ÉCOLE POLYTECHNIQUE **FILIÈRE MP**

OPTION INFORMATIQUE

CONCOURS D'ADMISSION 2005

COMPOSITION D'INFORMATIQUE

(Durée : 4 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.

Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie.

⋆⋆⋆

Circuits PLA et additionneurs

On attachera une grande importance à la concision, à la clarté, et à la précision de la rédaction. Les deux parties du problème sont quasi indépendantes.

I. Génération de circuits PLA

Dans ce problème, les booléens sont représentés par 0 pour la valeur faux, et 1 pour la valeur vraie. Soit $B = \{0, 1\}$ l'ensemble des booléens. Si x est un booléen, on notera $\overline{x} = 1 - x$ le complément de x. On écrira $x \cdot y$ pour la conjonction (produit) de x et y ; $x \dot{+} y$ pour la disjonction de x et y en posant $0 \dot{+} 0 = 0$ et $0 \dot{+} 1 = 1 \dot{+} 0 = 1 \dot{+} 1 = 1$; et $x \oplus y$ pour le ou-exclusif, défini par $0 = 0 \oplus 0 = 1 \oplus 1$ et $1 = 0 \oplus 1 = 1 \oplus 0$.

La représentation binaire $(x_0, x_1, \ldots x_{n-1})$ de l'entier x sur n bits $(0 \leq x < 2^n)$ est définie par :

$$x = x_0 2^0 + x_1 2^1 + \cdots + x_{n-1} 2^{n-1}$$

où x_i est un booléen pour tout i $(0 \leq i < n)$. Dans le problème, on ne considère que des représentations binaires d'entiers x vérifiant $x < 2^{30}$, et donc représentables comme de simples entiers positifs en Pascal ou Caml. On écrira aussi $\vec{x}$ pour la représentation binaire de x quand n est clair dans le contexte. Les *bits* de x sont les booléens x_0, x_1, $\ldots x_{n-1}$; x_0 est le bit le moins significatif ; x_{n-1} est le bit le plus significatif.

Enfin, on suppose que les deux langages Pascal et Caml possèdent les opérations logiques $\wedge$, $\vee$, $\oplus$, $\neg$ sur les entiers positifs définies par :

$$x \wedge y = (x_0 \cdot y_0)2^0 + (x_1 \cdot y_1)2^1 + \cdots + (x_{29} \cdot y_{29})2^{29}$$
$$x \vee y = (x_0 \dot{+} y_0)2^0 + (x_1 \dot{+} y_1)2^1 + \cdots + (x_{29} \dot{+} y_{29})2^{29}$$
$$x \oplus y = (x_0 \oplus y_0)2^0 + (x_1 \oplus y_1)2^1 + \cdots + (x_{29} \oplus y_{29})2^{29}$$
$$\neg x = \overline{x}_0 2^0 + \overline{x}_1 2^1 + \cdots + \overline{x}_{29} 2^{29}$$

si $x = x_0 2^0 + x_1 2^1 + \cdots + x_{29} 2^{29}$ et $y = y_0 2^0 + y_1 2^1 + \cdots + y_{29} 2^{29}$

qui s'exécutent en temps constant $O(1)$. Ainsi on autorise les expressions arithmétiques de Caml ou Pascal à être de la forme $e \wedge e'$, $e \vee e'$, $e \oplus e'$ et $\neg e$.

Question 1 Montrer que, si $0 < x < 2^{30}$, alors x est de la forme $x = 2^p$ $(p \geq 0)$ si et seulement si $x \wedge (x - 1) = 0$.

La distance de Hamming $d(x, y)$ entre deux entiers x et y est le nombre de bits dont ils diffèrent dans leur décomposition binaire $\vec{x}$ et $\vec{y}$ sur 30 bits.

x_1	x_0	$g(x_0,x_1)$
0	0	1
0	1	0
1	0	1
1	1	1

x_2	x_1	x_0	$h(x_0,x_1,x_2)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

FIG. 1: *Tables de vérité de g et h.*

Question 2 Écrire la fonction `aDistUn` qui prend comme argument deux entiers x et y et retourne, en temps constant, la valeur vrai si et seulement si la distance de Hamming entre x et y vaut 1.

(* *Caml* *)	{ *Pascal* }
`aDistUn : int -> int -> bool`	`function aDistUn (`x`:integer, `y`:integer) : boolean;`

Soit f une fonction booléenne de n arguments booléens ; donc f est une fonction de B^n dans B ($n > 0$). Deux exemples de fonctions booléennes sont g et h définies par : $g(x_0,x_1) = \overline{x}_0 \dot{+} x_0 \cdot x_1$, et $h(x_0,x_1,x_2) = x_0 \oplus x_2$. Une fonction booléenne peut être définie par sa table de vérité, comme sur la figure 1. Dans le problème, nous représenterons toute fonction booléenne f par l'ensemble des entiers dont la représentation binaire sur n bits est dans $f^{-1}(1)$, image inverse de 1. Ainsi g est définie par l'ensemble $\{0,2,3\}$ et h par $\{1,3,4,6\}$.

Un *littéral* est une variable x ou son complément $\overline{x}$; on dira que x est un littéral positif et $\overline{x}$ est un littéral négatif. Toute fonction booléenne $f(x_0,x_1,\dots x_{n-1})$ peut être mise en *forme normale disjonctive* (f.n.d.) en l'exprimant comme une somme de produits de littéraux formés à partir de x_0, x_1, $\dots x_{n-1}$. Ainsi $g(x,y) = \overline{x}\cdot\overline{y} \dot{+} \overline{x}\cdot y \dot{+} x\cdot y = \overline{x} \dot{+} x\cdot y = \overline{x} \dot{+} y$ s'écrit sous trois f.n.d. distinctes.

Question 3 Exprimer h avec une f.n.d. sous forme de la somme de 4 produits.

Pour réduire le nombre de produits dans la f.n.d. de f, on utilisera principalement l'identité

$$x\cdot p \dot{+} \overline{x}\cdot p = (x \dot{+} \overline{x})\cdot p = 1\cdot p = p$$

Ainsi $g(x,y) = \overline{x}\cdot\overline{y} \dot{+} \overline{x}\cdot y \dot{+} x\cdot y = \overline{x}\cdot(\overline{y} \dot{+} y) \dot{+} x\cdot y = \overline{x} \dot{+} x\cdot y$. Remarquons qu'on peut aussi utiliser l'identité $p \dot{+} p = p$ et obtenir $g(x,y) = \overline{x}\cdot(\overline{y} \dot{+} y) \dot{+} (\overline{x} \dot{+} x)\cdot y = \overline{x} \dot{+} y$. Dans cet exemple le nombre de produits n'est pas plus faible, mais en général cette identité supplémentaire permet de le faire baisser. Le but de cette partie est de trouver efficacement parmi toutes les réécritures de la f.n.d. l'une de celles ayant un nombre de produits minimal, ou presque. On appellera *f.n.d. réduite* le résultat de notre algorithme.

Question 4 Réduire le nombre de produits dans la f.n.d. de h.

Un ensemble $\{i_0, i_1, \dots i_{\ell-1}\}$ d'indices compris entre 0 et 29 ($0 \le i_k < 30$, $0 \le k < \ell$) est représentable par l'entier dont la représentation binaire a des bits à 1 aux seules positions i_0, i_1, $\dots i_{\ell-1}$. Un produit de littéraux $x_{i_0}\cdot x_{i_1}\cdots x_{i_{\ell-1}}$ ($0 \le i_0 < i_1 \dots < i_{\ell-1} < 30$) est représenté par un enregistrement contenant deux entiers : une valeur v et un masque m. Le masque m est l'entier désignant l'ensemble $E = \{i_0, i_1, \dots i_{\ell-1}\}$; la valeur v correspond au sous-ensemble de E où x_{i_k} est un littéral positif. Ainsi le produit $x_0\cdot x_2\cdot\overline{x}_3\cdot x_4$ est représenté par les deux entiers $v = 21$ et $m = 29$ dont les représentations binaires respectives (sur 30 bits) sont $(1,0,1,0,1,0,\dots 0)$ et $(1,0,1,1,1,0,\dots 0)$. En abrégé, le produit s'écrira `101-1--`$\cdots$`-` ou plus simplement `101-1`.

(* *Caml* *)	{ *Pascal* }
`type produit = {v: int; m: int};;`	`type produit = record v, m:integer; end;`

Question 5 Écrire les fonctions `varEliminable` et `unifier` qui prennent deux produits p et q en arguments ; et qui, la première, retourne la valeur vrai si $p = p' \cdot x_i \cdot p''$ et $q = p' \cdot \overline{x}_i \cdot p''$ ou $p = p' \cdot \overline{x}_i \cdot p''$ et $q = p' \cdot x_i \cdot p''$; et qui, la seconde, retourne le produit $p' \cdot p''$ dans le cas où la première fonction vaut vrai.

(* *Caml* *)	{ *Pascal* }
`varEliminable : produit -> produit -> bool`	`function varEliminable (`p`:produit, `q`:produit) : boolean;`
`unifier: produit -> produit -> produit`	`function unifier (`p`:produit, `q`:produit) : produit;`

Quelle est la complexité en temps de ces deux fonctions ?

Une somme (disjonction) de produits est représentée par une liste de produits :

(* *Caml* *)	{ *Pascal* }

`type somme == produit list;;`	`type somme = ^cellule;`
	`  cellule = record contenu:produit;`
	`  suivant:somme; end;`

En Pascal, la liste vide est `nil` et l'on pourra pour construire les listes utiliser la fonction suivante :

```
function cons(contenu:produit; suivant:somme) : somme;
var r:somme;
begin new(r); r^.contenu := contenu; r^.suivant := suivant; cons := r end;
```

Cette fonction est applicable pour construire les listes du type `somme`.

Question 6 Écrire la fonction `unique` qui prend en argument une somme a ; et qui retourne la même somme où chacun des produits n'apparaît qu'une seule fois.

(* *Caml* *)	{ *Pascal* }
`unique : somme -> somme`	`function unique (`a`:somme) : somme;`

Quelle est la complexité en temps de cette fonction par rapport à la longueur ℓ de la somme a ?

Question 7 Écrire la fonction `nouveauxProduits` qui prend en argument une somme a ; et qui retourne la somme de tous les produits qu'il est possible d'obtenir en éliminant une variable inutile entre deux produits de a.

(* *Caml* *)	{ *Pascal* }
`nouveauxProduits : somme -> somme`	`function nouveauxProduits (`a`:somme) : somme;`

Quelle est la complexité en temps de cette fonction par rapport à la longueur ℓ de la somme a ?

Pour générer une f.n.d. réduite de la fonction booléenne f de n arguments, on commence par fabriquer la somme a_0 de tous les produits de n variables correspondant à $f^{-1}(1)$, image inverse de 1 par f (par exemple, `00`, `01`, `11` pour g). Puis on applique la fonction `nouveauxProduits` à a_0, donnant la somme a_1. Et on recommence en rappelant cette fonction sur a_1, donnant une autre somme a_2, etc. On s'arrête quand on ne produit plus de nouveaux produits. Ainsi pour g, on obtient `0-`, `-1` pour a_1, et on s'arrête.

Question 8 Donner la suite des a_i obtenus pour h.

Pour le moment, on garde tous les produits. Il est donc plus simple de manipuler des listes de sommes pour stocker $\langle a_k, \ldots, a_2, a_1, a_0 \rangle$.

(* *Caml* *)	{ *Pascal* }

`type fnd == somme list;;`	`type fnd = ^celluleS;`
	`  celluleS = record contenu:somme;`
	`  suivant:fnd; end;`

Pour construire ces listes, on utilisera la fonction `consFnd` analogue de la fonction `cons`.

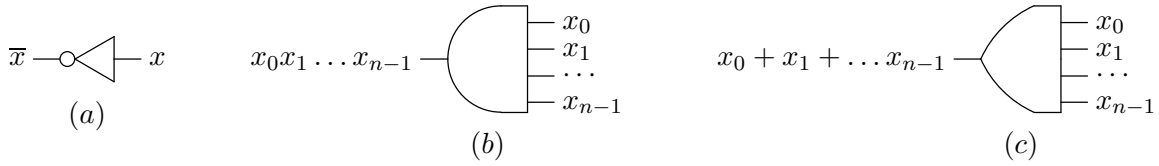

FIG. 2: *Trois portes logiques : (a) inverseur(1 entrée x, 1 sortie $\overline{x}$) ; (b) porte-*ET*(n entrées, 1 sortie) ; (c) porte-*OU*(n entrées, 1 sortie).*

Question 9 Écrire la fonction `developper` qui prend en argument une somme a ; et qui retourne la liste $\langle a_k, \ldots a_2, a_1, a_0 \rangle$ des sommes obtenues en itérant la fonction `nouveauxProduits` à partir de $\langle a \rangle$.

(Caml *)*	*{ Pascal }*
`developper : somme -> fnd`	`function developper (a:somme) : fnd;`

Quelle est la complexité de cette fonction par rapport aux longueurs ℓ_0, ℓ_1, $\ldots \ell_k$ des listes a_0, a_1, $\ldots a_k$?

Le résultat de la fonction `developper` est une liste de listes, ayant beaucoup plus de termes que la somme de départ, et dont on va extraire les produits formant la f.n.d. réduite. Cela se fait en éliminant tous les produits *couverts* par un autre produit. Ainsi le produit `-0` couvre les produits `00` et `10` ; de même `1-` couvre `10` et `11`. Formellement, le produit $p' \cdot p''$ couvre les produits $p' \cdot x_i \cdot p''$ et $p' \cdot \overline{x}_i \cdot p''$. En outre, p couvre p'' si p couvre p' et p' couvre p''.

Question 10 Écrire la fonction `couvertPar` qui prend en argument deux produits p et q ; et qui retourne la valeur vrai si p est couvert par q.

(Caml *)*	*{ Pascal }*
`couvertPar : produit -> produit -> bool`	`function couvertPar (p:produit, q:produit) : boolean;`

Quelle est la complexité en temps de cette fonction ?

Question 11 Écrire la fonction `reduire` qui prend en argument une f.n.d. s ; et qui retourne une f.n.d. calculant la même fonction où il n'y a plus aucun produit couvrant un autre.

(Caml *)*	*{ Pascal }*
`reduire : fnd -> fnd`	`function reduire (s:fnd) : fnd;`

Quelle est la complexité de cette fonction par rapport aux longueurs ℓ_0, ℓ_1, $\ldots \ell_k$ des listes a_0, a_1, $\ldots a_k$ composant la représentation $\langle a_k, \ldots a_2, a_1, a_0 \rangle$ de s ?

Question 12 Donner une borne supérieure de la complexité du calcul du résultat de cette f.n.d. réduite en fonction du nombre n d'arguments de la fonction f.

La partie combinatoire d'un circuit contient une combinaison de portes-ET, portes-OU et inverseurs représentés sur la figure 2. Les signaux circulant dans les fils des circuits sont assimilés aux booléens 0 et 1. Une porte-OU à n entrées x_0, x_1, $\ldots x_{n-1}$ calcule la somme (disjonction) $x_0 \dot{+} x_1 \dot{+} \cdots \dot{+} x_{n-1}$; de même une porte-ET à n entrées x_0, x_1, $\ldots x_{n-1}$ calcule le produit $x_0 \cdot x_1 \cdots x_{n-1}$, et l'inverseur à une entrée x calcule $\overline{x}$.

Cette partie combinatoire des circuits intégrés consiste souvent à calculer une fonction f de B^n dans B^m ($0 < n < 30$, $0 < m$) à l'aide d'un PLA (*Programmable Logic Array*). Un PLA, à n entrées et m sorties comme indiqué sur la figure 3, calcule sur chaque sortie la fonction booléenne $f_i(x_0, x_1, \ldots x_{n-1})$ ($0 \leq i < m$) associée à f en s'appuyant sur sa représentation en f.n.d.. Un PLA comporte deux parties : la partie-ET ne contenant que des portes-ET et quelques inverseurs, et la partie-OU ne contenant que des portes-OU.

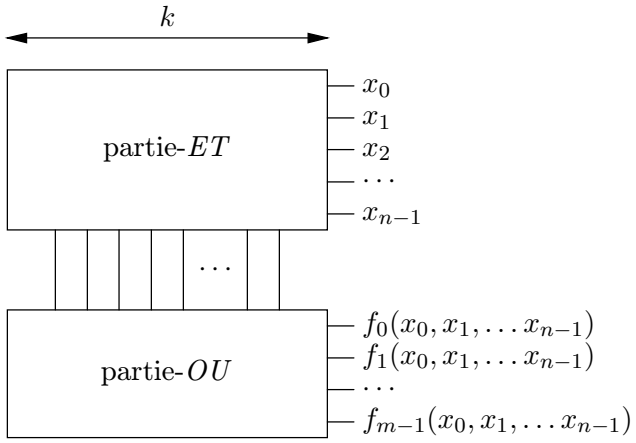

FIG. 3: *Circuit PLA.*

Question 13 Pour une fonction f donnée de B^n dans B^m ($0 < n < 30$, $0 < m$)

a) Expliquer le fonctionnement du PLA associé. Dessiner le quand $f_0(x_0, x_1, x_2) = g(x_0, x_1)$ et $f_1(x_0, x_1, x_2) = h(x_0, x_1, x_2)$

b) Déterminer la largeur k du PLA comme indiqué sur la figure 3 permettant de calculer sa surface. Comment réduire cette surface ?

II. Génération de circuits combinatoires

Dans cette partie, nous allons générer d'autres exemples de circuits combinatoires calculant des fonctions booléennes. Ces circuits ne seront composés que d'inverseurs, de portes-ET ou portes-OU à 2 entrées (cf. la figure 2), et de bits valant 0 ou 1. Ils seront représentés par des arbres donnant la valeur de la sortie en fonction des valeurs des entrées, c'est-à-dire des bits figurant à leurs feuilles. Le type de ces arbres est le suivant :

(* *Caml* *)

```
type circuit =
   Bit of int
 | Et of circuit*circuit
 | Ou of circuit*circuit
 | Non of circuit;;
```

{ *Pascal* }

```
type circuit = ^porte; porte = record case indicateur of
    Bit: (val:integer);
    Et: (a1:circuit; a2:circuit) ;
    Ou: (a1:circuit; a2:circuit) ;
    Non: (a1:circuit) ; end;
```

On ne se souciera pas du partage possible entre sous-arbres calculant la même valeur. Mais on remarquera que le temps mis par un circuit pour calculer son résultat est proportionnel à la hauteur de cet arbre. Par exemple, on peut calculer en 3 unités de temps le ou-exclusif de x et y comme suit :

(* *Caml* *)

```
let oux x y =
  Ou (Et (x, Non y),
      Et (Non x, y));;
```

{ *Pascal* }

```
function oux (x: circuit; y:circuit): circuit;
  begin oux := nouveauOu (nouveauEt(x, nouveauNon (y)),
       nouveauEt(nouveauNon(x), y)); end;
```

En Pascal, pour construire ces circuits, on utilisera les fonctions `nouveauEt`, `nouveauOu` et `nouveauNon`, analogues aux constructeurs `cons` et `consFnd` de la partie I.

Question 14 Écrire les fonctions `bitAdd` et `bitRetenue` qui prennent en argument trois circuits x, y et r fournissant les valeurs x', y' et r' ; et qui, la première, retourne le circuit calculant le reste modulo 2 de $x' + y' + r'$; et qui, la seconde, retourne le circuit calculant le quotient de la division par 2 de $x' + y' + r'$.

(* *Caml* *)

```
bitAdd : circuit -> circuit ->
      -> circuit -> circuit
bitRetenue: circuit -> circuit ->
      -> circuit -> circuit
```

{ *Pascal* }

```
function bitAdd (x:circuit; y:circuit;
                 r:circuit) : circuit;
function bitRetenue (x:circuit; y:circuit;
                 r:circuit) : circuit;
```

La représentation binaire $\vec{x}$ de l'entier x sur n bits ($n > 0$) est à présent décrite par un vecteur (tableau) de n circuits, dont la i-ème entrée vaut le bit x_i, comme indiqué dans la partie I. On appellera *mot machine* de longueur n ce tableau. Un additionneur série de $\vec{x}$ et $\vec{y}$ effectue l'addition successive des bits x_i et y_i en partant des bits les moins significatifs vers les bits les plus significatifs en propageant la retenue.

Question 15 Écrire la fonction `addSerie` qui prend en arguments deux mots machine x et y de longueur n ; et qui retourne le circuit calculant le mot machine de longueur $n+1$ représentant $x+y$. Quel est le temps de calcul de ce circuit ? Donner un ordre de grandeur du nombre de portes de ce circuit.

(* *Caml* *)

```
type mot == circuit vect;
addSerie : mot -> mot -> mot
```

{ *Pascal* }

```
type mot = array[0..256] of circuit;
procedure addSerie (var z: mot; n:integer;
   var x: mot; var y: mot);
```

En Caml, on supposera que les paramètres x et y ont une même longueur n. En Pascal, la fonction prend la longueur n effective du mot en argument ; le résultat est retourné dans le paramètre z passé par référence.

Question 16 Écrire la fonction `mux` qui prend en arguments un circuit s et deux mots machines x et y ; et qui retourne le circuit fournissant la valeur de x si s' fourni par s vaut 1, et celle de y si s' vaut 0.

(* *Caml* *)

```
mux: circuit -> mot -> mot -> mot
```

{ *Pascal* }

```
procedure mux (var z: mot; n:integer;
  s: circuit; var x: mot; var y: mot);
```

On peut calculer l'addition plus rapidement en coupant les mots machine x et y à additionner en deux parties basses x_b, y_b contenant les bits les moins significatifs et deux parties hautes x_h, y_h contenant les bits les plus significatifs. Pour ne pas attendre le résultat de la retenue de $x_b + y_b$ pour calculer $x_h + y_h$, on peut précalculer les résultats de l'addition de x_h et y_h avec la retenue valant 0 et celle valant 1. Puis le véritable résultat de la retenue de $x_b + y_b$ permet de donner le résultat voulu pour $x_h + y_h + r$.

Question 17 Écrire la fonction `addPar` qui prend en arguments deux mots machine x et y de longueur n et un circuit r fournissant une retenue r' ; et qui retourne le circuit calculant le mot machine de longueur $n+1$ représentant $x+y+r'$. Quel est le temps de calcul de ce circuit ? Donner un ordre de grandeur du nombre de portes de ce circuit.

(* *Caml* *)

```
addPar : mot -> mot -> circuit -> mot
```

{ *Pascal* }

```
procedure addPar (var z: mot; n:integer;
  var x: mot; var y: mot; r: circuit);
```

Question 18 Pourquoi ne pas utiliser un PLA pour réaliser ces additionneurs ? Quels auraient été les avantages/désavantages ?

* *
*