**CONCOURS COMMUNS POLYTECHNIQUES**

**EPREUVE SPECIFIQUE - FILIERE MP**

---

# INFORMATIQUE

**Durée : 3 heures**

---

*N.B. : Le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction. Si un candidat est amené à repérer ce qui peut lui sembler être une erreur d'énoncé, il le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'il a été amené à prendre.*

---

**Les calculatrices sont interdites**

PREAMBULE : les trois parties qui composent ce sujet sont indépendantes et peuvent être traitées par les candidats dans un ordre quelconque.

Pour les candidats ayant utilisé le langage CaML dans le cadre des enseignements d'informatique, la partie III (Algorithmique et programmation en CaML) se situe en page 6.

Pour les candidats ayant utilisé le langage PASCAL dans le cadre des enseignements d'informatique, la partie III (Algorithmique et programmation en PASCAL) se situe en page 13.

# Partie I : logique et calcul des propositions

Lors d'une séance d'un jeu informatique dérivé de la mythologie grecque, vous êtes accompagné par un couple de Sphinx. Ces créatures posent des énigmes logiques pour vous aider à progresser dans le jeu. Les énigmes suivent une règle que les Sphinx respectent scrupuleusement. Lorsque les Sphinx énoncent cette règle, ils disent toujours la vérité. Le premier Sphinx qui vous accompagne a énoncé la règle suivante :
« Dans les énigmes, je peux soit dire la vérité, soit mentir. Mais, pour une énigme donnée, la première et la dernière de mes affirmations seront de la même nature (soit vérité, soit mensonge) ; et toutes les autres affirmations seront de la nature opposée à ces deux là (mensonge, respectivement vérité, si les premières et dernières sont des vérités, respectivement des mensonges) ».
Le second Sphinx a énoncé la règle suivante : « Je suivrais la même règle que mon compagnon ».

**Question I.1** *Considérons que l'un des Sphinx fait une suite de $n$ déclarations $A_i$ dans une même énigme, proposer une formule du calcul des propositions qui représente la règle qu'il respecte.*

Vous vous retrouvez face à trois escaliers, l'un à gauche, l'autre à droite et le dernier au milieu entre les deux autres.
Le premier Sphinx $P$ énonce les affirmations suivantes :
– l'escalier de gauche est sûr,
– l'escalier du milieu est sûr ou celui de droite n'est pas sûr.
Le second Sphinx $S$ énonce les affirmations suivantes :
– ni l'escalier de gauche, ni celui du milieu ne sont sûrs,
– si les escaliers de gauche ou de droite sont sûrs, alors l'escalier du milieu est sûr.
Nous noterons $G$, $M$ et $D$ les variables propositionnelles associées au fait que les escaliers de gauche, du milieu et de droite sont sûrs.
Nous noterons $P_1$ et $P_2$, respectivement $S_1$ et $S_2$, les formules propositionnelles associées aux déclarations du premier Sphinx, respectivement du second Sphinx.

**Question I.2** *Représenter les déclarations des deux Sphinx sous la forme de formules du calcul des propositions $P_1$, $P_2$, $S_1$ et $S_2$ dépendant des variables $G$, $M$ et $D$.*

**Question I.3** *Appliquer la règle respectée par les Sphinx que vous avez proposée pour la question I.1. Nous noterons $P$, respectivement $S$, la formule du calcul des propositions dépendant des variables $P_1$ et $P_2$, respectivement $S_1$ et $S_2$, qui correspond au respect de la règle par le premier Sphinx, respectivement le second Sphinx, dans cette énigme. Nous noterons $R$ la formule du calcul des propositions dépendant des variables $P$ et $S$ qui décrit le respect global des règles par les deux Sphinx dans cette énigme.*

**Question I.4** *En utilisant le calcul des propositions (résolution avec les tables de vérité), déterminer quel est (ou quels sont) le (ou les) escalier(s) qui est (ou sont) sûr(s) ? Vous indiquerez explicitement les résultats intermédiaires correspondant aux formules $P_1$, $P_2$, $P$, $S_1$, $S_2$ et $S$.*

Vous vous retrouvez plus tard face à trois portes de couleurs rouge, verte et bleu.
Seul le premier Sphinx s'exprime et énonce les affirmations suivantes :
– la porte rouge n'est pas sûre ou la porte verte est sûre,
– si les portes rouge et verte sont sûres alors la porte bleue n'est pas sûre,

– la porte verte n'est pas sûre mais la porte bleue est sûre.
Nous noterons $P_3$, $P_4$ et $P_5$ les formules propositionnelles associées aux déclarations du premier Sphinx.
Nous noterons $R$, $V$ et $B$ les variables propositionnelles associées au fait que les portes rouge, verte et bleue sont sûres.

**Question I.5** *Représenter les déclarations du Sphinx sous la forme de formules du calcul des propositions $P_3$, $P_4$ et $P_5$ dépendant des variables $R$, $V$ et $B$.*

**Question I.6** *Appliquer la règle respectée par le premier Sphinx que vous avez proposée pour la question I.1. Nous noterons $P$, la formule du calcul des propositions dépendant des variables $P_3$, $P_4$ et $P_5$ qui correspond au respect de la règle par le premier Sphinx dans cette énigme.*

**Question I.7** *En utilisant le calcul des propositions (résolution avec les formules de De Morgan), déterminer quelle est (ou quelles sont) la (ou les) porte(s) qui est (ou sont) sûre(s).*

# Partie II : automates et langages

Le but de cet exercice est l'étude des propriétés des opérations de dérivation à gauche $m^{-1}.\mathcal{A}$ et à droite $\mathcal{A}.m^{-1}$ d'un automate fini $\mathcal{A}$ selon un mot $m$.

## 1 Automate fini complet déterministe

Pour simplifier les preuves, nous nous limiterons au cas des automates finis complets déterministes. Les résultats étudiés s'étendent au cadre des automates finis quelconques.

### 1.1 Représentation d'un automate fini complet déterministe

**Déf. II.1 (Automate fini complet déterministe)** Soit l'alphabet $X$ (un ensemble de symboles), soit $\Lambda$ le symbole représentant le mot vide ($\Lambda \notin X$), soit $X^\star$ l'ensemble contenant $\Lambda$ et les mots composés de séquences de symboles de $X$ (donc $\Lambda \in X^\star$) ; un automate fini complet déterministe sur $X$ est un quintuplet $A = (Q, X, i, T, \delta)$ composé :
– d'un ensemble fini d'états : $Q$ ;
– d'un état initial : $i \in Q$ ;
– d'un ensemble d'états terminaux : $T \subseteq Q$ ;
– d'une fonction totale de transition confondue avec son graphe : $\delta \subseteq Q \times X \mapsto Q$.

Pour une transition $\delta(o, e) = d$ donnée, nous appelons $o$ l'origine de la transition, $e$ l'étiquette de la transition et $d$ la destination de la transition.

### 1.2 Représentation graphique d'un automate

Les automates peuvent être représentés par un schéma suivant les conventions :
– les valeurs de la fonction totale de transition $\delta$ sont représentées par un graphe orienté dont les nœuds sont les états et les arêtes sont les transitions ;
– un état initial est entouré d'un cercle $(i)$ ;
– un état terminal est entouré d'un double cercle $((t))$ ;

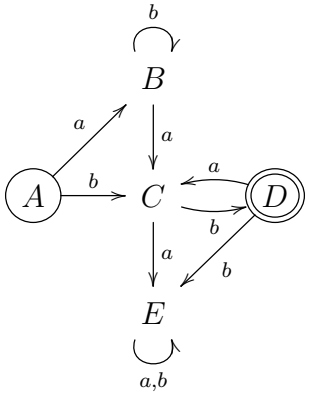
– un état qui est à la fois initial et terminal est entouré d'un triple cercle $\textcircled{\textcircled{\textcircled{it}}}$ ;

– une arête étiquetée par le symbole $e \in X$ va de l'état $o$ à l'état $d$ si et seulement si $\delta(o, e) = d$.

**Exemple II.1** L'automate $\mathcal{E} = (Q, X, i, T, \delta)$ avec :

$$Q = \{A, B, C, D, E\} \quad X = \{a, b\} \quad i = A \quad T = \{D\}$$
$$\delta(A, a) = B \quad \delta(B, a) = C \quad \delta(C, a) = E \quad \delta(D, a) = C \quad \delta(E, a) = E$$
$$\delta(A, b) = C \quad \delta(B, b) = B \quad \delta(C, b) = D \quad \delta(D, b) = E \quad \delta(E, b) = E$$

est représenté par le graphe suivant :




## 1.3 Langage reconnu par un automate fini déterministe

Soit $\delta^\star$ l'extension de $\delta$ à $Q \times X^\star \mapsto Q$ définie par :

$$\forall q \in Q,\ \delta^\star(q, \Lambda) = q$$

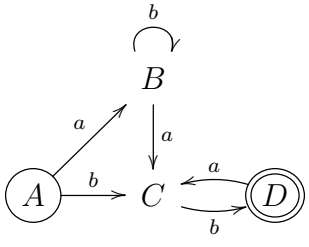
$$\begin{cases} \forall e \in X, \\ \forall m \in X^\star, \\ \forall o \in Q, \\ \forall d \in Q, \end{cases} \quad \delta^\star(o, e.m) = d \Leftrightarrow \exists q \in Q,\ (\delta(o, e) = q) \wedge (\delta^\star(q, m) = d).$$

Soit $X$ un alphabet, un langage sur $X$ est un sous-ensemble de $X^\star$.
Le langage sur $X$ reconnu par un automate fini déterministe est :

$$L(A) = \{m \in X^\star \mid \exists d \in T,\ \delta^\star(i, m) = d\}.$$

Notons que certains états et transitions ne sont pas utiles dans la description d'un langage car ils ne permettent pas d'aller de l'état initial à un état terminal.

**Exemple II.2** Le sous-automate de $\mathcal{E}$ (exemple II.1) composé des états et transitions utiles est représenté par le graphe suivant :



**Question II.1** *Donner, sans la justifier, une expression régulière ou ensembliste représentant le langage sur $X = \{a, b\}$ reconnu par l'automate $\mathcal{E}$ de l'exemple II.1.*

# 2 Opérations de dérivation

## 2.1 Définitions

Soient les opérations internes sur les automates finis complets déterministes définies par :

**Déf. II.2 (Dérivées selon un mot)** Soient $\mathcal{A} = (Q, X, i, T, \delta)$ un automate fini complet déterministe et $m \in X^\star$, les automates $m^{-1}.\mathcal{A}$ (dérivation à gauche selon $m$) et $\mathcal{A}.m^{-1}$ (dérivation à droite selon $m$) sont définis par :

$$m^{-1}.\mathcal{A} = (Q, X, \delta^\star(i, m), T, \delta)$$
$$\mathcal{A}.m^{-1} = (Q, X, i, \{q \in Q \mid \exists t \in T, t = \delta^\star(q, m)\}, \delta).$$

**Question II.2** *En considérant l'exemple II.1, construire les automates $a.b^{-1}.\mathcal{E}$, $b^{-1}.\mathcal{E}$, $\mathcal{E}.a^{-1}$ et $\mathcal{E}.a.b^{-1}$ (seuls les états et les transitions utiles, c'est-à-dire accessibles depuis les états initiaux, devront être construits).*

**Question II.3** *Caractériser les langages reconnus par $a.b^{-1}.\mathcal{E}$, $b^{-1}.\mathcal{E}$, $\mathcal{E}.a^{-1}$ et $\mathcal{E}.a.b^{-1}$, par une expression régulière ou ensembliste.*

## 2.2 Propriétés

**Question II.4** *Montrer que : si $\mathcal{A}$ est un automate fini complet déterministe et si $m \in X^\star$ est un mot, alors $m^{-1}.\mathcal{A}$ et $\mathcal{A}.m^{-1}$ sont des automates finis complets déterministes.*

**Question II.5** *Montrer que :*

$$\forall m \in X^\star,\ \forall n \in X^\star,\ \forall o \in Q,\ \forall q \in Q,\ \forall d \in Q,\ q = \delta^\star(o, m) \wedge d = \delta^\star(q, n) \Leftrightarrow d \in \delta^\star(o, m.n).$$

**Question II.6** *Soit $\mathcal{A}$ un automate fini complet déterministe, montrer que :*

$$\begin{cases} \forall m \in X^\star,\ \forall n \in X^\star,\ n \in L(m^{-1}.\mathcal{A}) \Leftrightarrow m.n \in L(\mathcal{A}) \\ \forall m \in X^\star,\ \forall n \in X^\star,\ m \in L(\mathcal{A}.n^{-1}) \Leftrightarrow m.n \in L(\mathcal{A}). \end{cases}$$

# Partie III : algorithmique et programmation en CaML

Cette partie doit être traitée par les étudiants qui ont utilisé le langage CaML dans le cadre des enseignements d'informatique. Les fonctions écrites devront être récursives ou faire appel à des fonctions auxiliaires récursives. Elles ne devront pas utiliser d'instructions itératives (c'est-à-dire `for`, `while`, ...) ni de références.

## 1 Exercice : le tri rapide

L'objectif de cet exercice est d'étudier une implantation particulière d'un algorithme de tri d'une séquence d'entiers et d'en proposer une évolution pour améliorer ses performances dans le pire cas.

**Déf. III.1** Une séquence $s$ de taille $n$ de valeurs $v_i$ avec $1 \leq i \leq n$ est notée $\langle v_n, \ldots, v_1 \rangle$. Sa taille $n$ est notée $\mid s \mid$. Une même valeur $v$ peut figurer plusieurs fois dans $s$, nous noterons $card(v, s)$ le cardinal de $v$ dans $s$, c'est-à-dire le nombre de fois que $v$ figure dans $s$ avec :

$$card(v, \langle v_n, \ldots, v_1 \rangle) = card\{i \in \mathbb{N} \mid 1 \leq i \leq n, v = v_i\}.$$

Le type `entiers` représente une séquence d'entiers. Le type `triplet` représente un triplet contenant une première séquence d'entiers située à gauche du triplet, puis un entier situé au milieu du triplet et enfin une seconde séquence d'entiers située à droite du triplet. Les définitions de ces types sont :

```
type entiers == int list;;

type triplet == entiers * int * entiers;;
```

Soit le programme en langage CaML :

```
let rec eclater e l =
  match l with
    | [] -> ([], e, [])
    | t::q ->
      let (g, v, d) = (eclater e q) in
      if (t <= v) then
        (t::g, v, d)
      else
        (g, v, t::d);;

let rec trier l =
  match l with
    | [] -> []
    | t::q ->
      let (g, v, d) = (eclater t q) in
      ((trier g)@(v::(trier d)));;
```

Soit la constante `exemple` définie et initialisée par :

```
let exemple = [ 3; 1; 4; 2];;
```

**Question III.1** *Détailler les étapes du calcul de* `(trier exemple)` *en précisant pour chaque appel aux fonctions* `eclater` *et* `trier`*, la valeur du paramètre et du résultat.*

**Déf. III.2 (Symbole de Kronecker)** Soient deux valeurs $v_1$ et $v_2$, le symbole de Kronecker $\delta$ est une fonction $\delta(v_1, v_2)$ égale à la valeur 1 si $v_1$ et $v_2$ sont égales et à la valeur 0 sinon.

**Question III.2** *Soient les entiers* `e` *et* `v`*, soient les séquences d'entiers* `p` $= \langle p_n, \ldots, p_1 \rangle$ *de taille* $n$*,* `g` $= \langle g_r, \ldots, g_1 \rangle$ *de taille* $r$ *et* `d` $= \langle d_s, \ldots, d_1 \rangle$ *de taille* $s$*, telles que* `(g,v,d) = (eclater e p)`*, montrer que :*

*(a)* $\forall i, 1 \leq i \leq n, \delta(p_i, e) + card(p_i, p) = card(p_i, g) + \delta(p_i, v) + card(p_i, d)$

*(b)* $0 \leq s \leq n$

*(c)* $0 \leq r \leq n$

*(d)* $n = s + r$

*(e)* $\forall i, 1 \leq i \leq r, g_i \leq v$

*(f)* $\forall i, 1 \leq i \leq s, v < d_i$

**Question III.3** *Soit la séquence* `p` $= \langle p_m, \ldots, p_1 \rangle$ *de taille* $m$*, soit la séquence* `r` $= \langle r_n, \ldots, r_1 \rangle$ *telle que* `r = (trier p)`*, montrer que :*

*(a)* $m = n$

*(b)* $\forall i, 1 \leq i \leq m, card(p_i, r) = card(p_i, p)$

*(c)* $\forall i, 1 \leq i < n, r_i \leq r_{i+1}$

**Question III.4** *Montrer que le calcul des fonctions* `eclater` *et* `trier` *se termine quelles que soient les valeurs de leurs paramètres respectant le type des fonctions.*

**Question III.5** *Donner des exemples de valeurs du paramètre* `s` *de la fonction* `trier` *qui correspondent aux meilleur et pire cas en nombre d'appels récursifs effectués.*
*Montrer que la complexité dans les meilleur et pire cas de la fonction* `trier` *en fonction du nombre de valeurs dans les séquences données en paramètre sont respectivement de* $O(\mid \texttt{s} \mid \times ln(\mid \texttt{s} \mid))$ *et* $O(\mid \texttt{s} \mid^2)$*. Cette estimation ne prend en compte que le nombre d'appels récursifs effectués.*

La complexité dans le pire cas de la fonction `trier` peut être ramenée à celle du meilleur cas par une modification simple de la fonction `eclater`.

**Question III.6** *Ecrire une nouvelle version de la fonction* `eclater` *telle que la complexité dans le pire cas de la fonction* `trier` *soit égale à sa complexité dans le meilleur cas.*

**Question III.7** *Prouver que cette nouvelle version de la fonction* `eclater` *satisfait les propriétés énoncées dans la question III.2.*

## 2 Problème : représentation de dictionnaire avec des arbres Patricia

L'utilisation de dictionnaires joue un rôle prépondérant dans l'analyse automatique d'informations en langage naturel. Il est donc essentiel de pouvoir exploiter ceux-ci efficacement dans des programmes, c'est-à-dire de pouvoir tester rapidement si un mot fait partie d'un dictionnaire, tout en minimisant l'occupation en mémoire de la représentation informatique du dictionnaire.
L'objectif de ce problème est l'étude des arbres Patricia (*Practical Algorithm To Retrieve Information Coded In Alphanumeric* ce qui signifie *algorithme pratique pour retrouver des informations codées par des caractères alphanumériques*) introduit par D. Morisson et G. Gwehenberger pour réduire la taille des arbres lexicographiques qui permettent un test d'appartenance rapide.

## 2.1 Préliminaires : mots et séquence de mots

Un dictionnaire contient une séquence de mots composés de caractères.

**Déf. III.3 (Mot)** Un mot $m$ est une séquence de taille $n$ de caractères $c_i$ avec $1 \leq i \leq n$. Il est noté $\langle c_n, \ldots, c_1 \rangle$. Sa taille $n$ est notée $\mid m \mid$.

Un mot est représenté par le type `mot` dont la définition est :

```
type mot == char list;;
```

Nous supposons prédéfinie la fonction `length : mot -> int` telle que l'appel `(length m)` sur un mot `m` renvoie la valeur entière $\mid m \mid$. Son calcul se termine quelle que soit la valeur de son paramètre.

**Exemple III.1** L'expression suivante :

```
[ 'f'; 'a'; 'c'; 'e' ]
```

est alors associée au mot *face*.

**Déf. III.4 (Séquence de mots)** Une séquence $s$ de taille $n$ de mots $m_i$ avec $1 \leq i \leq n$ est notée $\langle m_n, \ldots, m_1 \rangle$. Sa taille $n$ est notée $\mid s \mid$.

Une séquence de mots est représentée par le type `mots` dont la définition est :

```
type mots == mot list;;
```

Nous supposons prédéfinie la fonction `length : mots -> int` telle que l'appel `(length s)` sur la séquence `s` renvoie la valeur entière $\mid$ `s` $\mid$. Son calcul se termine quelle que soit la valeur de son paramètre.

**Exemple III.2** L'expression suivante :

```
[ [ 'a' ; 's'] ; [ 'f' ; 'a' ; 'c' ; 'e' ] ; [ 'l' ; 'a'] ]
```

est alors associée à la séquence de mots $\langle as, face, la \rangle$.

### 2.1.1 Distinction de deux mots

La structure des arbres Patricia impose de distinguer et classer les mots par leur premier caractère.

**Question III.8** *Ecrire en CaML une fonction `distinguer` de type `mot -> mot -> bool` telle que l'appel `(distinguer m1 m2)` sur deux mots `m1` et `m2` renvoie la valeur `true` si `m1` et `m2` ne sont pas vides et si le premier caractère de `m1` est strictement plus petit que le premier caractère de `m2` et la valeur `false` sinon.*

### 2.1.2 Ajout d'un préfixe dans une séquence de mots

L'extraction de l'ensemble des mots représentés par un arbre Patricia repose sur l'ajout d'un préfixe à l'ensemble des mots représentés par les sous-arbres de chaque noeud.

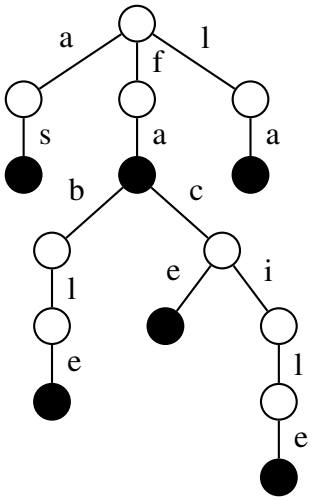
**Question III.9** *Ecrire en CaML une fonction* `prefixer` *de type* `mot -> mots -> mots` *telle que l'appel* `(prefixer m s)` *sur un mot* `m` *et une séquence de mots* `s` *renvoie une séquence de mots contenant les mêmes mots que* `s` *préfixés par le mot* `m`*. L'algorithme utilisé ne devra parcourir qu'une seule fois la séquence* `s`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

**Question III.10** *Calculer une estimation de la complexité de la fonction* `prefixer` *en fonction du nombre d'éléments de la séquence de mots* `s`*. Cette estimation ne prendra en compte que le nombre d'appels récursifs effectués.*

## 2.2 Arbres lexicographiques et Arbres Patricia

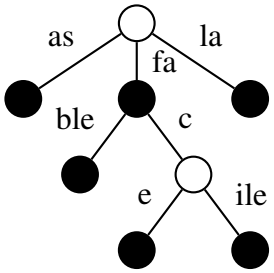
Les arbres lexicographiques sont des arbres $n$-aires utilisés pour représenter des dictionnaires et, plus généralement, des ensembles de mots. Les liens entre deux noeuds consécutifs dans l'arbre, c'est-à-dire entre un noeud père et un noeud fils, sont étiquetés par un caractère. La séquence des caractères qui étiquettent les liens le long d'un chemin de la racine de l'arbre jusqu'à un noeud forme donc un mot. Un noeud peut être terminal si le mot qui conduit de la racine à ce noeud est un mot du dictionnaire, ou non-terminal si ce mot n'est qu'un préfixe d'un mot du dictionnaire.

**Exemple III.3 (Arbre lexicographique)** Voici un exemple d'arbre lexicographique représentant le dictionnaire contenant les mots : *as*, *fa*, *fable*, *facile* et *la*. Les noeuds terminaux, respectivement non terminaux, sont de couleur noire, respectivement blanche :




Dans les arbres Patricia, les liens entre noeuds sont étiquetés par des mots au lieu de caractères. Les chemins entre deux noeuds ne contenant aucun branchement sont alors remplacés par un lien unique étiqueté par le mot composé des caractères étiquetant le chemin complet dans l'arbre lexicographique contenant également ce mot.

**Exemple III.4 (Arbre Patricia)** Voici un exemple d'arbre Patricia représentant le même dictionnaire que l'exemple III.3. Les noeuds terminaux, respectivement non terminaux, sont de couleur noire, respectivement blanche :




**Déf. III.5 (Arbre Patricia)** Un arbre Patricia $a$ est composé de noeuds qui peuvent être terminaux ou non-terminaux. L'ensemble des noeuds de l'arbre Patricia $a$ est noté $\mathcal{N}(a)$, l'ensemble des noeuds terminaux est noté $\mathcal{T}(a)$. La racine de l'arbre Patricia $a$ est un noeud. Chaque noeud $n \in \mathcal{N}(a)$ est composé d'une séquence éventuellement vide de $p$ fils $f_i \in \mathcal{N}(a)$ notée $\mathcal{F}(n) = \langle f_p, \ldots, f_1 \rangle$ avec $p \in \mathbb{N}$ et d'une séquence éventuellement vide de $p$ étiquettes associées $m_i$ notée $\mathcal{E}(n) = \langle m_p, \ldots, m_1 \rangle$ qui sont des mots. Un arbre Patricia est valide si et seulement si :
– un noeud $n$ qui ne possède pas de fils est terminal ;
– le noeud $a$ situé à la racine de l'arbre $a$ est non terminal sauf s'il ne possède aucun fils ;
– les étiquettes de chaque noeud doivent être distinctes et ordonnées de manière croissante selon la relation définie dans la question III.8.

**Question III.11** *Exprimer le fait qu'un arbre Patricia $a$ est valide sous la forme d'une propriété $VAP(a)$. Nous noterons $<$ la relation définie dans la question III.8.*

Un arbre Patricia est représenté par le type `patricia` et le type auxiliaire `fils`. Le type `fils` représente la séquence des fils et étiquettes associées d'un noeud, c'est-à-dire une liste de paires contenant un élément de type `mot` et un élément de type `patricia`. Les définitions en CaML de ces types sont :

```
type patricia = Noeud of bool * fils
and fils == (mot * patricia) list;;
```

Dans l'appel `Noeud(fin, fils)`, les paramètres `fin` et `fils` sont respectivement un drapeau booléen qui indique si le noeud est terminal, ou non terminal et la liste des fils et étiquettes associées de ce noeud de l'arbre. Chaque élément de cette liste est une paire contenant d'une part le mot qui étiquette le lien entre le noeud et son fils, et d'autre part le fils.

**Exemple III.5** L'expression suivante :

```
let mot_as = [`a`; `s`] in
let mot_fa = [`f`; `a`] in
let mot_ble = [`b`; `l`; `e`] in
let mot_c = [`c`] in
let mot_e = [`e`] in
let mot_ile = [`i`; `l`; `e`] in
let mot_la = [`l`; `a`] in

Noeud(false,
  [
    ( mot_as, Noeud(true, []))
  ;
    ( mot_fa,
      Noeud(true,
        [
          ( mot_ble, Noeud(true, []))
        ;
          ( mot_c,
            Noeud(false,
              [
                ( mot_e, Noeud(true, []))
              ;
                ( mot_ile, Noeud(true, []))
              ]))
        ]))
  ;
    ( mot_la, Noeud(true, []))
  ])
```

est alors associée à l'arbre Patricia représenté graphiquement dans l'exemple III.4 (page 10).

### 2.2.1 Création d'un arbre Patricia

**Question III.12** *Ecrire en CaML une fonction* `creer` *de type* `mot -> patricia` *telle que l'appel* `(creer m)` *sur un mot* `m` *renvoie un arbre Patricia valide contenant uniquement le mot* `m`*.*

### 2.2.2 Calcul du nombre de mots contenus dans un arbre Patricia

**Question III.13** *Ecrire en CaML une fonction* `compter` *de type* `patricia -> int` *telle que l'appel* `(compter a)` *sur un arbre Patricia valide* `a` *renvoie le nombre de mots contenus dans l'arbre* `a`*. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre* `a`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

### 2.2.3 Extraction des mots contenus dans un arbre Patricia

**Question III.14** *Ecrire en CaML une fonction* `extraire` *de type* `patricia -> mots` *telle que l'appel* `(extraire a)` *sur un arbre Patricia valide* `a` *renvoie une séquence de mots contenant les mêmes mots que l'arbre* `a`*. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre* `a`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

### 2.2.4 Validation d'un arbre Patricia

**Question III.15** *Ecrire en CaML une fonction* `valider` *de type* `patricia -> bool` *telle que l'appel* `(valider a)` *sur un arbre Patricia* `a` *renvoie la valeur* `true` *si l'arbre Patricia* `a` *est valide, c'est-à-dire si* $VAP(\mathtt{a})$ *et la valeur* `false` *sinon. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre* `a`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

### 2.2.5 Test d'appartenance d'un mot à un arbre Patricia

**Question III.16** *Ecrire en CaML une fonction* `accepter` *de type* `mot -> patricia -> bool` *telle que l'appel* `(accepter m a)` *sur un mot* `m` *et un arbre Patricia valide* `a` *renvoie la valeur* `true` *si l'arbre* `a` *contient le mot* `m` *et la valeur* `false` *sinon. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre* `a`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

### 2.2.6 Ajout d'un mot dans un arbre Patricia

**Question III.17** *Détailler les étapes de l'ajout des mots* facteur, lampe *et* lame *dans l'arbre Patricia de l'exemple III.4 en précisant toutes les étapes et les arbres intermédiaires.*

**Question III.18** *Ecrire en CaML une fonction* `ajouter` *de type* `mot -> patricia -> patricia` *telle que l'appel* `(ajouter m a)` *sur un mot* `m` *et un arbre Patricia valide* `a` *renvoie un arbre Patricia valide contenant le mot* `m` *et les mêmes mots que l'arbre* `a`*. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre* `a`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

### 2.2.7 Fusion de deux arbres Patricia

**Question III.19** *Ecrire en CaML une fonction* `fusionner` *de type* `patricia -> patricia -> patricia` *telle que l'appel* `(fusionner a1 a2)` *sur deux arbres Patricia valides* `a1` *et* `a2` *renvoie un arbre Patricia valide contenant exactement les mots contenus dans les arbres* `a1` *et* `a2`*. L'algorithme utilisé ne devra parcourir qu'une seule fois les arbres* `a1` *et* `a2`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

# Partie III : algorithmique et programmation en PASCAL

Cette partie doit être traitée par les étudiants qui ont utilisé le langage PASCAL dans le cadre des enseignements d'informatique. Les fonctions écrites devront être récursives ou faire appel à des fonctions auxiliaires récursives. Elles ne devront pas utiliser d'instructions itératives (c'est-à-dire `for`, `while`, `repeat`, ...).

## 1 Exercice : le tri rapide

L'objectif de cet exercice est d'étudier une implantation particulière d'un algorithme de tri d'une séquence d'entiers et d'en proposer une évolution pour améliorer ses performances dans le pire cas.

**Déf. III.1** Une séquence $s$ de taille $n$ de valeurs $v_i$ avec $1 \leq i \leq n$ est notée $\langle v_n, \ldots, v_1 \rangle$. Sa taille $n$ est notée $\mid s \mid$. Une même valeur $v$ peut figurer plusieurs fois dans $s$, nous noterons $card(v, s)$ le cardinal de $v$ dans $s$, c'est-à-dire le nombre de fois que $v$ figure dans $s$ avec :

$$card(v, \langle v_n, \ldots, v_1 \rangle) = card\{i \in \mathbb{N} \mid 1 \leq i \leq n, v = v_i\}.$$

Le type `ENTIERS` représente une séquence d'entiers. Nous supposons prédéfinies les constantes et les fonctions suivantes dont le calcul se termine quelles que soient les valeurs de leurs paramètres. Elles pourront éventuellement être utilisées dans les réponses aux questions :

- `Vide` est une constante de valeur `NIL` qui représente une séquence vide d'entiers ;
- `FUNCTION E_Inserer(e:INTEGER;s:ENTIERS):ENTIERS;` renvoie une séquence d'entiers composée d'un premier entier `e` et du reste de la séquence contenu dans `s` ;
- `FUNCTION E_Tete(s:ENTIERS):INTEGER;` renvoie le premier entier de la séquence `s`. Cette séquence ne doit pas être vide ;
- `FUNCTION E_Queue(s:ENTIERS):ENTIERS;` renvoie le reste de la séquence `s` privée de son premier entier. Cette séquence ne doit pas être vide ;
- `FUNCTION E_Juxtaposer(s1,s2:ENTIERS):ENTIERS;` renvoie une séquence composée des éléments de la séquence `s1` dans le même ordre que dans `s1` puis des éléments de la séquence `s2` dans le même ordre que dans `s2`.

Le type `TRIPLET` représente un triplet contenant une première séquence d'entiers située à gauche du triplet puis un entier situé au milieu du triplet et enfin une seconde séquence d'entiers située à droite du triplet. Nous supposons prédéfinies les constantes et les fonctions suivantes dont le calcul se termine quelles que soient les valeurs de leurs paramètres. Elles pourront éventuellement être utilisées dans les réponses aux questions :

- `FUNCTION T_Creer(g:ENTIERS;v:INTEGER;d:ENTIERS):TRIPLET;` renvoie un triplet contenant la séquence d'entiers `g`, puis la valeur entière `v` et enfin la séquence d'entiers `d` ;
- `FUNCTION T_Gauche(t:TRIPLET):ENTIERS;` renvoie la séquence d'entiers située à gauche du triplet `t` ;
- `FUNCTION T_Valeur(t:TRIPLET):INTEGER;` renvoie la valeur entière située au milieu du triplet `t` ;

– `FUNCTION T_Droite(t:TRIPLET):ENTIERS;` renvoie la séquence d'entiers située à droite du triplet `t`.

Soit le programme en langage PASCAL :

```
FUNCTION eclater(e:INTEGER; s:ENTIERS):TRIPLET;
VAR t, v : INTEGER; g, d : ENTIERS;
BEGIN
  IF (s = Vide) THEN
    eclater := T_Creer( Vide, e, Vide)
  ELSE BEGIN
    t := E_Tete( s);
    q := E_Queue( s);
    r := eclater( e, q);
    g := T_Gauche( r);
    d := T_Droite( r);
    v := Valeur( r);
    IF (t <= v) THEN
      eclater := T_Creer( E_Inserer( t, g), v, d)
    ELSE
      eclater := T_Creer( g, v, E_Inserer( t, d))
  END
END;

FUNCTION trier(s:ENTIERS):ENTIERS;
VAR q, g, rg, d, rd : ENTIERS; t, v : INTEGER; r : TRIPLET;
BEGIN
  IF (s = Vide) THEN
    trier := Vide
  ELSE BEGIN
    t := E_Tete( s);
    q := E_Queue( s);
    r := eclater( t, q);
    g := T_Gauche( r);
    v := T_Valeur( r);
    d := T_Droite( r);
    rg := trier( g);
    rd := trier( d);
    trier := E_Juxtaposer( rg, E_Inserer( v, rd))
  END
END;
```

Soit la constante `exemple` définie et initialisée par :

```
CONST exemple : ENTIERS
   = E_Inserer( 3,  E_Inserer( 1,
                        E_Inserer( 4, E_Inserer(2, Vide))));
```

**Question III.1** *Détailler les étapes du calcul de `trier(exemple)` en précisant pour chaque appel aux fonctions `eclater` et `trier`, la valeur du paramètre et du résultat.*

**Déf. III.2 (Symbole de Kronecker)** Soient deux valeurs $v_1$ et $v_2$, le symbole de Kronecker $\delta$ est une fonction $\delta(v_1, v_2)$ égale à la valeur 1 si $v_1$ et $v_2$ sont égales et à la valeur 0 sinon.

**Question III.2** *Soient les entiers* `e` *et* `v`*, la séquence d'entiers* `p` $= \langle p_n, \ldots, p_1 \rangle$ *de taille* $n$*, soient les séquences d'entiers* `g` $= \langle g_r, \ldots, g_1 \rangle$ *de taille* $r$ *et* `d` $= \langle d_s, \ldots, d_1 \rangle$ *de taille* $s$*, soit le triplet* `t`*, tel que* `t = eclater(e,p), g = T_Gauche(t), v = T_Valeur(t)` *et* `d = T_Droite(t)`*, montrer que :*

*(a)* $\forall i, 1 \leq i \leq n, \delta(p_i, e) + card(p_i, p) = card(p_i, g) + \delta(p_i, v) + card(p_i, d)$

*(b)* $0 \leq s \leq n$

*(c)* $0 \leq r \leq n$

*(d)* $n = s + r$

*(e)* $\forall i, 1 \leq i \leq r, g_i \leq v$

*(f)* $\forall i, 1 \leq i \leq s, v < d_i$

**Question III.3** *Soit la séquence* `p` $= \langle p_m, \ldots, p_1 \rangle$ *de taille* $m$*, soit la séquence* `r` $= \langle r_n, \ldots, r_1 \rangle$ *telle que* `r = trier(p)`*, montrer que :*

*(a)* $m = n$

*(b)* $\forall i, 1 \leq i \leq m, card(p_i, r) = card(p_i, p)$

*(c)* $\forall i, 1 \leq i < n, r_i \leq r_{i+1}$

**Question III.4** *Montrer que le calcul des fonctions* `eclater` *et* `trier` *se termine quelles que soient les valeurs de leurs paramètres respectant le type des fonctions.*

**Question III.5** *Donner des exemples de valeurs du paramètre* `s` *de la fonction* `trier` *qui correspondent aux meilleur et pire cas en nombre d'appels récursifs effectués.*
*Montrer que la complexité dans les meilleur et pire cas de la fonction* `trier` *en fonction du nombre de valeurs dans les séquences données en paramètre sont respectivement de* $O(\mid \texttt{s} \mid \times ln(\mid \texttt{s} \mid))$ *et* $O(\mid \texttt{s} \mid^2)$*. Cette estimation ne prend en compte que le nombre d'appels récursifs effectués.*

La complexité dans le pire cas de la fonction `trier` peut être ramenée à celle du meilleur cas par une modification simple de la fonction `eclater`.

**Question III.6** *Ecrire une nouvelle version de la fonction* `eclater` *telle que la complexité dans le pire cas de la fonction* `trier` *soit égale à sa complexité dans le meilleur cas.*

**Question III.7** *Prouver que cette nouvelle version de la fonction* `eclater` *satisfait les propriétés énoncées dans la question III.2.*

## 2 Problème : représentation de dictionnaire avec des arbres Patricia

L'utilisation de dictionnaires joue un rôle prépondérant dans l'analyse automatique d'informations en langage naturel. Il est donc essentiel de pouvoir exploiter ceux-ci efficacement dans des programmes, c'est-à-dire de pouvoir tester rapidement si un mot fait partie d'un dictionnaire, tout en minimisant l'occupation en mémoire de la représentation informatique du dictionnaire.
L'objectif de ce problème est l'étude des arbres Patricia (*Practical Algorithm To Retrieve Information Coded In Alphanumeric* ce qui signifie *algorithme pratique pour retrouver des informations codées par des caractères alphanumériques*) introduit par D. Morisson et G. Gwehenberger pour réduire la taille des arbres lexicographiques qui permettent un test d'appartenance rapide.

## 2.1 Préliminaires : mots et séquence de mots

Un dictionnaire contient une séquence de mots composés de caractères.

**Déf. III.3 (Mot)** Un mot $m$ est une séquence de taille $n$ de caractères $c_i$ avec $1 \leq i \leq n$. Il est noté $\langle c_n, \ldots, c_1 \rangle$. Sa taille $n$ est notée $\mid m \mid$.

Un mot est représenté par le type `MOT`. Nous supposons prédéfinies les constantes et les fonctions suivantes dont le calcul se termine quelles que soient les valeurs de leurs paramètres. Elles pourront éventuellement être utilisées dans les réponses aux questions :
- `Vide` est une constante de valeur `NIL` qui représente un mot vide ;
- `FUNCTION M_Inserer(c:CHARACTER;m:MOT):MOT;` renvoie un mot composé d'un premier caractère `c` suivi des caractères du mot `m` dans le même ordre que dans `m` ;
- `FUNCTION M_Tete(m:MOT):CHARACTER;` renvoie le premier caractère du mot `m`. Ce mot ne doit pas être vide ;
- `FUNCTION M_Queue(m:MOT):MOT;` renvoie le suffixe du mot `m` privé de son premier caractère. Ce mot ne doit pas être vide ;
- `FUNCTION M_Juxtaposer(m1,m2:MOT):MOT;` renvoie un mot composé des caractères du mot `m1` dans le même ordre que dans `m1` puis des caractères du mot `m2` dans le même ordre que dans `m2` ;
- `FUNCTION M_Taille(m:MOT):INTEGER;` renvoie la valeur entière $|\texttt{m}|$.

**Exemple III.1** L'expression suivante :

```
M_Inserer( 'f', M_Inserer( 'a',
  M_Inserer( 'c', M_Inserer( 'e', Vide))))
```

est alors associée au mot *face*.

**Déf. III.4 (Séquence de mots)** Une séquence $s$ de taille $n$ de mots $m_i$ avec $1 \leq i \leq n$ est notée $\langle m_n, \ldots, m_1 \rangle$. Sa taille $n$ est notée $\mid s \mid$.

Une séquence de mots est représentée par le type `MOTS`. Nous supposons prédéfinies les constantes et les fonctions suivantes dont le calcul se termine quelles que soient les valeurs de leurs paramètres. Elles pourront éventuellement être utilisées dans les réponses aux questions :
- `Vide` est une constante de valeur `NIL` qui représente une séquence de mots vide ;
- `FUNCTION S_Inserer(m:MOT;s:MOTS):MOTS;` renvoie une séquence composée d'un premier mot `m` suivi des mots de la séquence de mots `s` dans le même ordre que dans `s` ;
- `FUNCTION S_Tete(s:MOTS):MOT;` renvoie le premier mot de la séquence de mots `s`. Cette séquence ne doit pas être vide ;
- `FUNCTION S_Queue(s:MOTS):MOTS;` renvoie le reste de la séquence de mots `s` privée de son premier mot. Cette séquence ne doit pas être vide ;
- `FUNCTION S_Juxtaposer(s1,s2:MOTS):MOTS;` renvoie une séquence de mots composée des mots de la séquence de mots `s1` dans le même ordre que dans `s1` puis des mots de la séquence de mots `s2` dans le même ordre que dans `s2` ;
- `FUNCTION S_Taille(s:MOTS):INTEGER;` renvoie la valeur entière $\mid \texttt{s} \mid$.

**Exemple III.2** L'expression suivante :

```
S_Inserer(
  M_Inserer( 'a', M_Inserer( 's', Vide)),
  S_Inserer(
    M_Inserer( 'f', M_Inserer( 'a',
      M_Inserer( 'c', M_Inserer( 'e', Vide)))),
    S_Inserer(
      M_Inserer( 'l', M_Inserer( 'a', Vide))),
      Vide))
```

est alors associée à la séquence de mots $\langle as, face, la \rangle$.

### 2.1.1 Distinction de deux mots

La structure des arbres Patricia impose de distinguer et classer les mots par leur premier caractère.

**Question III.8** *Ecrire en PASCAL une fonction* `distinguer(m1,m2:MOT):BOOLEAN;` *telle que l'appel* `distinguer(m1,m2)` *sur deux mots* `m1` *et* `m2` *renvoie la valeur* `true` *si* `m1` *et* `m2` *ne sont pas vides et si le premier caractère de* `m1` *est strictement plus petit que le premier caractère de* `m2` *et la valeur* `false` *sinon.*

### 2.1.2 Ajout d'un préfixe dans une séquence de mots

L'extraction de l'ensemble des mots représentés par un arbre Patricia repose sur l'ajout d'un préfixe à l'ensemble des mots représentés par les sous-arbres de chaque noeud.

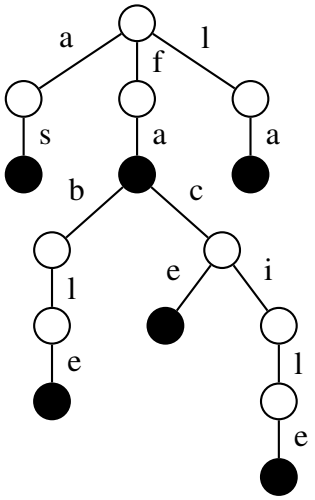
**Question III.9** *Ecrire en PASCAL une fonction* `prefixer(m:MOT;s:MOTS):MOTS;` *telle que l'appel* `prefixer(m,s)` *sur un mot* `m` *et une séquence de mots* `s` *renvoie une séquence de mots contenant les mêmes mots que* `s` *préfixés par le mot* `m`*. L'algorithme utilisé ne devra parcourir qu'une seule fois la séquence* `s`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

**Question III.10** *Calculer une estimation de la complexité de la fonction* `prefixer` *en fonction du nombre d'éléments de la séquence de mots* `s`*. Cette estimation ne prendra en compte que le nombre d'appels récursifs effectués.*

## 2.2 Arbres lexicographiques et Arbres Patricia

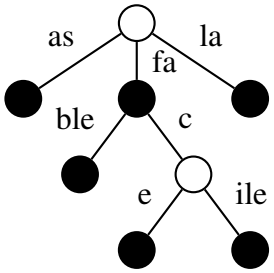
Les arbres lexicographiques sont des arbres $n$-aires utilisés pour représenter des dictionnaires et, plus généralement, des ensembles de mots. Les liens entre deux noeuds consécutifs dans l'arbre, c'est-à-dire entre un noeud père et un noeud fils, sont étiquetés par un caractère. La séquence des caractères qui étiquettent les liens le long d'un chemin de la racine de l'arbre jusqu'à un noeud forme donc un mot. Un noeud peut être terminal si le mot qui conduit de la racine à ce noeud est un mot du dictionnaire, ou non-terminal si ce mot n'est qu'un préfixe d'un mot du dictionnaire.

**Exemple III.3 (Arbre lexicographique)** Voici un exemple d'arbre lexicographique représentant le dictionnaire contenant les mots : *as*, *fa*, *fable*, *facile* et *la*. Les noeuds terminaux, respectivement non terminaux, sont de couleur noire, respectivement blanche :




Dans les arbres Patricia, les liens entre noeuds sont étiquetés par des mots au lieu de caractères. Les chemins entre deux noeuds ne contenant aucun branchement sont alors remplacés par un lien unique étiqueté par le mot composé des caractères étiquetant le chemin complet dans l'arbre lexicographique contenant également ce mot.

**Exemple III.4 (Arbre Patricia)** Voici un exemple d'arbre Patricia représentant le même dictionnaire que l'exemple III.3. Les noeuds terminaux, respectivement non terminaux, sont de couleur noire, respectivement blanche :




**Déf. III.5 (Arbre Patricia)** Un arbre Patricia $a$ est composé de noeuds qui peuvent être terminaux ou non-terminaux. L'ensemble des noeuds de l'arbre Patricia $a$ est noté $\mathcal{N}(a)$, l'ensemble des noeuds terminaux est noté $\mathcal{T}(a)$. La racine de l'arbre Patricia $a$ est un noeud. Chaque noeud $n \in \mathcal{N}(a)$ est composé d'une séquence éventuellement vide de $p$ fils $f_i \in \mathcal{N}(a)$ notée $\mathcal{F}(n) = \langle f_p, \ldots, f_1 \rangle$ avec $p \in \mathbb{N}$ et d'une séquence éventuellement vide de $p$ étiquettes associées $m_i$ notée $\mathcal{E}(n) = \langle m_p, \ldots, m_1 \rangle$ qui sont des mots. Un arbre Patricia est valide si et seulement si :
- un noeud $n$ qui ne possède pas de fils est terminal ;
- le noeud $a$ situé à la racine de l'arbre $a$ est non terminal sauf s'il ne possède aucun fils ;
- les étiquettes de chaque noeud doivent être distinctes et ordonnées de manière croissante selon la relation définie dans la question III.8.

**Question III.11** *Exprimer le fait qu'un arbre Patricia $a$ est valide sous la forme d'une propriété $VAP(a)$. Nous noterons $<$ la relation définie dans la question III.8.*

Un arbre Patricia est représenté par le type de base `PATRICIA`. La séquence des fils et étiquettes associées d'un noeud est représentée par le type de base `FILS`. Une paire composée d'un fils et de l'étiquette associée est représentée par le type de base `PAIRE`.
Nous supposons prédéfinies les constantes et les fonctions suivantes dont le calcul se termine quelles que soient les valeurs de leurs paramètres. Elles pourront éventuellement être utilisées dans les réponses aux questions :

- `Vide` est une constante de valeur `NIL` qui représente une séquence vide ;
- `FUNCTION P_Creer(mot:MOT;fils:PATRICIA):PAIRE;` renvoie une paire contenant l'étiquette `mot` et l'arbre Patricia `fils` ;
- `FUNCTION P_Etiquette(p:PAIRE):MOT;` renvoie l'étiquette contenue dans la paire `p` ;
- `FUNCTION P_Arbre(p:PAIRE):PATRICIA;` renvoie l'arbre contenu dans la paire `p` ;
- `FUNCTION F_Inserer(p:PAIRE;s:FILS):FILS;` renvoie une séquence de fils composée d'un premier fils `p` et du reste de la séquence contenu dans `s` ;
- `FUNCTION F_Tete(s:FILS):PAIRE;` renvoie le premier fils de la séquence `s`. Cette séquence ne doit pas être vide ;
- `FUNCTION F_Queue(s:FILS):FILS;` renvoie le reste de la séquence `s` privée du premier fils. Cette séquence ne doit pas être vide ;
- `FUNCTION F_Juxtaposer(s1,s2:FILS):FILS;` renvoie une séquence composée des éléments de la séquence `s1` dans le même ordre que dans `s1` puis des éléments de la séquence `s2` dans le même ordre que dans `s2` ;
- `FUNCTION Noeud(fin:BOOLEAN;fils:FILS):PATRICIA;` est une fonction qui renvoie un arbre Patricia dont le noeud situé à la racine est terminal si `fin` prend la valeur `FALSE` et non terminal sinon et dont les fils et leurs étiquettes associées sont contenus dans la séquence `fils` ;
- `FUNCTION EstTerminal(a:PATRICIA):BOOLEAN;` est une fonction qui renvoie la valeur `TRUE` si le noeud situé à la racine de l'arbre `a` est terminal et la valeur `FALSE` sinon ;
- `FUNCTION Fils(a:PATRICIA):FILS;` est une fonction qui renvoie la séquence des fils et étiquettes associées de la racine de l'arbre Patricia `a`.

**Exemple III.5** Le programme suivant :

```
VAR exemple : PATRICIA;
CONST
  mot_as  : MOT = M_Inserer( 'a', M_Inserer( 's', Vide));
  mot_fa  : MOT = M_Inserer( 'f', M_Inserer( 'a', Vide));
  mot_ble : MOT =
    M_Inserer( 'b', M_Inserer( 'l', M_Inserer( 'e', Vide)));
  mot_c   : MOT = M_Inserer( 'c', Vide);
  mot_e   : MOT = M_Inserer( 'e', Vide);
  mot_ile : MOT =
    M_Inserer( 'i', M_Inserer( 'l', M_Inserer( 'e', Vide)));
  mot_la  : MOT = M_Inserer( 'l', M_Inserer( 'a', Vide));
BEGIN
  exemple := Noeud(false,
    F_Inserer(
      P_Creer( mot_as, Noeud(true, Vide))
    F_Inserer(
      P_Creer( mot_fa,
        Noeud(true,
          F_Inserer(
            P_Creer( mot_ble, Noeud(true, Vide))
          F_Inserer(
            P_Creer( mot_c,
              Noeud(false,
                F_Inserer(
                  P_Creer( mot_e, Noeud(true, Vide)),
                F_Inserer(
                  P_Creer( mot_ile, Noeud(true, Vide)),
                Vide)))),
          Vide)))),
    F_Inserer(
      R_Creer( mot_la, Noeud(true, Vide)),
    Vide))))
END
```

affecte à la variable `exemple` l'arbre Patricia représenté graphiquement dans l'exemple III.4 (page 18).

### 2.2.1 Création d'un arbre Patricia

**Question III.12** *Ecrire en PASCAL une fonction `creer(m:MOT):PATRICIA;` telle que l'appel `creer(m)` sur un mot `m` renvoie un arbre Patricia valide contenant uniquement le mot `m`.*

### 2.2.2 Calcul du nombre de mots contenus dans un arbre Patricia

**Question III.13** *Ecrire en PASCAL une fonction `compter(a:PATRICIA):INTEGER;` telle que l'appel `compter(a)` sur un arbre Patricia valide `a` renvoie le nombre de mots contenus dans l'arbre `a`. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre `a`. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

### 2.2.3 Extraction des mots contenus dans un arbre Patricia

**Question III.14** *Ecrire en PASCAL une fonction* `extraire(a:PATRICIA):MOTS;` *telle que l'appel* `extraire(a)` *sur un arbre Patricia valide* `a` *renvoie une séquence de mots contenant les mêmes mots que l'arbre* `a`*. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre* `a`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

### 2.2.4 Validation d'un arbre Patricia

**Question III.15** *Ecrire en PASCAL une fonction* `valider(a:PATRICIA):BOOLEAN;` *telle que l'appel* `valider(a)` *sur un arbre Patricia* `a` *renvoie la valeur* `TRUE` *si l'arbre Patricia* `a` *est valide, c'est-à-dire si* $VAP(\mathtt{a})$ *et la valeur* `FALSE` *sinon. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre* `a`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

### 2.2.5 Test d'appartenance d'un mot à un arbre Patricia

**Question III.16** *Ecrire en PASCAL une fonction* `accepter(m:MOT;a:PATRICIA):BOOLEAN;` *telle que l'appel* `accepter(m,a)` *sur un mot* `m` *et un arbre Patricia valide* `a` *renvoie la valeur* `TRUE` *si l'arbre* `a` *contient le mot* `m` *et la valeur* `FALSE` *sinon. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre* `a`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

### 2.2.6 Ajout d'un mot dans un arbre Patricia

**Question III.17** *Détailler les étapes de l'ajout des mots* facteur, lampe *et* lame *dans l'arbre Patricia de l'exemple III.4 en précisant toutes les étapes et les arbres intermédiaires.*

**Question III.18** *Ecrire en PASCAL une fonction* `ajouter(m:MOT;a:PATRICIA):PATRICIA;` *telle que l'appel* `ajouter(m,a)` *sur un mot* `m` *et un arbre Patricia valide* `a` *renvoie un arbre Patricia valide contenant le mot* `m` *et les mêmes mots que l'arbre* `a`*. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre* `a`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

### 2.2.7 Fusion de deux arbres Patricia

**Question III.19** *Ecrire en PASCAL une fonction* `fusionner(a1,a2:PATRICIA):PATRICIA;` *telle que l'appel* `fusionner(a1,a2)` *sur deux arbres Patricia valides* `a1` *et* `a2` *renvoie un arbre Patricia valide contenant exactement les mots contenus dans les arbres* `a1` *et* `a2`*. L'algorithme utilisé ne devra parcourir qu'une seule fois les arbres* `a1` *et* `a2`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

**Fin de l'énoncé**