

LO21 – Projet C n°2

Evaluation d'expressions booléennes

```

*** Evaluation d'expressions booleennes ***
Entrer l'expression a evaluer : <<a.!b>+<c.(a+1)>>

Table de verite :
-----
| a b c |
|-----|
| 0 0 0 | 0
| 0 0 1 | 1
| 0 1 0 | 0
| 0 1 1 | 1
| 1 0 0 | 1
| 1 0 1 | 1
| 1 1 0 | 0
| 1 1 1 | 1
-----

Nouvelle expression (o/n) ?

```

```

*** Evaluation d'expressions booleennes ***
Entrer l'expression a evaluer : <a.!b>+<c.(a+1)>

Expression incorrecte
Exemple correct : <<a.!b>+<c.(a+1)>>
Nouvelle expression (o/n) ?

```

Introduction

Le programme à réaliser doit **évaluer des expressions booléennes**. Une expression booléenne est constituée :

- D'**opérateurs** :
 - Binaires : (**OU**, **+**) et (**ET**, **.**)
 - Unaire : (**NON**, **!**)
- De **constantes** : (**0**, **1**)
- De **variables** : (**a...z**, **A...Z**)

Les expressions doivent être **systématiquement parenthésées**.

Exemple : $((a \cdot !b) + (c \cdot (a+1)))$

On représente l'expression booléenne par un **arbre binaire**, et on stocke les variables dans une **liste doublement chaînée**.

Sommaire

<i>Introduction</i>	p.2
<i>Sommaire</i>	p.3
<i>1) Questions de l'énoncé</i>	p.4
1.1) <i>Question 1</i>	
1.2) <i>Question 2</i>	p.5
1.3) <i>Question 3</i>	p.6
1.4) <i>Question 4</i>	p.7
<i>2) Programme</i>	p.10
2.1) <i>Code</i>	
2.2) <i>Analyse du fonctionnement</i>	p.19
<i>Conclusion</i>	p.22

1) Questions de l'énoncé

1.1) Question 1

Donner la définition du type abstrait « arbre » et les profils de l'ensemble des sous-programmes nécessaires à sa manipulation.

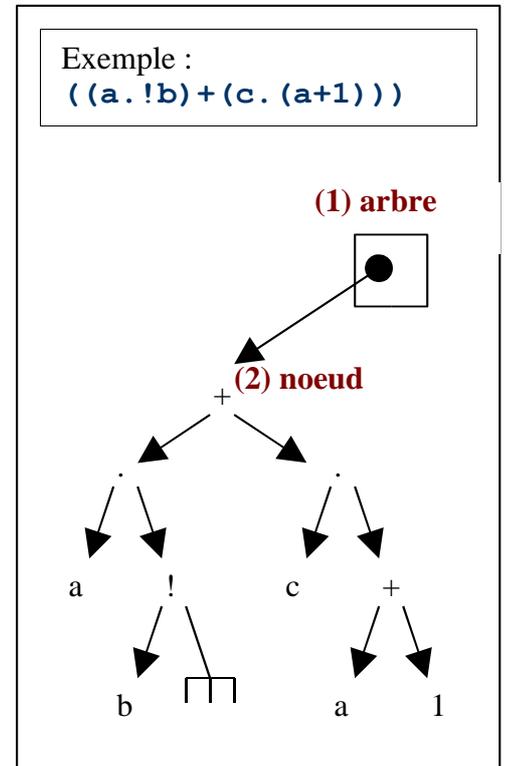
On représente les expressions booléennes par des arbres binaires.

Le type arbre est défini comme un **pointeur vers le premier nœud** (1) :

```
typedef noeud* arbre;
```

Un nœud est quant à lui défini comme **type structuré** composé d'une **valeur** et de **deux pointeurs sur deux arbres**, son « fils gauche » et son « fils droit » (2) :

```
typedef struct noeud{
char valeur;
    char valeur ;
    struct noeud* filsg;
    struct noeud* filsd;
}noeud;
```



On utilise deux types de sous-programmes pour manipuler l'arbre binaire: les constructeurs et les observateurs.

- **Les observateurs :**

```
int Vide ( arbre a );
int Feuille ( arbre a );
char Valeur ( arbre a );
... définis comme habituellement.
```

- **Les constructeurs :**

```
arbre CreerFeuille ( char c );
arbre Enraciner ( arbre a1, arbre a2, char c );
arbre FilsG ( arbre a );
arbre FilsD ( arbre a );
...définis comme habituellement.
```

```
arbre ConstruireArbre( expbool exp, arbre a, probleme
pb );
```

Fonction récursive construisant l'arbre à partir de l'expression booléenne tout en en

vérifiant la syntaxe (si erreur le paramètre **pb** est passé à 1).

```
void Decomposer ( expbool exp, expbool* exp_g, char* op,
expbool* exp_d );
```

Cette procédure récursive décompose l'expression booléenne **exp** en comptant les parenthèses ouvrantes. Elle renvoie l'opérateur central et deux nouvelles expressions booléennes (l'expression gauche et l'expression droite).

1.2) Question 2

Donner l'algorithme permettant de construire un arbre à partir d'une chaîne de caractères. Cette étape doit prendre en compte la vérification de la chaîne entrée. Attention, mis à part la sous-arborescence correspondant à la sous-expression incorrecte, le reste de l'arbre doit être construit.

La fonction récursive **ConstruireArbre** fait appel à trois autres sous-programmes :

- **DonnerNature** qui renvoie le type d'un caractère parmi une liste prédéfinie de types acceptés dans les expressions booléennes (variable, constante, opérateur binaire - ou/et -, parenthèses, point d'exclamation ou type inconnu).
- **Décomposer** et **Enraciner** (cf. 1.1).

Synopsis de **ConstruireArbre** :

<i>Données :</i>	<i>Résultat :</i>
- <i>expression booléenne</i>	- <i>arbre a</i>
	- <i>probleme pb</i>

L'algorithme de la fonction peut être résumé ainsi :

Cas trivial :

L'expression booléenne est une constante (**0, 1**) ou une variable (**A...Z, a...z**). Dans ce cas **ConstruireArbre** renvoie une feuille à laquelle correspondra la valeur de la variable ou de la constante.

Cas général :

Il y a deux possibilités :

- Le premier caractère est un point d'exclamation. Dans ce cas on l'enracine comme noeud avec pour fils **NULL** et l'appel récursif à **ConstruireArbre** du reste de l'expression.
- Le premier caractère est une parenthèse. Il faut alors utiliser **Décomposer** pour pouvoir enraciner l'opérateur central de l'expression avec pour fils l'appel récursif à **ConstruireArbre** des expressions gauche et droite.

Codé en C, l'algorithme devient :

```

arbre ConstruireArbre ( expbool exp, arbre a, probleme pb ){

    int longueur=strlen(exp);

    if (( (DonnerNature(exp[0]) == var) || (DonnerNature(exp[0]) == cte))
    \
        && (exp[1] == '\0') ){
        return Enraciner( NULL, NULL, exp[0]);
    }
    else if ( (DonnerNature(exp[0]) == op_un) ){
        int i;
        expbool temp;
        for ( i=0; i<longueur; i++ ){
            temp[i] = exp[i+1];
        }
        temp[i+1] = '\0';
        return Enraciner(NULL, ConstruireArbre(temp, a, pb), '!');
    }
    else if (DonnerNature(exp[0]) == par_ouv){
        char temp_op;
        expbool temp_g, temp_d;
        Decomposer (exp, &temp_g, &temp_op, &temp_d );
        return Enraciner ( ConstruireArbre ( temp_g, a, pb ), \
                            ConstruireArbre ( temp_d, a, pb ), \
                            temp_op );
    }
    *pb = 1;
    return NULL;
}
    
```

Si aucun des **if** n'a été exécuté, c'est que l'expression est incorrecte : **ConstruireArbre** renvoie un arbre vide et la variable **pb** est passée à **1** pour signifier que l'expression est incorrecte.

1.3) Question 3

Donner l'algorithme permettant d'évaluer l'expression booléenne pour une instantiation des variables.

C'est la fonction récursive **Evaluer** qui remplit ce rôle. Son synopsis est le suivant :

Données :

- liste *l* doublement chaînée contenant les variables de l'expression
- arbre *a* contenant l'expression
- tableau *vals* d'entiers contenant les valeurs des variables avec lesquelles évaluer l'expression

Résultat :

- 0 ou 1

L'algorithme est le suivant :

Cas trivial :

L'arbre **a** est une feuille. Dans ce cas si la valeur de **a** est une variable (**a...z, A...Z**), on

renverra la valeur correspondant à cette variable dans vals. Sinon, c'est une constante (0, 1), il suffit de la renvoyer.

Cas général :

La racine de l'arbre est un opérateur :

- Binaire : on renvoie le résultat de l'opération correspondante (ou/et) entre les appels récursif à **Evaluer** des deux fils
- Unaire : on renvoie la négation du résultat de l'appel récursif à **Evaluer** du fils droit

En C :

```
int Evaluer ( liste l, arbre a, int vals[] ){
    int i=0 ;
    int eval;
    if ( FilsD(a)==NULL && FilsG(a)==NULL ){
        if (DonnerNature(Valeur(a))==var){
            liste tmp = l;
            while (tmp->valeur != Valeur(a)){
                i=i+1;
                tmp=tmp->suiwant;
            }
            eval = vals[i];
        }
        else if (Valeur(a)=='0'){
            eval = 0;
        }
        else{
            eval = 1;
        }
    }
    else if ( Valeur(a)=='.' ){
        eval = ( Evaluer(l,FilsG(a),vals) && Evaluer(l,FilsD(a),vals) );
    }
    else if ( Valeur(a)=='+' ){
        eval = ( Evaluer(l,FilsG(a),vals) || Evaluer(l,FilsD(a),vals) );
    }
    else{
        eval = !Evaluer(l,FilsD(a),vals);
    }
    return(eval);
}
```

1.4) Question 4

Ecrire le programme général permettant, à partir de la saisie d'une chaîne correspondant à une expression booléenne, d'évaluer celle-ci et d'afficher le résultat sous forme d'une table.

C'est la procédure d'affichage **ConstruireTable** qui se charge de donner la table de vérité de l'expression en fonction des valeurs des variables.

Son synopsis :

Données :

- arbre *a* contenant l'expression
- liste *l* doublement chaînée contenant les variables de l'expression

Résultat :

- affichage de la table de vérité

Cette procédure affiche le résultat de l'expression pour chaque combinaison de valeurs des variables. Ces valeurs forment un nombre en base 2, stocké dans **vals**.

La procédure opère ligne par ligne en partant du cas où toutes les variables sont à **0**. Elle effectue une addition binaire (on ajoute de **1**, en tenant compte des retenues) pour passer à la combinaison suivante. Pour chaque combinaison de valeurs, elle effectue le calcul de l'expression à l'aide la fonction **Evaluer**, et affiche le résultat.

On a alors évalué toutes les combinaisons possibles. Au total il y en a 2 puissance le nombre de variables. Il est donc nécessaire de coder une fonction **Puissance**, celle-ci n'étant pas dans les bibliothèques standard. En effet, intégrer une bibliothèque complète pour une seule fonction est inutile.

En C :

```

void ConstruireTable(liste l , arbre a){

    int i, j;
    int vars[TAI];
    int nb_vars = 0;
    liste pt = l;

    while ( pt != NULL ){
        nb_vars++;
        pt=pt->suitant;
    }

    pt = l;
    while ( pt != NULL ){
        printf("%c",pt->valeur);
        pt=pt->suitant;
    }
    printf("\n") ;

    for ( i=0; i<nb_vars; i++ ){
        vars[i]=0;
        printf("%d", vars[i]);
    }

    int eval;
    int nb_eval=0;

    eval = Evaluer ( l , a, vars );
    printf("%d\n",eval);

    int ret;

    do{

        vars[nb_vars-1] = vars[nb_vars-1] + 1;

```

```
i=nb_vars-1;

do{

    if ( vars[i]>1 ){
        ret=1;
        vars[i]=0;
        vars[i-1]=vars[i-1]+1;
        i=i-1;
    }
    else{
        ret=0;
    }

}while( (i>=0) && (ret==1) );

for( j=0; j<nb_vars; j++ ){
    printf("%d",vars[j]);
}

eval = Evaluer ( l , a, vars );
printf("%d\n",eval);

nb_eval = nb_eval+1;

}while ( nb_eval<Puissance(2,nb_vars)-1 );
}
```

Nota :

Le code ci-dessus a été allégé des instructions destinées à améliorer l'affichage, ceci pour faciliter la compréhension.

2) Programme

2.1) Code

```

// projet2.c
//
// HEROUARD Brice (brice.herouard@utbm.fr)
// COUVIDOU Laurent (laurent.couvidou@utbm.fr)
//
// ÉVALUATION D'EXPRESSIONS BOOLÉENNES
// Composition d'une expression booléenne :
// Opérateurs binaires :
// + = OU
// . = ET
// Opérateur unaire :
// ! = NON
// Variables :
// (a,b,...,z)
// (A,B,...,Z)
// Constantes :
// 0 = FAUX
// 1 = VRAI

#include <stdio.h> // Entrées-sorties standard
#include <stdlib.h> // Fonctions standard
#include <string.h> // Traitement des chaînes
#include <conio.h> // Affichage sous dos

#define TAI 50 // Taille des chaînes et des tableaux

/***** DECLARATIONS DE TYPES *****/
/***** Type probleme *****/
typedef int* probleme;

/***** Type expbool *****/
typedef char expbool[TAI];

/***** Type nature *****/
typedef enum{
    var, cte, op_un, op_bin, par_ouv, par_fer, inconnu
}nature;

/***** Type noeud *****/
typedef struct noeu{
    char valeur;
    struct noeu* filsg; // pointeur vers le fils gauche
    struct noeu* filsd; // pointeur vers le fils droit
}noeud;

/***** Type arbre *****/
typedef noeud* arbre; // pointeur vers le premier noeud

/***** Type element *****/
typedef struct elem{
    char valeur;
    struct elem* suivant;
    
```

```

        struct elem* precedent;
    }element;

    /***** Type liste *****/
    typedef element* liste;

    /***** DECLARATIONS DE FONCTIONS *****/

    /***** Observateurs de l'arbre binaire *****/
    int Vide ( arbre a );
    // Teste si a est vide

    int Feuille ( arbre a );
    // Teste si a est une feuille, c'est-à-dire si
    // - a est un noeud (soit a non vide)
    // - a n'a ni fils gauche ni fils droit

    char Valeur ( arbre a );
    // Donne la valeur de la racine de a

    /***** Constructeurs de l'arbre binaire *****/
    arbre CreerFeuille ( char c );
    // Crée un arbre d'une feuille contenant le caractère c

    arbre Enraciner ( arbre a1, arbre a2, char c );
    // Retourne les arbres a1 et a2 liés par une racine contenant c

    arbre FilsG ( arbre a );
    // Retourne le fils gauche de a

    arbre FilsD ( arbre a );
    // Retourne le fils droit de a

    void Decomposer ( expbool exp, expbool* exp_g, char* op, expbool* exp_d );
    // Donne la partie gauche, la partie droite et l'opérateur d'une expression
    // Utilisée dans ConstruireArbre

    arbre ConstruireArbre ( expbool exp, arbre a, probleme pb );
    // Construit l'arbre binaire par récursivité
    // La variable pb se charge de signaler les éventuels problèmes

    /***** Nature d'un caractère *****/
    nature DonnerNature ( char lettre );
    // Détermine si un caractère est une variable, une constante, un
    // opérateur...

    /***** Construction de la liste *****/
    int EstPresent ( liste l, char c );
    // Teste la présence d'un élément c dans une liste l

    liste InsérerElement ( liste l, char c );
    // Insère l'élément c dans la liste l
    // Respecte l'ordre alphabétique
    
```

```

liste ConstruireListe ( expbool exp );
// Renvoie une liste construite à partir de l'expression booléenne exp

/***** Évaluation de l'expression booléenne *****/

int Evaluer ( liste l, arbre a, int vals[] );
// Évalue l'expression contenue dans a, pour les vvariables contenues
// dans l prenant leurs valeurs dans vals

void ConstruireTable ( liste l, arbre a );
// Affiche la table de vérité correspondant à l'expression contenue dans a,
// pour les variables contenues dans l

/***** Divers *****/

int Puissance(int n, int x);
// Donne la puissance n de x

/***** PROGRAMME PRINCIPAL *****/

int main(){

    char encore;

    do{

        clrscr();

        printf("*** Evaluation d'expressions booleennes ***\n\n");

        probleme pb = (probleme)malloc(sizeof(int));
        *pb = 0; // La valeur sera changée par ConstruireArbre si problème

        arbre a;
        a = (noeud*)malloc(sizeof(noeud));

        expbool exp; // Reçoit l'expression booléenne entrée

        printf("Entrer l'expression a evaluer : "); // Invite
        scanf("%s",exp);
        getchar(); // "Mange" le caractère retour à la ligne

        a = ConstruireArbre ( exp, a, pb ); // Construction de l'arbre

        if ( *pb == 0 ){ // Expression correcte

            printf("\n\nTable de verite :\n");
            liste l; // La liste des variables
            l = ConstruireListe(exp);
            ConstruireTable(l , a); // Affichage de la table de vérité
        }

        else{ // Expression incorrecte
            printf("\nExpression incorrecte");
            printf("\nExemple correct : ((a.!b+(c.(a+1)))");
        }

        printf("\nNouvelle expression (o/n) ? ");
        encore=getchar();
        getchar(); // "Mange" le caractère retour à la ligne
    }
}

```

```

    }while ( ( encore == 'o' ) || ( encore == 'O' ) );

    return 0; // Valeur de retour : le programme s'est terminé normalement
}

/***** DEFINITIONS DE FONCTIONS *****/

/***** Observateurs de l'arbre binaire *****/

int Vide ( arbre a ){
    return ( a==NULL );
}

int Feuille ( arbre a ){
    return ( (!Vide(a)) && (Vide(FilsG(a)) && (Vide(FilsD(a))) ) );
}

char Valeur ( arbre a ){
    return a->valeur;
}

/***** Constructeurs de l'arbre binaire *****/

arbre CreerFeuille ( char c ){
    arbre nouv;
    nouv = (arbre)malloc(sizeof(noead));
    nouv->valeur = c;
    nouv->filsg = NULL;
    nouv->filsd = NULL;
    return nouv;
}

arbre Enraciner ( arbre a1, arbre a2, char c ){
    arbre nouv;
    nouv = CreerFeuille(c);
    nouv->filsg = a1;
    nouv->filsd = a2;
    return nouv;
}

arbre FilsG ( arbre a ){
    return a->filsg;
}

arbre FilsD ( arbre a ){
    return a->filsd;
}

void Decomposer ( expbool exp, expbool *exp_g, char *op, expbool *exp_d ){

    int i=1, j=0; // Compteurs
    int position; // Position de l'opérateur
    int longueur=strlen(exp); // Longueur de l'expression
    expbool temp_g, temp_d; // Variables temporaires, pour le traitement
    int nb_par_ouv = 1, nb_op = 0; // Nombre de parenthèses et d'opérateurs

    while ( (nb_par_ouv != nb_op) && (i < longueur) ){

        if (DonnerNature(exp[i]) == par_ouv){

```

```

        nb_par_ouv++;
    }
    else if (DonnerNature(exp[i]) == op_bin){
        nb_op++;
    }
    i++;
} // On recherche la position de l'opérateur

position = i-1;

for (i = 0 ; i < position ; i++){
    temp_g[i] = exp[i+1];
}
temp_g[position-1] = '\0';
strcpy(*exp_g,temp_g);
// Extrait l'expression gauche

for (i = position ; i < longueur - 2 ; i++){
    temp_d[j] = exp[i+1];j++;
}
temp_d[j] = '\0';
strcpy(*exp_d,temp_d);
// Extrait l'expression droite

*op = exp[position];
// Extrait l'opérateur
}

arbre ConstruireArbre ( expbool exp, arbre a, probleme pb ){

    int longueur=strlen(exp); // Longueur de l'expression

    if (( (DonnerNature(exp[0]) == var) || (DonnerNature(exp[0]) == cte)) \
        && (exp[1] == '\0') ){

        return Enraciner( NULL, NULL, exp[0]);
    } // Variable ou constante : on crée une feuille

    else if ( (DonnerNature(exp[0]) == op_un) ){

        int i;
        expbool temp;
        for ( i=0; i<longueur; i++ ){
            temp[i] = exp[i+1];
        }
        temp[i+1] = '\0';
        return Enraciner(NULL,ConstruireArbre(temp,a,pb),'!');
    } // Négation : un fils droit seul enraciné sur l'opérateur

    else if ( DonnerNature(exp[0]) == par_ouv ){

        char temp_op;
        expbool temp_g, temp_d;
        Decomposer (exp, &temp_g, &temp_op, &temp_d );
        return Enraciner ( ConstruireArbre ( temp_g, a, pb ), \
                            ConstruireArbre ( temp_d, a, pb ), \
                            temp_op );
    } // Autres cas : fils droit et fils gauche, enracinés sur
      // l'opérateur

    *pb = 1; // Problème : un caractère est non conforme
    return NULL; // On renvoie un arbre vide
}

```

```

/***** Nature d'un caractère *****/
nature DonnerNature ( char lettre ){

    nature temp = inconnu; // Caractère inconnu au départ

    // On cherche si le caractère est connu :

    if ((lettre >= 'A') && (lettre <= 'z')){ // (A,B,...,Z) et (a,b,...z)
        temp = var;
    }
    if ((lettre == '0') || (lettre == '1')){ // (0 = FAUX) et (1 = VRAI)
        temp = cte;
    }
    if (lettre == '!'){ // (! = NON)
        temp = op_un;
    }
    if ((lettre == '+') || (lettre == '.')){ // (+ = OU ) et (. = ET)
        temp = op_bin;
    }
    if (lettre == '('){ // Parenthèse ouvrante
        temp = par_ouv;
    }
    if (lettre == ')'){ // Parenthèse fermante
        temp = par_fer;
    }

    return temp;
}

/***** Construction de la liste *****/
int EstPresent ( liste l, char c ){

    int present = 0;

    if (l != NULL){
        while ((l != NULL) && (present == 0)){
            if (l->valeur == c){
                present = 1;
            }
            l = l->suivant;
        }
    }

    return present;
}

liste InsereElement ( liste l, char val ){

    liste pt = l; // Pointeur de parcours de la liste

    liste temp; // Pointeur temporaire
    temp = (element*)malloc(sizeof(element));
    temp->valeur = val;

    if (pt == NULL){ // La liste est vide : élément unique

        temp->precedent = NULL;
        temp->suivant = NULL;
    }
}

```

```

        l = temp;
    }

    else{

        while ((pt->suitant != NULL) && (val > pt->valeur)){
            pt = pt->suitant;
        }
        // On avance, tant que le code ASCII du caractère est supérieur à
        // celui rencontré (nb : la table est ASCII dans l'ordre
        // alphabétique)

        if (l == pt){ // Liste d'1 élément

            if (val < pt->valeur){ // Le caractère se place avant
                temp->precedent = NULL;
                temp->suitant = pt;
                pt->precedent=temp;
                l=temp;
            }

            else{ // Le caractère se place après
                l->suitant = temp;
                temp->precedent = l;
                temp->suitant = NULL;
            }
        }

        else if (pt->suitant == NULL){ // Plus grand que tous sauf
            // dernier

            if (val < pt->valeur){ // Avant dernier caractère

                (pt->precedent)->suitant = temp;
                temp->precedent = pt->precedent;
                temp->suitant = pt;
                pt->precedent= temp;
            }

            else{ // Dernier caractère

                pt->suitant = temp;
                temp->precedent = pt;
                temp->suitant = NULL;
            }
        }

        else{ // Cas général

            (pt->precedent)->suitant = temp;
            temp->precedent = pt->precedent;
            temp->suitant = pt ;
            pt->precedent= temp;
        }
    }

    return l;
}

liste ConstruireListe ( expbool exp ){

    int longueur=strlen(exp); // Longueur de l'expression

```

```

int i;
liste l = NULL;

for (i = 0 ; i < longueur ; i++){
    if ( !EstPresent(l,exp[i]) && (DonnerNature(exp[i]) == var) ){
        l = InsérerElement( l, exp[i] );
    }
}
// On n'insère dans l que les variables, qui n'y sont pas déjà
// présentes

return l;
}

/***** Évaluation de l'expression booléenne *****/

int Evaluer ( liste l, arbre a, int vals[] ){

    int i=0; // Compteur
    int eval; // Résultat de l'expression évaluée :
                // (0 = FAUX) ou (1 = VRAI)

    if ( FilsD(a)==NULL && FilsG(a)==NULL ){ // Variable ou constante

        if (DonnerNature(Valeur(a))==var){
            liste tmp = l;
            while (tmp->valeur != Valeur(a)){
                i=i+1;
                tmp=tmp->suitant;
            }
            eval = vals[i]; // On récupère sa valeur
        }

        else if (Valeur(a)=='0'){ // FAUX
            eval = 0;
        }

        else{ // VRAI
            eval = 1;
        }
    }

    else if ( Valeur(a)=='.' ){ // "ET" logique
        eval = (Evaluer(l,FilsG(a),vals) && Evaluer(l,FilsD(a),vals));
    }

    else if ( Valeur(a)=='+' ){ // "OU" logique
        eval=(Evaluer(l,FilsG(a),vals) || Evaluer(l,FilsD(a),vals));
    }

    else{ // "NON" logique
        eval = !Evaluer(l,FilsD(a),vals);
    }

    return(eval);
}

void ConstruireTable(liste l , arbre a){

    int i, j; // Compteur

    int vars[TAI]; // Ligne contenant les valeurs des variables

```

```

int nb_vars = 0; // Nombre de variables

liste pt = l; // Pointeur de parcours de la liste

while ( pt != NULL ){
    nb_vars++;
    pt=pt->suitivant;
} // Compte le nombre d'éléments de l (à l'aide de pt)

printf(" ");
for ( i=1; i<=(2*nb_vars)+4; i++ ){
    printf("-");
}
printf("\n"); // Pour l'affichage

pt = l;

while ( pt != NULL ){
    printf(" %c",pt->valeur);
    pt=pt->suitivant;
} // Affiche chaque variable de l (toujours à l'aide de pt)

printf("    |\n");
for ( i=1; i<=(2*nb_vars)+4; i++ ){
    printf(" ");
}
printf("|\n"); // Pour l'affichage

for ( i=0; i<nb_vars; i++ ){
    vars[i]=0;
    printf("% d", vars[i]);
} // Initialise chaque case de vars à 0, et affiche cette lère ligne

int eval; // Résultat de l'évaluation d'une ligne
int nb_eval=0; // Nombre d'évaluations

eval = Evaluer ( l , a, vars ); // Évalue la lère ligne
printf(" %d |\n",eval); // L'affiche

int ret; // Retenue (cf ci-dessous)

do{ // Boucle 1 : Traite chaque ligne

    vars[nb_vars-1] = vars[nb_vars-1] + 1; // + 1 à la ligne en
                                           // cours
    i=nb_vars-1; // Initialisation du compteur

    do{ // Boucle 2 : Gère les retenues

        if ( vars[i]>1 ){ // Si on a dépassé 1 dans une
                        // variable :
            ret=1; // Il y a retenue
            vars[i]=0; // Cette variable passe à 0
            vars[i-1]=vars[i-1]+1; // Et la précédente augmente
                                // de 1
            i=i-1;
        }
        else{
            ret=0; // Pas de retenue
        }
    } while( (i>=0) && (ret==1) ); // On continue tant qu'il y a
                                // retenue
    
```

```

printf("|"); // Pour l'affichage

for( j=0; j<nb_vars; j++ ){
    printf(" %d",vars[j]);
} // Affiche les valeurs données aux variables pour cette ligne

eval = Evaluer ( l , a, vars ); // Évalue la ligne
printf(" %d |",eval); // Affiche l'évaluation
printf("\n");

nb_eval = nb_eval+1; // On a fait une évaluation

}while ( nb_eval<Puissance(2,nb_vars)-1 );
// Pour n variables, on aura 2^n évaluations à faire, moins une car on
// a déjà évalué le cas où toutes les variable sont nulles.

printf(" ");
for ( i=1; i<=(nb_vars*2)+4; i++ ){
    printf("-");
}
printf("\n"); // Pour l'affichage
}

/***** Divers
*****/

int Puissance(int x, int n){

    int res;

    if (n==0){
        res=1;
    }
    else{
        res=x*Puissance(x,n-1);
    }

    return res;
}

```

2.2) Analyse du fonctionnement

Vérifions d'abord que le programme n'est pas « mis en échec » par une expression simple, mais juste (**variable** ou **constante**) :

- Variable (**a...z, A...Z**) :
 > Pas de problème

Entrer l'expression à évaluer : G

Table de verite :

G	
0	0
1	1

```
Entrer l'expression a evaluer : 0
```

```
Table de verite :
```

```

-----
|  |  |  |
| 0 |  |  |
| 0 3998928 0 |
| 1 0 0 |
| 1 1 0 |
|  |  |  |
-----

```

- Constante (0,1) :

> **Problème**

On a une **erreur** en entrant une constante. C'était une erreur prévisible : tel qu'il est construit, le programme ne peut fonctionner qu'avec une ou plusieurs variables. En effet, la procédure **ConstruireTable** (cf 1.4) et la fonction **Evaluer** (cf 1.3) qu'elle appelle demandent toutes deux la liste **l** des variables pour fonctionner.

Le programme fonctionne-t-il pour les **opérations de base** (c'est-à-dire : les tables de vérité sont-elles justes) ?

```
Entrer l'expression a evaluer : (a+b)
```

```
Table de verite :
```

```

-----
| a b |  | |
| 0 0 | 0 |
| 0 1 | 1 |
| 1 0 | 1 |
| 1 1 | 1 |
|  |  |  |
-----

```

- Opération binaire (OU, +) :

> Pas de problème

```
Entrer l'expression a evaluer : (a.b)
```

```
Table de verite :
```

```

-----
| a b |  | |
| 0 0 | 0 |
| 0 1 | 0 |
| 1 0 | 0 |
| 1 1 | 1 |
|  |  |  |
-----

```

- Opération binaire (ET, .) :

> Pas de problème

```
Entrer l'expression a evaluer : !a
```

```
Table de verite :
```

```
-----
| a |
| 0 | 1 |
| 1 | 0 |
-----
```

- Opération unaire (**NON, !**) :

> Pas de problème

Testons enfin le programme pour une **expression complète**. Prenons par exemple celle de l'énoncé, dont on a déjà la table de vérité :

```
*** Evaluation d'expressions booleennes ***
Entrer l'expression a evaluer : <<a.!b>+<c.<a+1>>

Table de verite :
-----
| a b c |
| 0 0 0 | 0 |
| 0 0 1 | 1 |
| 0 1 0 | 0 |
| 0 1 1 | 1 |
| 1 0 0 | 1 |
| 1 0 1 | 1 |
| 1 1 0 | 0 |
| 1 1 1 | 1 |
-----

Nouvelle expression <o/n> ?
```

On obtient bien le **même résultat** que celui présenté dans l'énoncé. Il faut noter que, sans les parenthèses extérieures, le programme indique une erreur de syntaxe :

```
*** Evaluation d'expressions booleennes ***
Entrer l'expression a evaluer : <a.!b>+<c.<a+1>>

Expression incorrecte
Exemple correct : <<a.!b>+<c.<a+1>>
Nouvelle expression <o/n> ?
```

C'est normal : sans ces parenthèses, la fonction **ConstruireArbre** (cf. 1.2) ne peut pas prendre en compte l'opérateur « du milieu ».

Conclusion

Le programme écrit **réponds aux demandes de l'énoncé** (cf 2.2).

Une demande n'a par contre pas été implémentée : l'arbre n'est pas construit lorsque l'expression est incorrecte, et la table de vérité non plus. En effet, il faudrait, par exemple, remplacer arbitrairement la sous-expression incorrecte par une variable. Mais, avec cette solution comme avec une autre, le programme ne répondrais pas aux attentes de l'utilisateur.

On pourrait aussi lui apporter **quelques améliorations**, au-delà des demandes de l'énoncé :

- Avertir l'utilisateur par un message d'erreur s'il entre uniquement une constante, ou établir une « table de vérité » particulière lorsque l'expression se réduit à '0' ou '1'.
- Prendre en compte les priorités des opérateurs. Soit, dans l'ordre de priorité croissant :
(OU, +) < (ET, .) < (NON, !)
Le programme pourrait ainsi prendre en compte de véritables expressions booléennes. La contrainte des parenthèses donne en effet un programme lourd à manipuler.
- Proposer une correction lorsque l'utilisateur entre une expression incorrecte. Ce procédé semble plus cohérent qu'une évaluation incomplète de l'expression. Le programme se contente ici d'indiquer un exemple correct (celui de l'énoncé), qui peut servir de guide.