

Module CAO

Cours du Professeur A.Greiner / Juin 2006

Durée 3h00

Partie A (14 points):

Utilisation des ROBDD pour la représentation des fonctions Booléennes

Soit une fonction Booléenne quelconque F , représentée par un ROBDD. On cherche à exprimer la fonction F comme une somme de produits.

Il y a en général plusieurs chemins entre le noeud BDD représentant la fonction F et le noeud BDD représentant la constante 1. Tous ces chemins n'ont pas la même longueur, mais chaque chemin correspond à un produit de variables (opérateur AND). L'expression Booléenne construite en effectuant la somme (opérateur OR) entre tous les produits associés aux différents chemins répond à la question.

A1 (2pts) Dessiner graphiquement le multi-ROBDD permettant de représenter les deux fonctions Booléennes F_1 et F_2 associées aux expressions Booléennes infixées suivantes :

$$F_1 = A \text{ xor } B \text{ xor } C$$

$$F_2 = (A \text{ or } B \text{ or } C') \text{ and } (B' \text{ or } C) \quad \text{La notation } C' \text{ signifie not}(C)$$

On construira le graphe pour l'ordre $A > B > C$, et on attachera à chaque noeud BDD du graphe ROBDD une expression Booléenne décrivant la fonction Booléenne représentée par ce noeud.

Un chemin entre le noeud BDD représentant la fonction F et le noeud BDD représentant le noeud 1 est une suite d'arcs. Dans un ROBDD, tout arc est caractérisé par deux informations : la variable associée au noeud source, et l'étiquette (0 ou 1) qui précise si le noeud destination est le cofacteur FH ou FL . On décide donc d'associer à chaque arc un nom de variable de la façon suivante : Si un noeud du graphe ROBDD possède l'index x associé à la variable X , on associe la variable X à l'arc sortant étiqueté par 1, et on associe la variable X' à l'arc sortant étiqueté par 0. On peut maintenant associer à chaque chemin le produit (AND) des variables (X ou X') associées aux arcs constituant le chemin.

A2 (2pts) Appliquer la méthode suggérée ci-dessus pour exprimer les deux fonctions F_1 et F_2 sous forme d'une somme de produits.

On définit $\text{Chemins}(F)$ comme l'ensemble des chemins entre le noeud BDD représentant la fonction F et le noeud BDD représentant le noeud 1. Chaque chemin définit un produit de variables, directes ou complémentées.

A3 (2pts) Ecrire, en Français, la relation de récurrence qui existe entre les ensembles $\text{Chemins}(F)$, $\text{Chemins}(FH)$, et $\text{Chemins}(FL)$, où FH et FL sont les cofacteurs dans la décomposition de Shannon de la fonction F , par rapport à la variable X d'index le plus élevé du support de F .

On étudiera successivement les 6 cas terminaux, avant de traiter le cas général :

- FH = 1 et FL = 0
- FH = 0 et FL = 1
- FH = 1 et FL différent de 0
- FH = 0 et FL différent de 1
- FL = 1 et FH différent de 0
- FL = 0 et FH différent de 1

On utilisera les notations suivantes :

- $\{X\}$ est l'ensemble de chemins contenant un seul chemin, ce chemin étant composé d'un seul arc entre le noeud BDD représentant la fonction X et le noeud BDD représentant la constante 1.
- $\{X'\}$ est l'ensemble de chemins contenant un seul chemin, ce chemin étant composé d'un seul arc entre le noeud BDD représentant la fonction $\text{not}(X)$ et le noeud BDD représentant la constante 1.
- $\{X.\text{Chemins}(F)\}$ est l'ensemble de chemins obtenus en concaténant la variable X en tête de tous les chemins appartenant à l'ensemble $\text{Chemins}(F)$. L'ensemble $\{X.\text{Chemins}(F)\}$ a donc le même nombre d'éléments que l'ensemble $\text{Chemins}(F)$, mais la longueur de chaque chemin est augmentée d'une unité.
- Si $E1$, $E2$ sont des ensembles de chemins alors $E1$ union $E2$ est l'ensemble constitué de tous les chemins de $E1$ et de tous les chemins de $E2$.

On pourra utiliser les structures de données et les fonctions suivantes :

- **chemin**

Un chemin est une liste ordonnée d'arcs dans le ROBDD. On définit la structure `C arc_t`, qui permet de représenter un chemin sous forme d'une liste chaînée de variables (chaque variable pouvant être directe ou complémentée) :

```
typedef struct arc_t {
    struct arc_t *NEXT ; // pointeur sur l'arc suivant dans le chemin
    struct var_t *VAR ;   // pointeur sur la variable associée au noeud source
    int          NOT ;    // la variable est complémentée si NOT est différent de 0
} arc_t ;
```

La fonction `arc_t * cons_arc (arc_t *next, var_t *var, int not)` permet de créer en mémoire un objet de type `arc_t` et de l'initialiser.

On représentera un ensemble de chemins comme une liste chaînée de bipointeurs, ou le champs `DATA` du bipointeur pointe sur le premier arc du chemin.

- **noeud bdd**

```
typedef struct bdd_t {
    struct bdd_t *HIGH ; // pointeur sur le noeud représentant le cofacteur FH
    struct bdd_t *LOW ;  // pointeur sur le noeud représentant le cofacteur FL
    unsigned      INDEX ;
} bdd_t
```

La fonction `bdd_t * create_bdd (unsigned index, bdd_t *high, bdd_t *low)` permet de créer en mémoire un objet de type `bdd_t` et de l'initialiser (si celui n'existe pas déjà).

- **bi-pointeur**

```
typedef struct bip_t {
    struct bip_t *NEXT ; // pointeur sur le bipointeur suivant de la liste chaînée
    void *DATA ; // pointeur sur un élément de la liste chaînée
} bip_t
```

La fonction **bip_t * cons_bip (bip_t *next, void *data)** permet de créer en mémoire un objet de type bip_t et de l'initialiser.

La fonction **bip_t * union_bip (bip_t *l1, bip_t *l2)** prend pour argument deux listes de bipointeurs l1 et l2 éventuellement vide et qui rend l'union au sens des ensembles.

- **variable**

```
typedef struct var_t {
    char *NAME ; // Nom de la variable Booléenne
    unsigned INDEX ; // Index de la variable Booléenne
    unsigned VALUE ; // Valeur logique de la variable Booléenne
} var_t
```

La fonction **var_t * cons_var (char *name, unsigned index, unsigned value)** permet de créer en mémoire un objet de type var_t, de l'initialiser, et de la ranger dans un dictionnaire.

La fonction **var_t * get_var_index(unsigned index)** prend pour argument un entier représentant l'index d'une variable et renvoie un pointeur sur l'objet var_t correspondant.

A4 (3pts) Ecrire, en langage C la fonction récursive **bip_t *chemins_bdd(bdd_t *p)**. Cette fonction prend pour argument un pointeur sur un noeud BDD représentant une fonction F, et renvoie un pointeur sur une liste chaînée de bipointeurs représentant l'ensemble des chemins entre le noeud représentant F et le noeud BDD représentant la constante 1. Vous écrirez une fonction intermédiaire **bip_t *add_var_to_all (var_t *v, int not, bip_t *c)** qui concatène l'arc défini par la variable v et le type not (=0 la variable n'est pas inversée, =1 la variable est inversée), en tête de tous les chemins appartenant à l'ensemble représenté par c. La liste rendue est c (passée en paramètre) mais chaque chemin est augmenté d'une unité.

On souhaite maintenant construire en mémoire l'arbre ABL représentant les expressions Booléennes de type « somme de produits » associées à l'ensemble de chemins construits par la fonction chemins_bdd().

A5 (2pts) Dessiner graphiquement l'arbre ABL représentant l'expression Booléenne de type « somme de produits » associée à la fonction F1, obtenue à la question B2.

A6 (3pts) Ecrire en langage C la fonction **bip_t * cons_sum_of_product(bip_t *p)**. Cette fonction prend pour argument un pointeur sur une liste de bipointeurs représentant un ensemble de chemins (liste construite par la fonction chemins_bdd), et renvoie un pointeur sur le bipointeur racine de l'arbre ABL représentant l'expression Booléenne « somme de produits ». On écrira une fonction intermédiaire **bip_t * cons_product(arc_t *chemin)**, qui prend pour argument un pointeur sur un chemin, et renvoie un pointeur sur l'arbre ABL représentant le produit des variables associés à ce chemin (en complétant les variables qui doivent être complétées).

On n'oubliera pas de traiter le cas où la liste des chemins ne contient qu'un seul chemin (l'opérateur OR est inutile), ainsi que le cas où un chemin ne comporte qu'un seul arc (l'opérateur AND est inutile).

On pourra utiliser le type suivant pour coder les opérateurs NOT, OR et AND :

```
typedef enum op_t {
    NOT = 0 ,
    OR = 1 ,
    AND = 2
} op_t
```

Partie B (6 points) : Expression d'une grammaire YACC

On souhaite écrire la grammaire avec la syntaxe yacc du format de fichier DOT. Pour cet exercice nous allons nettement simplifier le format de fichier DOT afin de le réduire à l'expression d'un graphe non orienté éventuellement partitionné. Les nœuds du graphe sont représentés par des index, les arcs ne sont pas nommés. Pour décrire la syntaxe nous vous proposons un exemple commenté.

Les commentaires commencent par // et s'achève en fin de ligne. Le caractère retour chariot est considéré comme un séparateur (au même titre que l'espace ou la tabulation).

```
graph // ceci définit un graphe non orienté
{
    // les subgraph permettent de définir les parties du graphe, ils ne sont pas obligatoires.
    // le mot subgraph doit être suivi du mot cluster auquel on accole le numéro de la partie.

    subgraph cluster0 { 5 4 1 0 } // on place entre accolades les numéros des noeuds de
                                   // la partie, ici la partie 0 contient les noeuds 5, 4, 1, 0.

    subgraph cluster1 { 7 6 3 2 } // la partie 1 contient les noeuds 7, 6, 3 et 2

    0--1 // ceci définit un arête entre le nœud 0 et le nœud 1.
    0--4 // on peut avoir plusieurs arêtes sortant du même nœud.
    0--5 // l'ordre des arêtes n'a pas d'importance
    1--4
    1--5
    2--3
    2--6
    3--7
    3--6
    4--5
    6--7
}
```

B1 (2pts) Donner la liste des mots clé (token) du langage, et leur définition sous forme d'expressions régulières.

B2 (2pts) Représenter l'arbre syntaxique du langage sous la forme d'un dessin. Vous choisirez le nom des règles non-terminales. Nous vous imposons le nom de la règle de plus haut niveau : **graph**.

B3 (2pts) Représenter le même arbre syntaxique avec la syntaxe yacc. **Nous ne demandons pas d'écrire des actions pour les règles, ni d'écrire le fichier yacc complet, il s'agit seulement d'écrire les règles.**