TME 5 : Représentation des expressions Boolèennes : ABL

1. Objectif
2. A) Structures de données et fonctions de base
3. B) Représentation Graphique
4. C) Fonction d?affichage
5. D. Calcul du nombre de littéraux
6. E) Calcul du support
7. F) Fonction d?évaluation
8. Compte-rendu

Objectif

L'objectif principal de ce TME est de vous familiariser avec la représentation des expressions Booléennes sous forme d'arbres ABL (Abres Binaires Lisp-Like), et à rafraîchir vos connaissances dans le domaine de la programmation des fonctions récursives.

Les expressions Booléennes multi-niveaux sont des expressions Booléennes préfixées pouvant comporter plusieurs niveaux d?imbrication de parenthèses. On représente une expression Booléenne comme une liste chaînée de bipointeurs. Le premier élément de la liste définit l?opérateur (les 4 opérateurs utilisés sont NOT, OR, AND et XOR). Les éléments suivants de la liste chaînée représentent les opérandes. Chaque opérande peut être :

- soit une variable Booléenne, définie par son nom.
- soit une autre expression Booléenne.

A) Structures de données et fonctions de base

La structure C permettant de représenter une variable Booléenne possède trois champs : Le champs NAME représente le nom de la variable, le champs INDEX définit un numéro, et le champs VALUE représente la valeur logique de la variable. Un variable peut être identifiée soit par son nom, soit par son index, ce qui signifie que toutes les variables doivent avoir des noms et des index différents.

```
typedef struct var_t {
        char            *NAME;
        unsigned int    INDEX;
        unsigned int    VALUE;
} var_t
```

La structure C permettant de représenter un noeud de l?arbre ABL est un bipointeur, défini de la façon suivante :

```
typedef struct bip_t {
        struct bip_t    *NEXT;
        void            *DATA;
} bip_t
```

Dans le cas des arbres ABL, le champs DATA peut contenir soit un pointeur vers un noeud de l?arbre (bip_t *), soit un pointeur vers une variable Booléenne (var_t *), soit un entier définissant le code d?un opérateur Booléen (unsigned int). Par conséquent, un cast est nécessaire chaque fois qu?on veut affecter une valeur ou lire la valeur contenue dans ce champs de la structure.

On dispose des fonctions suivantes :

```
extern var_t  *cons_var(char *name, unsigned int index, unsigned int value)
extern var_t *get_var_index(unsigned int x)
extern var_t *get_var_name(char *s)
extern bip_t  *cons_bip(bip_t *next, void *data)
extern void  free_bip(bip_t *p)
extern bip_t *parse_abl( char *expr)
```

- La fonction **cons_var()** crée un objet de type var_t, vérifie qu'il n'existe pas de variable portant le même nom ou le même index, initialise les champs NAME, INDEX et VALUE de la structure de donnée, range la variable dans un dictionnaire, et renvoie un pointeur sur la variable ainsi créée.
- Les deux fonctions **get_var_index()** et **get_var_name()** renvoient un pointeur vers la variable désignée soit par son nom, soit par son index, et affichent un message d?erreur si la variable n?existe pas.
- La fonction **cons_bip()** crée un bipointeur de type bip_t, en affectant la valeur data au champs DATA, et la valeur next au champs NEXT, et renvoie un pointeur sur le bipointeur ainsi créé. La fonction **free_bip()** permet de libérer la mémoire allouée par la fonction cons_bip().
- La fonction **parse_abl()** prend en entrée une chaîne de caractères décrivant une expression Booléenne préfixée possédant un nombre quelconque de niveaux de parenthèsage. Elle construit en mémoire l?arbre ABL représentant cette expression Booléenne et renvoie un pointeur sur le bipointeur correspondant à la racine. Cette fonction n?accepte que les expressions Booléennes préfixées, et les seuls opérateurs acceptés sont NOT, OR, AND et XOR. Elle affiche un message d?erreur en cas d?erreur de syntaxe, ou si elle rencontre un nom de variable non déclarée préalablement.

Pour utiliser ces structures de données et ces fonctions, vous devez inclure dans votre programmes les fichiers suivants:

```
#include <bip.h>
#include <var.h>
#include <parse_abl.h>
```

Récupérez l'archive TME5. En la décompressant, vous obtiendrez un répertoire `tme5` contenant les fichiers dont vous avez besoin pour ce TME, et en particulier un fichier `Makefile` et un fichier `main.c` que vous devrez complêter:

B) Représentation Graphique

Soit l?expression `E0 = (a? . (b + c + d?) . e) + c`

La notation a? signifie NOT(a)

- **B.1)** Re-écrire cette expression sous une forme préfixée, et représentez graphiquement l?arbre ABL correspondant à l?expression E0.

C) Fonction d?affichage

- **C.1)** Ecrire en langage C la fonction récursive display_abl() présentée en cours. cette fonction prend en entrée un pointeur p sur un bipointeur représentant la racine d'un arbre ABL, et affiche sur le terminal standard l?expression Booléenne normalisée correspondante.

```
void display_abl(bip_t *p)
```

- **C.2)** Mofifiez la fonction main() pour qu'elle effectue les opérations suivantes: construction en mémoire de l?arbre ABL correspondant à l?expression Booléenne E0 en utilisant la fonction parse_abl(), puis affichage de cette expression sur le terminal standard, au moyen de la fonction display_abl(). Compilez ce programme et exécutez-le.

D. Calcul du nombre de littéraux

On cherche maintenant à écrire en langage C la fonction récursive count_abl(). cette fonction prend en entrée un pointeur p sur la racine d?un arbre ABL, et renvoie un entier représentant le nombre de littéraux (c?est và dire le nombre de feuilles de l?arbre ABL).

```
int count_abl(bip_t *p)
```

- **D.1)** Rappeler la relation de récurrence permettant d?exprimer le nombre de littéraux d?une expression Booléenne, connaissant le nombre de littéraux de chacun de ses opérandes.
- **D.2)** Ecrire le code de la fonction en langage C, et modifier le programme main() de la question C) pour afficher également le nombre de littéraux de l?expression construite en mémoire.
- **D.3)** Exécutez le programme pour les expressions Booléennes suivantes :

```
E1 = (a.b) + (a.c) + (b.c)
E2 = (a.b.c) + (a.b.c?) + (a.b?.c) + (a?.b.c)
```

E) Calcul du support

On appelle support d?une expression Booléenne l?ensemble de toutes les variables qui apparaissent au moins une fois dans cette expression. Pour représenter un ensemble de variables, on utilise également des bipointeurs : Un ensemble est représenté par une liste chaînée de bipointeurs où le champs DATA pointe sur un élément de l?ensemble (ici une structure var_t). Attention : on utilise la même structure de donnée bip_t pour représenter des ensembles de variables, et pour représenter des arbres ABL.

On cherche à écrire la fonction support_abl(), qui prend en entrée un pointeur p sur la racine d?un arbre ABL, et qui renvoie un pointeur sur la liste chaînée des n variables constituant son support.

```
bip_t *support_abl(bip_t *p)
```

- **E.1)** Commencer par écrire la fonction display_varlist() qui affiche sur le terminal standard la liste des noms des variables contenues dans une liste chaînée de bipointeurs, où chaque bipointeur pointe sur un nom de variable. Cette fonction, affiche la liste de noms qu'on lui passe en argument sans aucun traitement.

```
void display_varlist(bip_t *p)
```

Pour éviter de confondre cet ensemble de noms avec une expression Booléenne, on affichera cette liste entre deux accolades de la façon suivante : { a b c d } Valider cette fonction en construisant explicitement dans la fonction main() une liste chainée de variables au moyen des fonctions cons_bip() et consvar(), et affichez cette liste.

- **E.2)** Ecrire la relation de récurrence qui permet d?exprimer le support d?une expression Boléenne, connaissant le support de chacun de ses opérandes.
- **E.3)** Ecrire la fonction union_list() qui effectue l?union de deux ensembles.

```
bip_t *union_list(bip_t *p1, bip_t *p2)
```

Cette fonction prend en entrée deux pointeurs sur deux listes chainées L1 et L2, et renvoie un pointeur sur la liste chaînée représentant l?union des deux listes. Les deux listes L1 et L2 ne doivent pas être modifiés par la fonction union_list(). Attention: l?union est une opération différente de la simple concaténation, car chaque élément ne doit figurer qu?une seule fois dans la liste. On considère que les deux ensembles L1 et L2 qu'on veut réunir respectent cette condition. Valider cette fonction union_list() en construisant explicitement deux listes de variables non disjointes. On affichera les deux listes L1 et L2, ainsi que la liste résultant de l?union de L1 et L2.

- **E.4)** Ecrire effectivement la fonction support_abl(), en veillant à éviter les « fuites de mémoire ». On parle de fuite de mémoire chaque fois qu?un programme fait de l?allocation dynamique de mémoire sans libérer la mémoire qui n?est plus utilisée. Soyez attentifs à libérer la mémoire allouée par la fonction cons_bip() ...
- **E.5**) Intégrez la fonction support_abl() dans le programme main(), et vérifiez que cette fonction calcule correctement le support des trois expressions E0, E1 et E2.

F) Fonction d?évaluation

Pour les plus courageux (ou les plus rapides), on cherche maintenant à écrire une fonction d?évaluation d?une expression Booléenne représentée par un ABL, quand on connaît les valeurs de toutes les variables constituant son support. On ne considère que deux valeurs possibles pour le champs VALUE : 0 et 1.

```
unsigned int eval_abl(bip_t *p)
```

- **F.1)** Rappeler les relations de récurrence qui relient la valeur d?une expression Booléenne à la valeur de ses opérandes, en analysant successivement le cas des 4 opérateurs NOT, OR, AND et XOR.
- **F.2)** Ecrire en langage C la fonction récursive eval_abl(), et valider cette fonction en l?intégrant dans le programme main().

Compte-rendu

Aucun compte-rendu ne vous sera demandé pour ce TME, mais vous devrez présenter une démonstration de votre code au début du prochain TME.