[`<*Sexpression*>`]`

<atome> ::= *<Nombre>*
<bool>
<string>
<Symbole>

(*<Nombre>*, *<bool>*, *<string>* et *<Symbole>* sont définis dans la carte de référence)

5.1.2. Spécification des fonctions primitives

Tout d'abord, nous devons pouvoir savoir si une Sexpression est un atome ou une liste de Sexpressions. Pour ce faire, il existe en Scheme la fonction `list?` :

```
;;; list?: Valeur -> bool
;;; (list? v) rend #t ssi v est une liste (éventuellement vide)
```

Les autres fonctions de base sont les fonctions que nous avons déjà vues (`car`, `cdr`, `pair?`...).

5.1.3. Une implantation des arbres binaires à l'aide des Sexpressions

La notion de Sexpression permet d'implanter efficacement – en Scheme – les arbres binaires (ainsi que tous les types d'arbres). En effet :

- lorsque l'arbre est vide, on peut le représenter par la liste vide (qui est également la Sexpression vide) ;
- lorsque l'arbre n'est pas vide, on peut le représenter par une liste contenant l'étiquette de la racine et les deux Sexpressions qui représentent ses sous-arbres immédiats. Notez que l'on a six ordres possibles (d'abord l'étiquette puis le sous arbre gauche et enfin le sous-arbre droit ou le sous-arbre gauche puis l'étiquette et enfin le sous-arbre droit...). Ces six possibilités sont aussi valables les unes que les autres. Il faut en choisir une et s'en souvenir pour toutes les fonctions qui opèrent sur les arbres non vides. Ci-dessous, nous donnons une implantation écrite en mettant systématiquement l'étiquette de la racine en premier et le sous-arbre gauche avant le sous-arbre droit. La compréhension de cette implantation est facile, aussi nous la donnons telle quelle, sans explication.

```
;;;; Implantation des arbres binaires à l'aide des Sexpressions. Pour les arbres
;;;; non vides, on met systématiquement l'étiquette de la racine en premier et
;;;; le sous-arbre gauche avant le sous-arbre droit.

;;;;;;;;;;;;;;;; Constructeurs

;;; ab-noeud : α * ArbreBinaire[α] * ArbreBinaire[α] -> ArbreBinaire[α]
;;; (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».
(define (ab-noeud e B1 B2)
  (list e B1 B2))

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

;;;;;;;;;;;;;;;; Reconnaisseurs

;;; ab-noeud? : ArbreBinaire[α] -> bool
;;; (ab-noeud? B) rend vrai ssi «B» n'est pas l'arbre vide.
(define (ab-noeud? B)
  (pair? B))

;;; ab-vide? : ArbreBinaire[α] -> bool
;;; (ab-vide? B) rend vrai ssi «B» est l'arbre vide.
(define (ab-vide? B)
  (not (ab-noeud? B)))
```

```
;;; ab-etiquette : ArbreBinaire[α] -> α
;;; (ab-etiquette B) rend l'étiquette de la racine de l'arbre «B»
;;; ERREUR lorsque «B» ne satisfait pas ab-noeud?
(define (ab-etiquette B)
  (car B))

;;; ab-gauche : ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-gauche B) rend le sous-arbre gauche de «B»
;;; ERREUR lorsque B ne satisfait pas ab-noeud?
(define (ab-gauche B)
  (cadr B))

;;; ab-droit : ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-droit B) rend le sous-arbre droit de «B»
;;; ERREUR lorsque «B» ne satisfait pas ab-noeud?
(define (ab-droit B)
  (caddr B))
```

5.2. Notion de Vecteur

Les listes, que nous avons vues dans les cours précédents, permettent de rassembler plusieurs valeurs avec la possibilité d'extraire immédiatement la première valeur (en utilisant `car`) ; mais si l'on veut extraire le $i^{ème}$ élément de la liste (i étant variable), on doit écrire une fonction qui parcourt cette dernière (en utilisant la fonction `cdr`).

Les vecteurs[4] sont des structures de données Scheme qui permettent d'agréger plusieurs valeurs – les composants du vecteur – avec la possibilité d'extraire immédiatement l'une de ces valeurs.

Plus précisément, un vecteur est caractérisé par des **indices**, une **longueur** et des **composants** :

- la longueur du vecteur est le nombre de ses composants,
- ses composants sont indicés par les entiers naturels compris entre 0 (inclus) et sa longueur (exclue) ;
- ainsi, le premier composant du vecteur est celui d'indice 0, le deuxième composant est celui d'indice 1... et le dernier composant est celui ayant comme indice sa longueur moins un.

Notons qu'il existe un vecteur particulier, le vecteur vide, dont la longueur est nulle et qui ne contient aucun composant.

Remarque : sous Drscheme, l'affi chage de la valeur d'un vecteur commence par le caractère #, continue par la longueur du vecteur et se termine par, entre parenthèses, la liste des valeurs des composants du vecteur.

5.2.1. Spécification des fonctions primitives

Constructeur

La fonction `vector`, qui peut avoir un nombre quelconque d'arguments, rend un vecteur dont les composants sont ses arguments, dans l'ordre où ils sont donnés :

```
;;; vector : Valeur ... -> Vecteur
;;; (vector x0 ...) rend le vecteur dont le premier composant (celui d'indice 0) est
;;; «x0», dant le deuxième composant est «x1»...
;;; (vector) rend le vecteur vide (de longueur 0 et qui ne contient aucune valeur)
```

Exemple

`(vector 'a 'b 'c)` →`#3(a b c)`

Accesseurs

```
;;; vector-length : Vecteur -> nat
;;; (vector-length V) rend la longueur de «V», c'est-à-dire son nombre de composants
```

Exemple

[4] On parle de tableaux dans d'autres langages de programmation (avec une nuance que nous ne verrons pas ici).

`(vector-length (vector 'a 'b 'c))` →3

Rappelons que le dernier indice d'un vecteur `V` est égal à `(- (vector-length V) 1)`.

Nous avons dit que la caractéristique d'un vecteur était de pouvoir accéder immédiatement à un composant d'un indice donné. Ceci se fait grâce à la fonction `vector-ref` :

```
;;; vector-ref : Vecteur * nat -> Valeur
;;; (vector-ref V k) rend le composant d'indice «k» du vecteur «V»
;;; ERREUR lorsque «k» n'appartient pas à l'intervalle [0 .. (vector-length V)[
```

Exemples

`(vector-ref (vector 'a 'b 'c) 0)` →`a`

`(vector-ref (vector 'a 'b 'c) 2)` →`c`

`(vector-ref (vector 'a 'b 'c) 3)` rend une erreur (`vector-ref: index 3 out of range [0, 2] for vector: #3(a b c)`)

Fonctions de conversion

Comme nous l'avons dit, les listes et les vecteurs représentent des suites d'éléments (mais les fonctions de base sont très différentes). Aussi existe-t-il deux fonctions, `vector->list` et `list->vector`, qui permettent de passer d'une structure de données à l'autre :

```
;;; vector->list : Vecteur -> LISTE[Valeur]
;;; (vector->list V) rend la liste des composants du vecteur «V»
```

Exemple

`(vector->list (vector 'a 'b 'c))` →`(a b c)`

```
;;; list->vector : LISTE[Valeur] -> Vecteur
;;; (list->vector L) rend le vecteur ayant comme premier composant le premier
;;; élément de la liste «L», comme deuxième composant le deuxième élément de la liste «L»...
```

Exemple

`(list->vector '(a b c))` →`#3(a b c)`

5.2.2. Une implantation des arbres binaires à l'aide des vecteurs

On peut implanter les arbres binaires en utilisant des vecteurs :

- l'arbre vide est représenté par le vecteur vide,
- un arbre non-vide est représenté par un vecteur de longueur trois ayant comme composants l'étiquette de la racine de l'arbre – par exemple en indice 0 –, la représentation du sous-arbre gauche – par exemple en indice 1 – et la représentation du sous-arbre droit – par exemple en indice 2.

Constructeurs

La fonction `ab-noeud` est implantée en utilisant la fonction `vector` :

```
;;; ab-noeud : α * ArbreBinaire[α] * ArbreBinaire[α] -> ArbreBinaire[α]
;;; (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».
(define (ab-noeud e B1 B2)
  (vector e B1 B2))
```

La fonction `ab-vide` est implantée par le vecteur vide en utilisant aussi la fonction `vector` (mais sans argument) :

```
;;; ab-vide : -> ArbreBinaire[α]
;;; (ab-vide) rend l'arbre binaire vide.
(define (ab-vide)
  (vector))
```

Reconnaisseurs

Pour savoir si un arbre est, ou n'est pas, vide il suffi t de comparer la longueur du vecteur à 0 :

```
;;; ab-vide? : ArbreBinaire[α] -> bool
;;; (ab-vide? B) rend vrai ssi «B» est l'arbre vide.
(define (ab-vide? B)
  (= (vector-length B) 0))

;;; ab-noeud? : ArbreBinaire[α] -> bool
;;; (ab-noeud? B) rend vrai ssi «B» n'est pas l'arbre vide.
(define (ab-noeud? B)
  (> (vector-length B) 0))
```

Accesseurs

Pour les accesseurs, on utilise la fonction `vector-ref`, l'étiquette étant en indice 0, le sous-arbre gauche en indice 1 et le sous-arbre droit en indice 2 :

```
;;; ab-etiquette : ArbreBinaire[α] -> α
;;; (ab-etiquette B) rend l'étiquette de la racine de l'arbre «B»
;;; ERREUR lorsque «B» ne satisfait pas ab-noeud?
(define (ab-etiquette B)
  (vector-ref B 0))

;;; ab-gauche : ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-gauche B) rend le sous-arbre gauche de «B»
;;; ERREUR lorsque B ne satisfait pas ab-noeud?
(define (ab-gauche B)
  (vector-ref B 1))

;;; ab-droit : ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-droit B) rend le sous-arbre droit de «B»
;;; ERREUR lorsque «B» ne satisfait pas ab-noeud?
(define (ab-droit B)
  (vector-ref B 2))
```

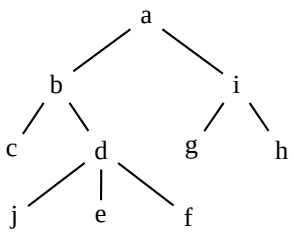
6. Arbres généraux

6.1. Caractéristiques des arbres généraux

Parmi la famille des arbres, les arbres généraux sont caractérisés par :

- information attachée à chaque nœud,
- non existence de l'arbre vide,
- chaque nœud a un nombre quelconque (éventuellement 0) de sous-arbres immédiats.

Voici un exemple d'arbre général :



Lorsque le type des étiquettes attachées aux nœuds est α, on notera `ArbreGeneral[`α`]` le type des arbres généraux.

Pour les arbres binaires, les sous-arbres immédiats étaient le sous-arbre gauche et le sous-arbre droit. Pour les arbres généraux, comme le nombre de sous-arbres immédiats est quelconque, on ne peut pas les caractériser aussi

Parallèlement, ainsi défini-on la notion de forêt : une **foret** est une liste d'arbres (généraux). Nous noterons `Foret[`α`]` le type des forêts dont les arbres appartiennent à `ArbreGeneral[`α`]`.
Ainsi, `Foret[`α`]` est le type `LISTE[ArbreGeneral[`α`]]`.

Ainsi, les sous-arbres immédiats d'un arbre général constituent une forêt : un arbre général est constitué par une racine, et l'étiquette attachée à cette racine, et par la forêt de ses sous-arbres immédiats.

La forêt des sous-arbres immédiats d'un arbre peut être la liste vide : l'arbre est alors réduit à une *feuille*.

6.2. Barrière d'abstraction des arbres généraux

6.2.1. Constructeur

La barrière d'abstraction des arbres généraux ne comporte qu'un constructeur, la fonction `ag-noeud` :

```
;;; ag-noeud : α * Foret[α] -> ArbreGeneral[α]
;;; avec Foret[α] == LISTE[ArbreGeneral[α]]
;;; (ag-noeud e foret) rend l'arbre formé de la racine d'étiquette «e» et,
;;; comme sous-arbres immédiats, les arbres de la forêt «foret».
```

Exemples

Définissons la fonction `ag-feuille` spécifiée par :

```
;;; ag-feuille : α -> ArbreGeneral[α]
;;; (ag-feuille e) rend l'arbre général réduit à une feuille ayant «e» comme étiquette
```

Pour ce faire, il suffit de remarquer qu'il s'agit de l'arbre ayant `e` comme étiquette de la racine est ayant la liste vide comme forêt des sous-arbres immédiats :

```
(define (ag-feuille e)
  (ag-noeud e '()))
```

Comme autre exemple, l'arbre dessiné ci-dessus peut être construit avec l'expression :

```
(ag-noeud
 'a
 (list (ag-noeud
        'b
        (list (ag-feuille 'c)
              (ag-noeud
               'd
               (list (ag-feuille 'j)
                     (ag-feuille 'e)
                     (ag-feuille 'f)))))
       (ag-noeud
        'i
        (list (ag-feuille 'g)
              (ag-feuille 'h)))))
```

6.2.2. Affichage

Comme pour les arbres binaires, on ne vous montre rien de la représentation des arbres ainsi construits. Aussi la barrière d'abstraction que nous vous fournissons comporte la fonction `ab-expression`, qui a comme spécification :

```
;;; ag-expression: ArbreGeneral[α] -> Sexpression
;;; (ag-expression g) rend une Sexpression reflétant la construction de l'arbre «g»
```

Exemple

`(ag-expression (ag-feuille 'a))` rend
`(ag-noeud 'a (list))`

6.2.3. Reconnaisseurs

Tout d'abord, revenons sur la notion de reconnaisseur. Lorsque l'on utilise une barrière d'abstraction, les objets manipulés sont (tous) « fabriqués » en utilisant les constructeurs et, lorsque l'on veut défi nir une fonction qui a comme donnée un tel objet, on a besoin de savoir avec quel constructeur il a été fabriqué. C'est le rôle des reconnaisseurs, qui permettent d'aiguiller les défi nitions entre « l'objet a été fabriqué par tel constructeur » ou « l'objet a été fabriqué par tel autre constructeur » ou... Mais alors, lorsque la barrière d'abstraction ne comporte qu'un constructeur, tout objet est « fabriqué » en utilisant ce constructeur et on n'a pas besoin de reconnaisseur.

Ainsi, dans la barrière d'abstraction des arbres généraux, comme il n'y a qu'un constructeur, il n'y a pas de reconnaisseur.

6.2.4. Accesseurs

`;;;` *ag-etiquette : ArbreGeneral[α] -> α*
`;;;` *(ag-etiquette g) rend l'étiquette de la racine de l'arbre «g».*

`;;;` *ag-foret : ArbreGeneral[α] -> Foret[α]*
`;;;` *avec Foret[α] == LISTE[ArbreGeneral[α]]*
`;;;` *(ag-foret g) rend la forêt des sous-arbres immédiats de «g».*

Exemple

On peut écrire la défi nition du prédicat `ag-feuille?` spécifi ée par :

`;;;` *ag-feuille? : ArbreGeneral[α] -> bool*
`;;;` *(ag-feuille? G) rend #t ssi «G» est un arbre réduit à une feuille*

en utilisant le prédicat `pair?` :

```
(define (ag-feuille? G)
  (not (pair? (ag-foret G))))
```

6.2.5. Propriétés remarquables

Les fonctions de la barrière d'abstraction des arbres généraux vérifi ent les propriétés suivantes :

Pour toute liste d'arbres généraux (ou forêt), `F`, et toute valeur, `v` :

```
(ag-etiquette (ag-noeud v F) )  →  v
(ag-foret (ag-noeud v F) )  →  F
```

Pour tout arbre général, `G` :

```
(ag-noeud (ag-etiquette G) (ag-foret G) )  →  G
```

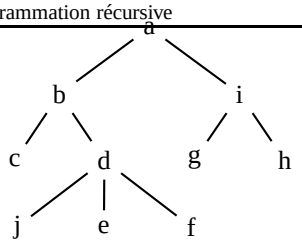
6.3. Exemples d'utilisations de la barrière d'abstraction

Les défi nitions récursives sur les arbres généraux sont plus compliquées que les défi nitions sur les arbres binaires car on ne peut pas atteindre directement tous les sous-arbres immédiats d'un arbre donné. En fait, souvent, pour défi nir une fonction ayant comme donnée un arbre général, nous aurons besoin de défi nir une autre fonction ayant comme donnée une forêt, ces deux fonctions étant mutuellement récursives (l'une appelle l'autre et inversement). On parle alors de récursivité croisée.

6.3.1. Profondeur d'un arbre

Comme nous l'avons fait pour les arbres binaires, nous voudrions calculer la profondeur des arbres généraux. La profondeur d'un arbre peut être défi nie comme étant égale à un de plus que la profondeur de la forêt constituée par ses sous-arbres immédiats, sachant que la profondeur d'une forêt est la profondeur de son arbre le plus profond (et elle est nulle pour la forêt vide).

Par exemple, la profondeur de l'arbre



est égale à

4

Remarquer que la profondeur d'un arbre est le nombre de niveaux lorsque l'on dessine l'arbre comme ci-dessus. Défi nissons donc la fonction `ag-profondeur` ayant comme spécifi cation

```
;;; ag-profondeur : ArbreGeneral[α] -> nat
;;; (ag-profondeur G) rend la profondeur de l'arbre «G»
```

Comme cela a été dit dans l'introduction, pour défi nir cette fonction, nous avons besoin de défi nir aussi la fonction qui rend la profondeur d'une forêt. Cette fonction n'étant qu'une fonction auxiliaire, nous la défi nissons à l'intérieur de la défi nition de la fonction `ag-profondeur` :

```
(define (ag-profondeur G)
  ;; profondeurForet : Foret[α] -> nat
  ;; (profondeurForet F) rend la profondeur de F, c.-à-d. le maximum des profondeurs
  ;; des arbres de F (rend 0 lorsque la forêt est vide).
... à faire
... à faire
... à faire
... à faire
  ;; expression de (ag-profondeur G) :
... à faire
```

D'après la défi nition de la profondeur d'un arbre, la profondeur de G est égale à un de plus que la profondeur de la forêt de ses sous-arbres immédiats :

```
(define (ag-profondeur G)
  ;; profondeurForet : Foret[α] -> nat
  ;; (profondeurForet F) rend la profondeur de F, c.-à-d. le maximum des profondeurs
  ;; des arbres de F (rend 0 lorsque la forêt est vide).
... à faire
... à faire
... à faire
... à faire
  ;; expression de (ag-profondeur G) :
  (+ 1 (profondeurForet (ag-foret G))))
```

Défi nissons maintenant la fonction qui calcule la profondeur d'une forêt. Une forêt étant une liste, ce n'est qu'un exercice de révisionsur les listes : la profondeur de la forêt est égale au maximum de la profondeur du premier arbre de la forêt et de la profondeur de la forêt obtenue, à partir de la forêt donnée, en supprimant son premier arbre. Cette relation de récurrence n'étant défi nie que pour une liste non vide, il faut extraire de l'appel récursif le cas où `(pair? F)` est faux. D'où la défi nition :

```
(define (ag-profondeur G)
  ;; profondeurForet : Foret[α] -> nat
  ;; (profondeurForet F) rend la profondeur de F, c.-à-d. le maximum des profondeurs
  ;; des arbres de F (rend 0 lorsque la forêt est vide).
  (define (profondeurForet F)
```

```
    (if (pair? F)
        (max (ag-profondeur (car F)) (profondeurForet (cdr F)))
        0))
  ;; expression de (ag-profondeur G) :
  (+ 1 (profondeurForet (ag-foret G))))
```

Autre définition

Pour la définition de la fonction `profondeurForet`, au lieu d'écrire explicitement la récursivité comme nous l'avons fait ci-dessus, on peut utiliser les fonctionnelles sur les listes.

Nous avons besoin de la profondeur de tous les arbres de la forêt, ce que nous pouvons calculer en utilisant la fonctionnelle `map` – appliquée à la fonction `ag-profondeur` et à `F` et qui rend la liste des profondeurs des arbres. Ensuite, nous devons calculer le maximum des éléments de cette liste, ce que l'on peut faire en appliquant la fonctionnelle `reduce` à cette liste, à la fonction `max` et en prenant `0` comme valeur initiale :

```
;;; ag-profondeur : ArbreGeneral[α] -> nat
;;; (ag-profondeur G) rend la profondeur de l'arbre «G»
(define (ag-profondeur G)
  ;; profondeurForet : Foret[α] -> nat
  ;; (profondeurForet F) rend la profondeur de «F», c.-à-d. le maximum des profondeurs
  ;; des arbres de «F» (rend 0 lorsque la foret est vide).
  (define (profondeurForet F)
    (reduce max 0 (map ag-profondeur F)))
  ;; expression de (ag-profondeur G) :
  (+ 1 (profondeurForet (ag-foret G))))
```

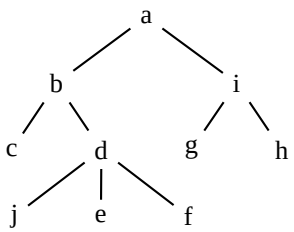
Une définition encore plus concise

En fait, en utilisant les fonctionnelles `map` et `reduce`, nous n'avons pas besoin de définir explicitement la fonction `profondeurForet` : dans la définition de `ag-profondeur` il suffit de substituer son appel par son expression (en n'oubliant pas les arguments de l'appel) :

```
;;; ag-profondeur : ArbreGeneral[α] -> nat
;;; (ag-profondeur G) rend la profondeur de l'arbre «G»
(define (ag-profondeur G)
  (+ 1 (reduce max 0 (map ag-profondeur (ag-foret G)))))
```

6.3.2. Liste préfixe des étiquettes d'un arbre

La liste préfixe des étiquettes d'un arbre général est égale à la concaténation de l'étiquette de sa racine et de la liste préfixe des étiquettes de chacun de ses sous-arbres immédiats. Par exemple, la liste préfixe de l'arbre



est égale à

```
(a b c d j e f i g h)
```

Cherchons une définition de la fonction `ag-liste-prefixe` de spécification :

```
;;; ag-liste-prefixe : ArbreGeneral[α] -> LISTE[α]
;;; (ag-liste-prefixe G) rend la liste préfixe des étiquettes de l'arbre «G»
```

Pour définir cette fonction, comme d'habitude, nous utilisons une fonction qui « fait le travail » pour la forêt de ses sous-arbres immédiats :

```
(define (ag-liste-prefixe G)
  ;; liste-prefixe-foret : Foret[α] -> LISTE[α] rend la concaténation
  ;; des listes préfixes des étiquettes des arbres de F
... à faire
... à faire
... à faire
... à faire
  ;; expression de (ag-liste-prefixe G) :
  (cons (ag-etiquette G) (liste-prefixe-foret (ag-foret G))))
```

La fonction `liste-prefixe-foret` se défi nit comme d'habitude pour les fonctions sur les listes/

```
(define (ag-liste-prefixe G)
  ;; liste-prefixe-foret : Foret[α] -> LISTE[α] rend la concaténation
  ;; des listes préfixes des étiquettes des arbres de F
  (define (liste-prefixe-foret F)
    (if (pair? F)
        (append (ag-liste-prefixe (car F))
                (liste-prefixe-foret (cdr F)))
        '()))
  ;; expression de (ag-liste-prefixe G) :
  (cons (ag-etiquette G) (liste-prefixe-foret (ag-foret G))))
```

Autre définition

Encore une fois, on peut aussi utiliser les fonctionnelles habituelles sur les listes :

```
;;; ag-liste-prefixe : ArbreGeneral[α] -> LISTE[α]
;;; (ag-liste-prefixe G) rend la liste préfixe des étiquettes de l'arbre «G»
(define (ag-liste-prefixe G)
  ;; liste-prefixe-foret : Foret[α] -> LISTE[α] rend la concaténation
  ;; des listes préfixes des étiquettes des arbres de «F»
  (define (liste-prefixe-foret F)
    (reduce append '() (map ag-liste-prefixe F)))
  ;; expression de (ag-liste-prefixe G) :
  (cons (ag-etiquette G) (liste-prefixe-foret (ag-foret G))))
```

Une définition encore plus concise

Et nous n'avons pas besoin de défi nir explicitement la fonction `liste-prefixe-foret` :

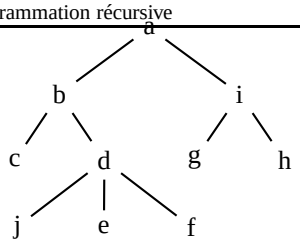
```
;;; ag-liste-prefixe : ArbreGeneral[α] -> LISTE[α]
;;; (ag-liste-prefixe G) rend la liste préfixe des étiquettes de l'arbre «G»
(define (ag-liste-prefixe G)
  (cons (ag-etiquette G)
        (reduce append
                '()
                (map ag-liste-prefixe (ag-foret G)))))
```

6.3.3. Affichage d'un arbre

Nous voudrions affi cher les arbres généraux comme nous l'avons fait pour les arbres binaires. Autrement dit, nous voudrions défi nir la fonction `ag-affichage` qui, étant donné un arbre général, rend un paragraphe

- dont la première ligne est égale à l'étiquette de la racine de l'arbre
- et dont les lignes suivantes sont obtenues en préfi xant par un tiret chaque ligne de la représentation ainsi défi nie de la suite des sous-arbres de l'arbre.

Par exemple `ag-affichage` appliquée à l'arbre



donne

```
"
a
-b
--c
--d
---j
---e
---f
-i
--g
--h
"
```

Spécification de la fonction

Ainsi, la fonction `ag-affichage` a comme spécifi cation :

```
;;; ag-affichage : ArbreGeneral[α] -> Paragraphe
;;; (ag-affichage G) rend le paragraphe dont la première ligne est l'étiquette de
;;; la racine de l'arbre «G» et dont les lignes suivantes sont égales à la
;;; représentation de la suite des sous-arbres de «G», toutes les lignes étant
;;; préfixées par un tiret
```

Une première implantation

Idée

Comme pour les arbres binaires, la première ligne est la représentation de l'étiquette de l'arbre et les lignes suivantes sont obtenues en préfi xant chaque ligne de la concaténation des représentations des sous-arbres par un tiret :

```
a
-b
--c
--d
---j
---e
---f
-i
--g
--h
```

Définition de la fonction

Comme pour les arbres binaires, nous défi nissons une fonction auxiliaire qui rend la liste des lignes du paragraphe résultat. La défi nition est alors (nous vous demandons de bien l'étudier) :

```
(define (ag-affichage G)
  ;; add-tiret-prefixe : Ligne -> Ligne
  ;; (add-tiret-prefixe ligne) rend la ligne obtenue en ajoutant un tiret devant «ligne»
  (define (add-tiret-prefixe ligne)
    (string-append "-" ligne) )
```

```
  ;; liste-lignes-affichage : ArbreGeneral[α] -> LISTE[Ligne]
  ;; (liste-lignes-affichage G) rend (lignes (ag-affichage G))
  (define (liste-lignes-affichage G)
    (cons (->string (ag-etiquette G))
                  (map add-tiret-prefixe
                       (reduce append
                               '()
                               (map liste-lignes-affichage
                                    (ag-foret G))))) )

  ;; expression de (ag-affichage G):
  (paragraphe (liste-lignes-affichage G)) )
```

Une seconde implantation

Pour écrire une définition plus efficace de cette fonction, comme nous l'avons fait pour les arbres binaires, nous pouvons utiliser une fonction auxiliaire, `aff-arbre`, qui affiche un arbre donné en préfixant chaque ligne par une chaîne donnée :

```
(define (ag-affichage G)
  ;; aff-arbre : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-arbre pref G) rend le paragraphe obtenu en préfixant chaque ligne de
  ;; (ag-affichage G) par la chaîne «pref»
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
  ;; expression de (ag-affichage G)
  (aff-arbre "" G))
```

Pour un arbre et un préfixe donnés, la fonction `aff-arbre` rend le paragraphe dont la première ligne est égale à l'image de l'étiquette de la racine de l'arbre donné précédée du préfixe donné et dont les lignes suivantes sont égales à la représentation de la forêt des sous-arbres de l'arbre donné préfixées par le préfixe donné et un nouveau tiret. En utilisant la nouvelle fonction auxiliaire `aff-foret`, spécifiée ci-dessous, la fonction `aff-arbre` peut être définie par :

```
(define (ag-affichage G)
  ;; aff-arbre : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-arbre pref G) rend le paragraphe obtenu en préfixant chaque ligne de
  ;; (ag-affichage G) par la chaîne «pref»
  (define (aff-arbre pref G)
    (paragraphe-cons (string-append pref (->string (ag-etiquette G)))
                     (aff-foret (string-append pref "-") (ag-foret G))))
  ;; aff-foret : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-foret pref F) rend le paragraphe obtenu en préfixant chaque ligne
  ;; de la représentation des arbres de «F» par la chaîne «pref»
... à faire
... à faire
```

```
... à faire
... à faire
... à faire
... à faire
... à faire
  ;; expression de (ag-affichage G)
  (aff-arbre "" G))
```

Reste à définir la fonction `aff-foret`. Donnons directement une définition qui utilise les fonctionnelles. Que doit-on faire ? Il faut concaténer tous les éléments de la liste des paragraphes représentant chacun des arbres de la forêt donnée, chaque ligne étant préfixée par le préfixe donné. Pour fabriquer cette liste, nous utilisons la fonctionnelle `map`, appliquée à la fonction, `aff-arbre-pref`, qui affiche un arbre en préfixant chaque ligne par le préfixe donné ; cette fonction doit être définie à l'intérieur de la définition de `aff-foret`, la variable contenant le préfixe étant globale dans cette fonction :

```
(define (ag-affichage G)
  ;; aff-arbre : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-arbre pref G) rend le paragraphe obtenu en préfixant chaque ligne de
  ;; (ag-affichage G) par la chaîne «pref»
  (define (aff-arbre pref G)
    (paragraphe-cons (string-append pref (->string (ag-etiquette G)))
                     (aff-foret (string-append pref "-") (ag-foret G))))
  ;; aff-foret : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-foret pref F) rend le paragraphe obtenu en préfixant chaque ligne
  ;; de la représentation des arbres de «F» par la chaîne «pref»
  (define (aff-foret pref F)
    ; aff-arbre-pref : ArbreGeneral[α] -> Paragraphe
    ; (aff-arbre-pref G) rend le paragraphe obtenu en préfixant chaque ligne de
    ; (ag-affichage G) par la chaîne «pref»
... à faire
... à faire
    ; expression de (aff-foret pref F)
    (reduce paragraphe-append (paragraphe '()) (map aff-arbre-pref F)))
  ;; expression de (ag-affichage G)
  (aff-arbre "" G))
```

Pour finir, la définition de la fonction `aff-arbre-pref` n'est qu'une application de la fonction `aff-arbre` :

```
(define (ag-affichage G)
  ;; aff-arbre : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-arbre pref G) rend le paragraphe obtenu en préfixant chaque ligne de
  ;; (ag-affichage G) par la chaîne «pref»
  (define (aff-arbre pref G)
    (paragraphe-cons (string-append pref (->string (ag-etiquette G)))
                     (aff-foret (string-append pref "-") (ag-foret G))))
  ;; aff-foret : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-foret pref F) rend le paragraphe obtenu en préfixant chaque ligne
  ;; de la représentation des arbres de «F» par la chaîne «pref»
  (define (aff-foret pref F)
    ; aff-arbre-pref : ArbreGeneral[α] -> Paragraphe
    ; (aff-arbre-pref G) rend le paragraphe obtenu en préfixant chaque ligne de
    ; (ag-affichage G) par la chaîne «pref»
    (define (aff-arbre-pref G)
      (aff-arbre pref G))
    ; expression de (aff-foret pref F)
    (reduce paragraphe-append (paragraphe '()) (map aff-arbre-pref F)))
```

```
  ;; expression de (ag-affichage G)
  (aff-arbre "" G))
```

Une définition plus concise

Encore une fois, nous n'avons pas besoin de défi nir explicitement la fonction sur les forêts :

```
(define (ag-affichage G)
  ;; aff-arbre : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-arbre pref G) rend le paragraphe obtenu en préfixant chaque ligne de
  ;; (ag-affichage G) par la chaîne «pref»
  (define (aff-arbre pref G)
    (let ((pref2 (string-append pref "-")))
      ; aff-arbre-pref2 : ArbreGeneral[α] -> Paragraphe
      ; (aff-arbre-pref2 G) rend le paragraphe obtenu en préfixant chaque
      ; ligne de (ag-affichage G) par la chaîne «pref2»
      (define (aff-arbre-pref2 G)
        (aff-arbre pref2 G))

      ; expression de (aff-arbre pref G)
      (paragraphe-cons (string-append pref (->string (ag-etiquette G)))
                       (reduce paragraphe-append
                               (paragraphe '())
                               (map aff-arbre-pref2 (ag-foret G))))))

  ;; expression de (ag-affichage G)
  (aff-arbre "" G))
```

6.4. Implantation des arbres généraux

Comme pour les arbres binaires, nous allons montrer que pour une barrière d'abstraction il peut y avoir différentes implantations : ici nous allons en voir trois, l'une à l'aide des Sexpressions, une autre à l'aide des vecteurs et la troisième à l'aide des vecteurs et des Sexpressions.

6.4.1. À l'aide des Sexpressions

Dans cette première implantation, nous représentons les arbres généraux à l'aide de Sexpressions : un arbre général est représenté par une liste dont le premier élément est l'étiquette de la racine et dont les éléments suivants constituent la forêt de ses sous-arbres immédiats.

Définition de la fonction `ag-noeud`

Rappelons sa spécifi cation :

```
;;; ag-noeud : α * Foret[α] -> ArbreGeneral[α]
;;; avec Foret[α] == LISTE[ArbreGeneral[α]]
;;; (ag-noeud e F) rend l'arbre formé de la racine d'étiquette «e» et, comme
;;; sous-arbres, les arbres de la forêt «F».
```

Puisque nous avons décidé que nous représenterions un arbre par une Sexpression dont le premier élément est l'étiquette de la racine et le reste de la liste est la forêt de ses sous-arbres, pour construire un arbre à partir de l'étiquette `e` de la racine et de la forêt `F` de ses sous-arbres, il suffi t d'utiliser la fonction `cons` :

```
(define (ag-noeud e F)
  (cons e F))
```

Définition de la fonction `ag-etiquette`

Rappelons sa spécifi cation :

```
;;; ag-etiquette : ArbreGeneral[α] -> α
;;; (ag-etiquette g) rend l'étiquette de la racine de l'arbre «g».
```

Nous avons dit que l'étiquette était le premier élément de la liste qui représente l'arbre :

```
(define (ag-etiquette g)
  (car g))
```

Définition de la fonction `ag-foret`

Rappelons sa spécification :

```
;;; ag-foret : ArbreGeneral[α] -> Foret[α]
;;; avec Foret[α] == LISTE[ArbreGeneral[α]]
;;; (ag-foret g) rend la forêt des sous-arbres immédiats de «g».
```

Nous avons dit que la forêt des sous-arbres était constituée par la liste qui représente l'arbre hormis son premier élément :

```
(define (ag-foret g)
  (cdr g))
```

6.4.2. À l'aide des vecteurs

Implantons maintenant les arbres généraux en utilisant des vecteurs : un arbre général est représenté par un vecteur, le premier composant du vecteur contenant l'étiquette de la racine de l'arbre et les composants suivants contenant les représentations des sous-arbres immédiats de l'arbre.

Définition de la fonction `ag-noeud`

Rappelons sa spécification :

```
;;; ag-noeud : α * Foret[α] -> ArbreGeneral[α]
;;; avec Foret[α] == LISTE[ArbreGeneral[α]]
;;; (ag-noeud e F) rend l'arbre formé de la racine d'étiquette «e» et, comme
;;; sous-arbres, les arbres de la forêt «F».
```

Nous devons rendre un vecteur dont nous connaissons le premier composant et la liste des autres composants. Pour ce faire, on peut utiliser la fonction `list->vector` :

```
(define (ag-noeud e F)
  (list->vector (cons e F)))
```

Définition de la fonction `ag-etiquette`

Rappelons sa spécification :

```
;;; ag-etiquette : ArbreGeneral[α] -> α
;;; (ag-etiquette g) rend l'étiquette de la racine de l'arbre «g».
```

L'étiquette étant le premier composant (celui d'indice 0) du vecteur qui représente l'arbre, il suffit d'appliquer la fonction `vector-ref` :

```
(define (ag-etiquette g)
  (vector-ref g 0))
```

Définition de la fonction `ag-foret`

Rappelons sa spécification :

```
;;; ag-foret : ArbreGeneral[α] -> Foret[α]
;;; avec Foret[α] == LISTE[ArbreGeneral[α]]
;;; (ag-foret g) rend la forêt des sous-arbres immédiats de «g».
```

La forêt des sous-arbres étant la liste des arbres contenu dans les composants (hormis le premier) du vecteur, nous utilisons la fonction `vector->list` :

```
(define (ag-foret g)
  (cdr (vector->list g)))
```

6.4.3. A l'aide des vecteurs et des Sexpressions

Dans cette implantation, nous représentons un arbre général par un vecteur, de longueur deux, dont le premier composant est l'étiquette de la racine et dont le second composant est la liste de ses sous-arbres immédiats (c'est donc la forêt de ses sous-arbres immédiats).

Définition de la fonction `ag-noeud`

Rappelons sa spécification :

```
;;; ag-noeud : α * Foret[α] -> ArbreGeneral[α]
;;; avec Foret[α] == LISTE[ArbreGeneral[α]]
;;; (ag-noeud e F) rend l'arbre formé de la racine d'étiquette «e» et, comme
;;; sous-arbres, les arbres de la forêt «F».
```

D'après la spécification de l'implantation des arbres donnée ci-dessus, cette fonction doit rendre un vecteur dont le premier composant est l'étiquette donnée et dont le second composant est la forêt donnée :

```
(define (ag-noeud e F)
  (vector e F))
```

Définition de la fonction `ag-etiquette`

Rappelons sa spécification :

```
;;; ag-etiquette : ArbreGeneral[α] -> α
;;; (ag-etiquette g) rend l'étiquette de la racine de l'arbre «g».
```

Il suffit de rendre le premier composant du vecteur (celui d'indice 0) :

```
(define (ag-etiquette g)
  (vector-ref g 0))
```

Définition de la fonction `ag-foret`

Rappelons sa spécification :

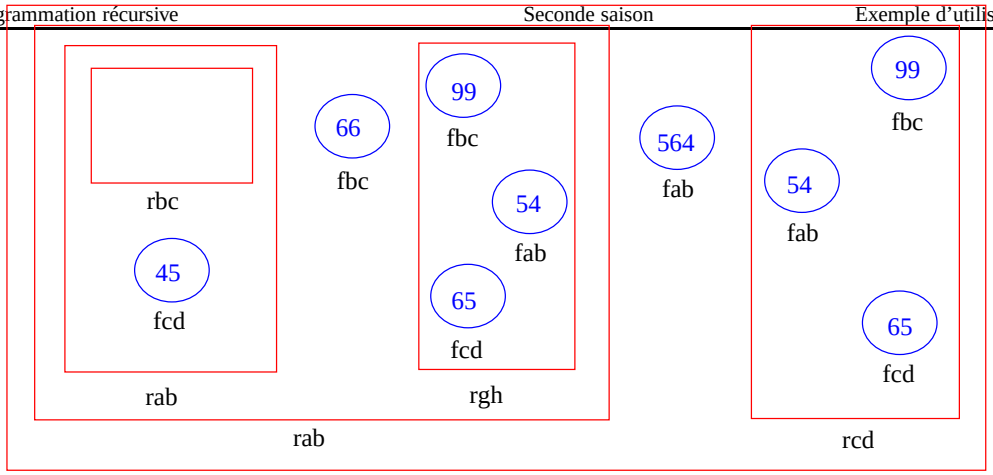
```
;;; ag-foret : ArbreGeneral[α] -> Foret[α]
;;; avec Foret[α] == LISTE[ArbreGeneral[α]]
;;; (ag-foret g) rend la forêt des sous-arbres immédiats de «g».
```

Il suffit de rendre le second composant du vecteur (celui d'indice 1) :

```
(define (ag-foret g)
  (vector-ref g 1))
```

7. Exemple d'utilisation des arbres généraux

Les arbres généraux permettent de représenter efficacement un système de fichiers comme il en existe sous Linux ou sous Windows. En effet, dans un tel système, des répertoires contiennent des fichiers et des répertoires qui contiennent à leur tour des fichiers et des répertoires qui...



7.1. Notion de descripteur de fichier

Dans un tel système, fichiers et répertoires sont décrits à l'aide de **descripteurs** qui, dans la réalité, comportent le nom du fichier ou du répertoire, la date de création, la date de modification... ainsi que des informations pour situer le fichier sur le disque. Dans notre modèle, le descripteur ne comportera que les informations suivantes :

- le nom du fichier ou du répertoire,
- est-ce un fichier ou un répertoire ?,
- pour un fichier, la taille du fichier.

Barrière d'abstraction `Descripteur`

Nous manipulerons les descripteurs à l'aide des fonctions suivantes :

```
;;;;;;;;; Constructeurs
;;; fichier : string * nat -> Descripteur
;;; (fichier nom taille) rend le descripteur du fichier de nom «nom» et
;;; de taille «taille»

;;; repertoire : string -> Descripteur
;;; (repertoire nom) rend le descripteur du répertoire de nom «nom»


;;;;;;;;; Reconnaisseurs
;;; fichier? : Descripteur -> bool
;;; (fichier? desc) rend #t ssi «desc» est le descripteur d'un fichier

;;; repertoire? : Descripteur -> bool
;;; (repertoire? desc) rend #t ssi «desc» est le descripteur d'un répertoire


;;;;;;;;; Accesseurs
;;; nom : Descripteur -> string
;;; (nom desc) rend le nom du répertoire ou du fichier dont le descripteur est «desc»

;;; taille : Descripteur -> nat
;;; ERREUR lorsque la description donnée est la description d'un répertoire
;;; (taille desc) rend la taille du fichier dont le descripteur est «desc»
```

Implantation de la barrière d'abstraction `Descripteur`

On peut implanter cette barrière d'abstraction en utilisant des vecteurs (de longueur deux pour les répertoires et 3 pour les fichiers) :

- le premier composant contient 'D (lorsque c'est un répertoire) ou 'F (lorsque c'est un fichier),

– le second composant contient le nom du répertoire ou du fichier,
– pour les fichiers, le troisième composant contient la taille du fichier.

Remarque : dans notre exemple, les nombres d'informations à mémoriser étant différents pour les fichiers et pour les répertoires, le premier composant du vecteur est inutile. Nous avons tout de même préféré la solution donnée ci-dessus, car, dans le futur, on pourrait être amené à ajouter, dans notre modèle, une autre information pour les répertoires (par exemple, le nombre d'éléments qu'il contient).

L'implantation est alors très simple et nous ne la commenterons pas :

```
;;;;;;;; Constructeurs
;;; fichier : string * nat -> Descripteur
;;; (fichier nom taille) rend le descripteur du fichier de nom «nom» et
;;; de taille «taille»
(define (fichier nom taille)
  (vector 'F nom taille))
;;; repertoire : string -> Descripteur
;;; (repertoire nom) rend le descripteur du répertoire de nom «nom»
(define (repertoire nom)
  (vector 'D nom))
;;;;;;;; Reconnaisseurs
;;; fichier? : Descripteur -> bool
;;; (fichier? desc) rend #t ssi «desc» est le descripteur d'un fichier
(define (fichier? desc)
  (equal? (vector-ref desc 0) 'F))
;;; repertoire? : Descripteur -> bool
;;; (repertoire? desc) rend #t ssi «desc» est le descripteur d'un répertoire
(define (repertoire? desc)
  (equal? (vector-ref desc 0) 'D))
;;;;;;;; Accesseurs
;;; nom : Descripteur -> string
;;; (nom desc) rend le nom du répertoire ou du fichier dont le descripteur est «desc»
(define (nom desc)
  (vector-ref desc 1))
;;; taille : Descripteur -> nat
;;; ERREUR lorsque la description donnée est la description d'un répertoire
;;; (taille desc) rend la taille du fichier dont le descripteur est «desc»
(define (taille desc)
  (vector-ref desc 2))
```

7.2. Représentation d'un système de fichiers

Un système de fichiers peut être représenté par un arbre général dont les étiquettes sont des descripteurs.

Noter que tout arbre ainsi défini ne représente pas un système de fichiers valide : dans un système de fichiers réel, tout nœud qui a comme étiquette un descripteur de fichier doit être une feuille et les noms des différents éléments d'un répertoire doivent être tous différents. Dans la suite, nous nommerons `Systeme` le type des éléments de `ArbreGeneral[Descripteur]` qui vérifient ces deux propriétés.

7.3. Définition de la fonction `du-s`

Nous voudrions définir la fonction `du-s` qui correspond à la commande `du -s` d'Unix :

```
(du-s  (syst1)) → 1111
```

```
;;; du-s : Systeme -> nat
;;; (du-s systeme) rend la quantité d'espace disque utilisée par les fichiers du
;;; système «systeme»
```

Autrement dit, `(du-s systeme)` rend la somme des tailles de tous les fichiers présents dans le système. La définition de cette fonction est une définition « classique » pour les arbres généraux :

– soit l'étiquette de l'arbe est le descripteur d'un fichier et le résultat est la taille de ce fichier,
– soit c'est un répertoire et il faut sommer les quantités d'espace disque utilisées par les différents éléments du répertoire, cette sommation pouvant être effectuée en utilisant la fonctionnelle `map` :

```
(define (du-s systeme)
  (if (fichier? (ag-etiquette systeme))
      (taille (ag-etiquette systeme))
      (reduce + 0 (map du-s (ag-foret systeme)))))
```

7.4. Définition de la fonction `ll`

Nous voudrions définir la fonction `ll` qui correspond à la commande `ls -l` d'Unix :

```
(ll  (syst1)) →
"
0       rab/
0       rgh/
66      fbc
"

;;; ll : Systeme -> Paragraphe
;;; (ll systeme) rend le paragraphe contenant, pour chaque élément de «systeme»
;;; (on ne considère que les sous-éléments immédiats), une ligne formée:
;;; pour un fichier, de sa taille et de son nom (séparés par une tabulation),
;;; pour un répertoire, de 0, d'une tabulation, de son nom immédiatement suivi
;;; du caractère "/"
```

Pour calculer le résultat de cette fonction, on peut :

1. extraire la liste des descripteurs de tous les sous-arbres du système (en utilisant la fonctionnelle `map` appliquée à la forêt des sous-répertoires et à la fonction `ag-etiquette`),
2. fabriquer la liste des lignes du paragraphe recherché (toujours à l'aide de la fonctionnelle `map`, appliquée à la liste des descripteurs obtenue dans le point précédent et à une fonction – à définir – qui rend la ligne pour un descripteur donné),
3. et il n'y a plus qu'à fabriquer le paragraphe de toutes ces lignes.

Voici la définition Scheme correspondante :

```
(define (ll systeme)
  ;; desc-ll : Descripteur -> Ligne
  ;; (desc-ll descripteur) rend l'image de «descripteur»: ligne formée:
  ;; pour un fichier, de sa taille et de son nom (séparés par une tabulation),
  ;; pour un répertoire, de 0, d'une tabulation, de son nom immédiatement suivi
  ;; du caractère "/"
  (define (desc-ll descripteur)
    (string-append
     (->string (if (fichier? descripteur)
                   (taille descripteur)
                   0))
     (string #\tab)
     (nom descripteur)
     (if (fichier? descripteur) "" "/")))
  ;; expression de (ll systeme) :
  (paragraphe (map desc-ll
                   (map ag-etiquette (ag-foret systeme)))))
```

7.3. Définition de la fonction `find`

Nous voudrions définir la fonction `find` qui correspond à la commande Unix de même nom :

```
;;; find : string * Systeme -> Paragraphe
;;; (find ident systeme) rend le paragraphe dont les lignes sont constituées par les noms
;;; complets des fichiers ou répertoires du système «systeme» dont le nom est «ident».
```

Voici un exemple de résultat de cette fonction :

```
(find "fbc" (syst1)) →
"
./rab/rgh/fbc
./rab/fbc
./rcd/fbc
"
```

Comme il faut les noms complets, on doit définir une fonction auxiliaire qui a un argument de plus, une chaîne de caractères qui sera un préfixe des lignes de son résultat. De plus, comme d'habitude, pour les définitions de fonctions ayant un arbre généralisé comme donnée, il faut également définir une fonction qui a une forêt comme donnée. D'autre part, on peut noter que la valeur de l'argument « ident » (le nom à chercher) sera inchangée pour tous les appels récursifs : dans les deux fonctions auxiliaires précédentes, cet argument peut être mis en variable globale. D'où la structure de la définition de la fonction `find` :

```
(define (find ident systeme)
  ;; findAux : string * Systeme -> Paragraphe
  ;; (findAux path systeme) rend le paragraphe dont les lignes sont constituées par les
  ;; noms complets des fichiers ou répertoires du système «systeme» dont le nom est
  ;; «ident», chaque ligne étant préfixée par «path».
  ... à faire
  ... à faire
  ;; find-foret : string * Foret[Descripteur] -> Paragraphe
  ;; (find-foret path F) rend le paragraphe obtenu en concaténant, pour tout arbre «A»
  ;; de la fôret «F», (findAux path A)
  ... à faire
  ... à faire
  ;; expression de (find ident systeme) :
  (findAux "" systeme))
```

Définition de la fonction `findAux`

Le résultat de l'application de `findAux` est composé

- d'une ligne comportant le nom complet du système donné lorsque le nom de ce système est le nom recherché,
- du résultat de la recherche sur la forêt du contenu du système donné lorsque ce système est un répertoire.

D'où la définition :

```
(define (find ident systeme)
  ;; findAux : string * Systeme -> Paragraphe
  ;; (findAux path systeme) rend le paragraphe dont les lignes sont constituées par les
  ;; noms complets des fichiers ou répertoires du système «systeme» dont le nom est
  ;; «ident», chaque ligne étant préfixée par «path».
  (define (findAux path systeme)
    (paragraphe-append
     (if (equal? ident (nom (ag-etiquette systeme)))
         (paragraphe (list (string-append path ident)))
         (paragraphe '()))
     (if (repertoire? (ag-etiquette systeme))
         (find-foret (string-append path (nom (ag-etiquette systeme)) "/")
                     (ag-foret systeme))
         (paragraphe '()))))
```

```
  ;; find-foret : String * Foret[Descripteur] -> Paragraphe
  ;; (find-foret path F) rend le paragraphe obtenu en concaténant, pour tout arbre «A»
  ;; de la fôret «F», (findAux path A)
  ...
```

Définition de la fonction `find-foret`

Nous pouvons utiliser les fonctionnelles `map` et `reduce`, `map` permettant de fabriquer la liste des paragraphes résultats de la recherche sur les éléments de la forêt et `reduce` permettant de concaténer ces paragraphes.

- Comme d'habitude, pour concaténer les paragraphes, nous appliquons `reduce` à la fonction `paragraphe-append` et au paragraphe vide.
- Pour fabriquer la liste des paragraphes, nous devons appliquer la fonction `findAux` à chacun des éléments de la forêt, mais avec un autre argument, le préfixe : nous définissons donc une fonction interne qui a ce dernier argument comme variable globale.

D'où la définition :

```
(define (find ident systeme)
  ;; findAux : string * Systeme -> Paragraphe
  ;; (findAux path systeme) rend le paragraphe dont les lignes sont constituées par les
  ;; noms complets des fichiers ou répertoires du système «systeme» dont le nom est
  ;; «ident», chaque ligne étant préfixée par «path».
  ... cf. ci-dessus
  ;; find-foret : string * Foret[Descripteur] -> Paragraphe
  ;; (find-foret path F) rend le paragraphe obtenu en concaténant, pour tout arbre «A»
  ;; de la fôret «F», (findAux path A)
  (define (find-foret path F)
    ; findAux-path : ArbreGeneral[Descripteur] -> Paragraphe
    ; (findAux-path G) rend (findAux path G)
    (define (findAux-path G)
      (findAux path G))
    (reduce paragraphe-append (paragraphe '()) (map findAux-path F)))
  ...
```