



Les candidats indiqueront en tête de leur copie le langage de programmation choisi (Pascal ou Caml). Les candidats ayant choisi Caml devront donner le type de chaque fonction écrite. Les candidats travaillant en Pascal pourront écrire des fonctions ou des procédures.

## I Arbres de décision

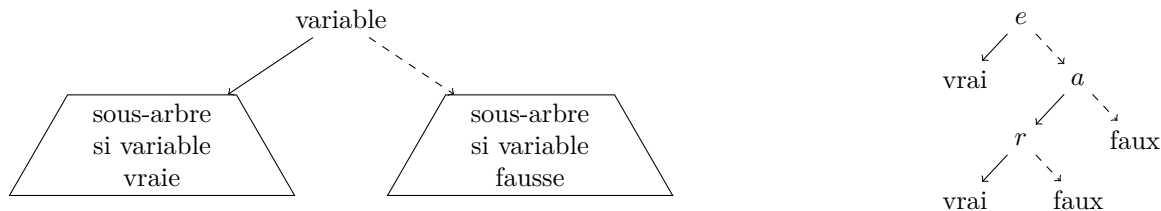
Un arbre de décision est un arbre binaire dans lequel :

- un nœud interne est associé à une variable, parmi un ensemble  $V$  de variables ;
- une feuille est associée à un booléen (vrai ou faux).

Si chaque variable de l'ensemble  $V$  reçoit une valeur booléenne, un tel arbre permet de *prendre une décision* en parcourant l'arbre :

- on part de la racine ;
- quand on arrive sur un nœud interne (racine comprise), on regarde quelle est la valeur de la variable associée au nœud : si elle vaut *vrai* on poursuit le parcours dans le sous-arbre gauche, sinon on poursuit le parcours dans le sous-arbre droit ;
- quand on arrive sur une feuille, le booléen associé constitue la *décision*.

Conventionnellement, on représente l'arête menant au sous-arbre pour le « cas vrai » en trait plein ; l'arête menant au sous-arbre « cas faux » en pointillés. Schématiquement, un arbre est structuré comme indiqué ci-dessous à gauche.



Le schéma de droite ci-dessus illustre l'exemple : un module de cours est validé si l'examen est réussi ( $e$ ), ou sinon, si l'étudiant a été assidu en cours ( $a$ ) et qu'il réussit un examen de rattrapage ( $r$ ). Cela revient à définir la validation du module par la formule logique  $e \vee (a \wedge r)$ .

On envisage une représentation simple d'un arbre de décision, à l'aide d'un tableau. On numérote les nœuds : la racine reçoit le numéro 0, les autres nœuds sont numérotés arbitrairement par des entiers consécutifs à partir de 1. On crée un tableau contenant autant de cases que de nœuds, indicé à partir de 0. La case d'indice  $i$  contient soit un triplet (nom de variable, numéro du fils gauche, numéro du fils droit) si le nœud numéro  $i$  est un nœud interne, soit un booléen si le nœud numéro  $i$  est une feuille.

En Caml, on définit le type :

```
type noeud =  
  Feuille of bool  
  | Decision of string * int * int;;
```

Un arbre de décision est donc représenté par un vecteur de nœuds (type `noeud vect`).

En Pascal :

```
type SorteNoeud = (Feuille, Decision);  
  
type Noeud = record  
  sorte: SorteNoeud;  
  variable: string;           (* Utilisé si sorte = Decision *)  
  g, d: integer;            (* Utilisés si sorte = Decision *)  
  valeurFeuille: boolean;   (* Utilisé si sorte = Feuille *)  
end;
```

En Pascal un arbre de décision contenant  $n$  nœuds sera donc de type `array[0..n-1] of Noeud`.

**I.A** – Définir une variable `monAD` représentant l'arbre de décision illustré précédemment.

Dans les deux questions suivantes, on veut faire déterminer une décision en fournissant une valuation des variables, c'est-à-dire la liste des seules variables qui sont vraies dans l'évaluation.

**I.B** – Définir une fonction `eval_var` qui, étant donné le nom d'une variable (`string`) et une liste (Caml) ou un tableau (Pascal) des seules variables vraies, renvoie un booléen correspondant à la valuation de la variable indiquée.

**I.C** – Définir une fonction `eval` qui, étant donné un arbre de décision et une liste (Caml) ou un tableau (Pascal) des seules variables vraies, renvoie un booléen correspondant à la décision finale.

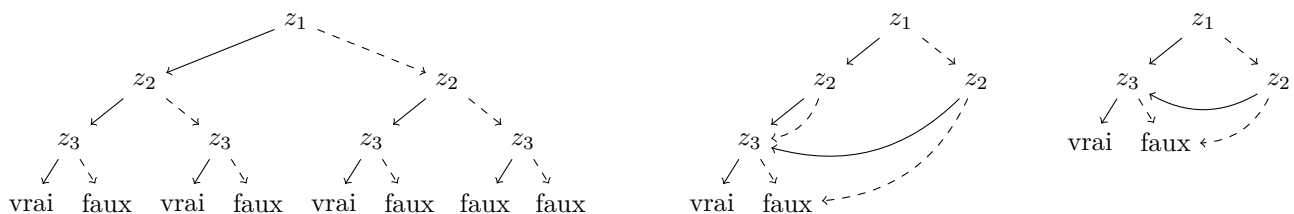
## II Diagrammes de décision

On souhaite compacter la représentation en mémoire des arbres de décision. Si plusieurs sous-arbres sont identiques, on n'a pas envie de les stocker plusieurs fois.

En raisonnant sur la représentation informatique des arbres de décision, on voit assez facilement une façon de procéder : si les arbres de numéros  $i$  et  $j$  sont identiques, on peut (par exemple) au niveau du parent  $p$  de  $j$  indiquer comme numéro de fils  $i$  au lieu de  $j$  et ainsi éliminer  $j$  de la représentation.

Ce faisant, on ne représente plus un arbre (car  $i$  a maintenant deux parents), mais un *graphe orienté* : on parle de *diagrammes de décision*. Néanmoins aucune connaissance particulière en théorie des graphes n'est requise pour aborder ce problème. On dira qu'il existe un *arc* de  $p$  vers  $i$  et on le notera  $p \xrightarrow{b} i$  avec  $b \in \{T, F\}$  (pour *vrai* ou *faux*), selon que l'arc est suivi dans le cas où  $p$  est *vrai* (précédemment : fils gauche) ou dans le cas où  $p$  est *faux* (précédemment : fils droit). Lorsque  $p \xrightarrow{b} i$ , on note  $i = \text{succ}_b(p)$ .

Exemple : l'expression  $(z_1 \wedge z_3) \vee (z_2 \wedge z_3)$  admet (entre autres) les diagrammes de décision ci-dessous.



**II.A** – Créer une fonction (Caml) ou procédure (Pascal) `redirige` à trois paramètres — un diagramme ainsi que deux indices  $v$  et  $w$  — qui supprime le nœud  $v$  dans le graphe et transforme tous les arcs  $u \xrightarrow{b} v$  en arcs  $u \xrightarrow{b} w$ . Les cases du tableau qui deviennent inoccupées sont remplies avec la valeur spéciale `Vide`.

Pour ce faire en Caml on complète :

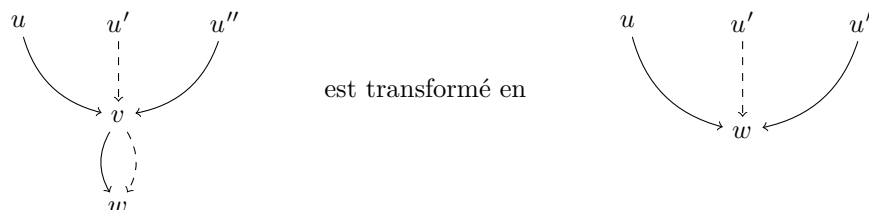
```
type noeud =
  Feuille of bool
  | Decision of string * int * int
  | Vide;;
```

De même en Pascal on ajoute une sorte de nœuds (le type `Nœud` lui-même reste inchangé) :

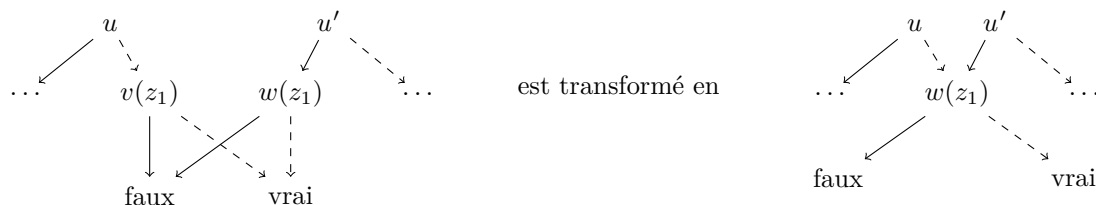
```
type SorteNœud = (Feuille, Decision, Vide);
```

Pour transformer un arbre en diagramme sans répétition, on applique deux règles de simplification.

– *Élimination* : Si pour un nœud  $v$  on a  $\text{succ}_F(v) = \text{succ}_T(v) = w$  alors on élimine  $v$  et on transforme les arcs  $u \xrightarrow{b} v$  en  $u \xrightarrow{b} w$ .



– *Isomorphisme* : Soit  $v$  et  $w$  deux nœuds,  $v \neq w$ . Si ce sont des feuilles avec  $\text{valeur}(v) = \text{valeur}(w)$  ou si ce sont des nœuds internes tels que  $\text{variable}(v) = \text{variable}(w)$  et  $\text{succ}_F(v) = \text{succ}_F(w)$  et  $\text{succ}_T(v) = \text{succ}_T(w)$  alors on élimine  $v$  et on transforme les arcs  $u \xrightarrow{b} v$  en  $u \xrightarrow{b} w$ .



**II.B** – Créer une fonction `trouve_elimination`, prenant en paramètre un diagramme et renvoyant l'indice d'un nœud pouvant être supprimé par élimination, s'il en existe un. Sinon, elle doit renvoyer `-1`.

**II.C** – De même, créer une fonction `trouve_isomorphisme`, prenant en paramètre un diagramme, et renvoyant un couple d'indices correspondant à deux nœuds pouvant être simplifiés par isomorphisme, s'il en existe un. Sinon, elle doit renvoyer le couple  $(-1, -1)$ .

Les candidats qui composent en Caml peuvent directement manipuler des couples. Les candidats qui composent en Pascal créeront une procédure recevant en paramètres deux variables entières transmises par référence :

```
procedure trouve_isomorphisme(diagramme: array of noeud; var i, j: integer);
```

On dit que le diagramme est *sous forme réduite* s'il n'existe pas de nœuds différents qui correspondent à la même formule logique.

**II.D** – Prouver l'assertion suivante :

Un diagramme est sous forme réduite si, et seulement si, ni la règle d'élimination ni la règle d'isomorphisme ne peuvent lui être appliquées.

**II.E** – Créer une fonction sans résultat (Caml) ou une procédure (Pascal) appelée `reduit`, prenant en paramètre un diagramme, qui détecte les deux simplifications possibles, effectue les redirections correspondantes, jusqu'à ce qu'il ne soit possible de faire aucune simplification supplémentaire.

On obtient à ce stade une représentation du diagramme simplifié sous forme d'un tableau dans lequel certaines cases ne sont plus utilisées : elles sont marquées `Vide`.

### III Diagrammes de décision ordonnés

Nous nous intéressons maintenant à la construction d'arbres de décision à partir de formules logiques.

Étant données des formules logiques  $t$ ,  $e_1$  et  $e_2$ , on définit l'opérateur  $t \rightarrow e_1, e_2$  par :

$$t \rightarrow e_1, e_2 = (t \wedge e_1) \vee (\neg t \wedge e_2)$$

**III.A** – Soient  $x$  et  $y$  des formules logiques quelconques. Montrer que les trois formules logiques suivantes

$$\neg x, \quad x \vee y, \quad x \wedge y$$

peuvent s'écrire en utilisant uniquement les constantes 0 (faux), 1 (vrai), l'opérateur  $\cdot \rightarrow \cdot, \cdot$  défini précédemment et les variables  $x$  et  $y$ .

**III.B** – Montrer que  $(a \rightarrow b, c) \rightarrow d, e = a \rightarrow (b \rightarrow d, e), (c \rightarrow d, e)$ .

**III.C** – Dédurre de ce qui précède une méthode systématique de construction d'un arbre de décision à partir d'une formule logique quelconque.

**III.D** – Soient  $t$  une variable booléenne et  $e$  une formule logique. Que vaut  $t \rightarrow e, e$  ?

Un diagramme de décision est dit *ordonné* si, pour un ordre donné entre les variables  $x_1 \prec x_2 \prec \dots \prec x_n$ , alors tout chemin partant de la racine vers les feuilles parcourt les variables dans cet ordre.

La fonction booléenne représentée par un diagramme de décision  $u$  est notée  $f^u$ .

La méthode de construction d'un arbre de décision imaginée en **III.C** ne respecte pas forcément un certain ordre des variables. Dans cette partie nous proposons une autre méthode de construction, un ordre étant donné a priori.

Pour une variable  $t$ , une expression  $e$  et une fonction booléenne  $f$ , on note  $f[t = e]$  la fonction déduite de  $f$  en remplaçant toutes les occurrences de  $t$  par  $e$ .

**III.E** – Que vaut  $t \rightarrow f[t = 1], f[t = 0]$  ?

**III.F** – En déduire une méthode de construction d'un diagramme de décision réduit ordonné à partir d'une fonction booléenne sur un ensemble ordonné de variables.

**III.G** – Montrer que pour toute fonction booléenne  $f$  de  $n$  variables ordonnées  $x_1 \prec x_2 \prec \dots \prec x_n$ , il existe un *unique* diagramme de décision réduit ordonné  $u$  tel que  $f^u = f$ .

**III.H** – À l'aide de ce qui précède, donner une méthode simple permettant de décider de l'égalité entre deux fonctions booléennes portant sur le même ensemble de  $n$  variables.

**III.I** – Comment déterminer facilement si une formule logique est une tautologie ?

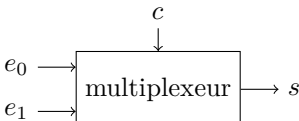
### IV Circuits logiques

On sait que toute fonction booléenne peut se mettre sous *forme normale disjonctive* :  $f$  s'écrit comme une disjonction (un *ou*) de *mintermes*, sachant qu'un minterme est une conjonction (un *et*) entre toutes les variables de  $f$ , chacune d'entre elles pouvant éventuellement être niée.

Par exemple,  $f = z_1 \wedge (z_2 \vee \neg z_3)$  s'écrit en forme normale disjonctive :  $f = (z_1 \wedge \neg z_2 \wedge \neg z_3) \vee (z_1 \wedge z_2 \wedge \neg z_3) \vee (z_1 \wedge z_2 \wedge z_3)$ . C'est une disjonction de trois mintermes.

**IV.A** – Pour une fonction booléenne  $f$ , évaluer le nombre de portes logiques nécessaires à sa réalisation directe à partir de sa forme normale disjonctive, en fonction du nombre de variables de  $f$ .

On appelle *multiplexeur* à deux entrées un circuit logique qui recopie sur sa sortie  $s$  l'une de ses deux entrées,  $e_0$  ou  $e_1$ , en fonction de la valeur (resp. 0 ou 1) d'un signal de commande  $c$ .

$$s = \begin{cases} e_0 & \text{si } c = 0 \\ e_1 & \text{si } c = 1 \end{cases}$$


**IV.B** – Donner la table de vérité du multiplexeur à deux entrées.

**IV.C** – Donner un schéma pour réaliser le multiplexeur à deux entrées à partir de portes logiques élémentaires.

**IV.D** – Donner une méthode simple permettant de déterminer un circuit logique réalisant la fonction booléenne représentée par un diagramme de décision.

## V Automates

On s'intéresse dans cette partie aux combinaisons booléennes d'équations linéaires sur les entiers. Par exemple, avec deux variables, la résolution du système  $(2x_1 - x_2 = -4) \wedge (7x_1 - 3x_2 = 1)$  ou de  $(3x_1 - 5x_2 = 2) \vee (\neg(4x_1 - 3x_2 = 1))$ . Les combinaisons logiques peuvent être traitées par l'arbre de décision précédent. On veut construire des automates permettant de résoudre les équations linéaires. Dans le cas général, pour  $n$  un entier positif, ces équations à  $n$  variables peuvent s'écrire sous la forme  $\langle \vec{a} | \vec{x} \rangle = k$  avec  $\vec{a} = (a_1, \dots, a_n) \in \mathbb{Z}^n$ ,  $k \in \mathbb{Z}$ ,  $\vec{x} = (x_1, \dots, x_n) \in \mathbb{N}^n$ , et  $\langle \cdot | \cdot \rangle$  dénotant le produit scalaire usuel entre deux vecteurs. On note  $E = \{0, 1\}$  et on prend comme alphabet  $A$  l'ensemble  $E^n$ . Un mot

$$x = \begin{pmatrix} x_{1,1} \\ \vdots \\ x_{n,1} \end{pmatrix} \dots \begin{pmatrix} x_{1,m} \\ \vdots \\ x_{n,m} \end{pmatrix} \in A^*$$

représente le vecteur d'entiers naturels  $\vec{x} = (x_1, \dots, x_n)$  tel que  $x_i = \sum_{j=1}^m x_{i,j} 2^{j-1}$  pour tout  $1 \leq i \leq n$ .

On note  $\vec{x}$  le vecteur d'entiers et  $x$  le mot de  $A^*$  associé à ce vecteur d'entiers. On rappelle que «  $\cdot$  » représente la concaténation entre deux mots.

**V.A** – Résoudre le système  $(2x_1 - x_2 = -4) \wedge (7x_1 - 3x_2 = 1)$ . Écrire le mot correspondant.

Étant donnée une équation linéaire  $\langle \vec{a} | \vec{x} \rangle = k$ , on souhaite construire un automate  $\mathcal{A}_{\vec{a},k}$  qui reconnaisse seulement les mots correspondant aux solutions de l'équation.

**V.B** – Soit  $v$  un mot de longueur au moins égale à 2 que l'on écrit sous la forme  $v = b \cdot v'$ , où  $b$  est une lettre et  $v'$  un mot. Montrer que  $\vec{v}$  est solution de l'équation  $\langle \vec{a} | \vec{x} \rangle = k$  si et seulement si  $k - \langle \vec{a} | \vec{b} \rangle$  est un entier pair et  $\vec{v}'$  est solution de l'équation  $\langle \vec{a} | \vec{x} \rangle = k'$ , pour une valeur de  $k'$  que l'on précisera.

On peut donc ainsi construire l'automate. Les états sont indexés par les valeurs  $k_i$  accessibles. On ajoute un état « rebut » noté  $\perp$ . À partir d'un état  $k_i$ , pour toutes les lettres  $b$  de l'alphabet  $A$ , on crée les états  $k_j$ , s'ils n'existent pas encore et les transitions  $(k_i, b, k_j)$ , si  $k_i - \langle \vec{a} | \vec{b} \rangle$  est un entier pair (la valeur de  $k_j$  étant le  $k'$  de la question précédente) ou les transitions  $(k_i, b, \perp)$  sinon.

**V.C** – Préciser, en le justifiant l'état initial de l'automate  $\mathcal{A}_{\vec{a},k}$ , ainsi que le (les) état(s) final(aux).

**V.D** – Montrer que l'on construit ainsi un nombre fini d'états de l'automate  $\mathcal{A}_{\vec{a},k}$ .

**V.E** – Donner un algorithme permettant de déterminer s'il existe des solutions de l'équation  $\langle \vec{a} | \vec{x} \rangle = k$ . Justifier que cet algorithme se termine.

**V.F** – Construire l'automate pour l'équation :  $x_1 - 4x_2 + 2x_3 = 1$ . Donner également le tableau indiquant les transitions : en ligne les états accessibles ; en colonne les différentes lettres de l'alphabet (dans l'ordre lexical précisé ci-dessous) ; chaque case du tableau contient l'état atteint par lecture de la lettre à partir de l'état correspondant.

Pour éviter de surcharger le dessin de l'automate, on pourra ne pas représenter les transitions vers l'état  $\perp$ .

On définit l'ordre lexical sur les lettres de l'alphabet  $E^n$  par : si  $a = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}$  et  $b = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$  sont dans l'alphabet, on a  $(a < b) \Leftrightarrow [(a_1 < b_1) \text{ ou } (\exists i, 2 \leq i \leq n / \forall j, 1 \leq j < i, a_j = b_j \text{ et } a_i < b_i)]$ .

**V.G** – En déduire toutes les solutions du problème avec  $x_1 < 8$ ,  $x_2 < 8$  et  $x_3 < 8$ .

**V.H** – Comment peut-on modifier l'automate précédent pour prendre en compte les quantificateurs du premier ordre, c'est-à-dire des formules comme  $\exists x_1, \forall x_2, (x_1 - 4x_2 + 2x_3 = 1)$  ?

• • • FIN • • •