Evaluation d'expressions booléennes

Le but de ce projet est de construire, à partir d'une chaîne de caractères représentant une expression booléenne, l'arbre binaire et de l'évaluer pour obtenir la table de vérité correspondant à la chaîne entrée par l'utilisateur.

1- Définition du type abstrait « arbre » et des sous-programmes élémentaires

a) Définition

```
typedef struct elem
        {
        char cara[1] ;
        struct elem * filsg ;
        struct elem * filsd ;
        } noeud ;

typedef struct noeud * arbre ;
```

Remarque : « cara » est un tableau de caractères associé à chaque nœud de l'arbre ; « filsg » pointe vers le fils gauche d'un nœud ; « filsd » pointe vers le fils droit d'un nœud. Dans le ***compilateur lcc***, la dernière ligne doit être transformée en : « typedef nœud * arbre ; » sinon ça ne marche pas.

b) Fonctions élémentaires

```
arbre Creer()
        {
        return (NULL) ;
        }
```

Remarque : cette fonction crée un arbre binaire vide.

```
arbre CreerFeuille(char x)
        {
        arbre a = (arbre) malloc (sizeof (nœud)) ;
        a->filsg = NULL ;
        a->filsd = NULL ;
        a->cara = x ;
        return (a) ;
        }
```

Remarque : cette fonction crée un arbre qui n'a pas de feuille.

```
arbre Enraciner(arbre a, arbre b, arbre c)
        {
        a->filsg = b ;
        a->filsd = c ;
        return (a) ;
        }
```

Remarque : cette fonction construit un arbre binaire « a » à partir de deux arbres « b » et « c ».

```
arbre FilsGauche(arbre a)
        {
        if (a->filsg == NULL)
                return (NULL) ;
        else return (a->filsg) ;
        }
```

Remarque : cette fonction donne l'arbre gauche d'un arbre donné.

```
arbre FilsDroit(arbre a)
        {
        if (a->filsd == NULL)
                return (NULL) ;
        else return (a->filsd) ;
        }
```

Remarque : cette fonction donne l'arbre droit d'un arbre donné.

```
int Vide(arbre a)
        {
        if (a == NULL)
                return (1) ;
        else return (0) ;
        }
```

Remarque : cette fonction renvoie 1 si l'arbre "a" est vide ; 0 sinon.

```
int Feuille(arbre a)
        {
        if (a->filsg == NULL && a->filsd == NULL)
                return (1) ;
        else return (0) ;
        }
```

Remarque : cette fonction renvoie 1 si l'arbre "a" est une feuille ; 0 sinon.

```
char Racine(arbre a)
        {
        return (a->cara) ;
        }
```

Remarque : cette fonction donne la chaîne de la racine de l'arbre.

2- Algorithme de construction de l'arbre à partir d'une chaîne de caractères (construction_arbre)

***Données** : chaine, chaîne de caractères à évaluer (construire) ; debut et fin des entiers représentant les index de début et fin de la chaine.*
***Résultats** : a, arbre binaire représentant la chaine à évaluer.*
***Privées** : arret, i compteurs entiers ; nbouv, nbferm entiers représentant le nombre de parenthèses ouvrantes et fermantes.*

```
Début
    arret <- 0
    i <- debut
    nbouv <- 0
    nbferm <- 0
    si (fin = debut) alors a <- CreerFeuille (chaine[i])
    sinon
       tant que (arret = 0 et i <= fin) faire
            si (chaine[i] = '(') alors nbouv <- nbouv + 1
            finsi
            si (chaine[i] = ')') alors nbferm <- nbferm + 1
            finsi
            si (chaine[i] = '.') alors si (nbouv – nbferm = 1) alors
                        arret <- 1
                        a <- Enraciner (construction_arbre (chaine, debut+1 , i-
                        1), construction_arbre (chaine, i+1 , fin-1 ), CreerFeuille
                        ('.'))
                  finsi
            finsi
            si (chaine[i] = '+') alors si (nbouv – nbferm = 1) alors
                        arret <- 1
                        a <- Enraciner (construction_arbre (chaine , debut+1 , i-
                        1), construction_arbre (chaine , i+1 , fin-1), CreerFeuille
                        ('+'))
                  finsi
            finsi
            si (chaine[i] = '!') alors si (nbouv – nbferm <> 2) alors
                        arret <- 1
                        a <- Enraciner (construction_arbre (chaine , i+1 , fin-1),
                        Creer(),CreerFeuille('!'))
                  finsi
            finsi
            i <- i + 1
    fintantque
    finsi
Fin
```

Remarque : cet algorithme (fonction) est effectué de manière récursive.

3- Algorithme d'évaluation de l'arbre construit (evaluer)

***Données** : p, liste chaînée créée à partir de la chaîne entrée par l'utilisateur ; b arbre à évaluer.*
***Résultats** : result, résultat de l'évaluation.*

```
Début
    j <- p
    si (b -> filsd = NULL et b -> filsg =NULL)
    alors   si (b -> cara = '1')
            alors result <- 1
            finsi
            si (b -> cara = '0')
            alors result <- 0
            sinon   tant que (b -> cara <> j -> var) faire
                            j <- j -> suivant
                    fintantque
                    result <- j -> valeur
            finsi
    sinon   si (b -> cara = '+')
            alors result <- evaluer (b -> filsd , p) ou evaluer ( b-> filsg , p)
            finsi
            si (b -> cara ='.')
            alors result <- evaluer (b -> filsg , p) et evaluer (b -> filsd , p)
            finsi
            si (b -> cara ='!')
            alors   si (b -> filsg <> NULL)
                    alors result <- ! (evaluer (b -> filsg , p))
                    finsi
            finsi
    finsi
Fin
```

Remarque : cet algorithme (fonction) s'effectue de manière récursive ; il fait appel à plusieurs fonctions décrites ultérieurement.

4- Programme général en C permettant de construire un arbre à partir d'expressions booléennes et de l'évaluer par la suite

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct element
      {
      char cara ;
      struct element *filsd;
      struct element *filsg;
      } noeud ;
typedef noeud  * arbre;

typedef struct elem
      {
      char var  ;
      int valeur ;
      struct elem *suivant ;
      } elemlist ;
typedef elemlist * list  ;

arbre creer()
      {
      return  NULL;
      }

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


arbre creerf(char h)
      {
      arbre p;
      p = (arbre)malloc(sizeof (noeud));
      p->cara = h;
      p->filsg = NULL;
      p->filsd = NULL;
      return p;
      }

arbre enraciner (arbre a1, arbre a2, arbre a)
      {
      a ->filsg = a1;
      a ->filsd = a2;
      return a;
      }
```

```
arbre filsga(arbre a)
      {
       if (a != NULL)
       return (a ->filsg);
       else return a;
       }

arbre filsdr(arbre a)
       {
       if (a != NULL)
       return (a ->filsd);
       else return a;
       }

char racine (arbre a)
       {
       return (a ->cara);
       }

arbre construction_arbre(char *chaine, int debut, int fin)
       {
       int arret, i, nbouv, nbferm;
       arret = 0;
       i = debut;
       nbouv = 0;
       nbferm = 0;
       if (fin == debut) return creerf(chaine[i]);
       else
             {
             while ((arret == 0) && (i <=fin))
                   {
                   switch (chaine[i])
                         {
                         case '(' :
                               nbouv++;
                         break;
                         case ')' :
                               nbferm++;
                         break;
                         case '.' :
                               if (nbouv - nbferm == 1)
                                     {
                                     arret = 1;
                                     return enraciner(construction_arbre
                                     (chaine,debut+1,i-1) , construction_arbre
                                     (chaine,i+1,fin-1), creerf('.'));
                                     }
                         break;
                         case '+' :
                               if (nbouv - nbferm == 1)
                                     {
                                     arret = 1;
                                     return enraciner(construction_arbre
                                     (chaine,debut+1, i-1),
                                     construction_arbre(chaine,i+1,fin-1), creerf('+'));
                                     }
```

```
                        break;
                        case '!' :
                              if (nbouv - nbferm != 2)
                                    {
                                    arret = 1;
                                    return enraciner(construction_arbre
                                    (chaine,i+1,fin-1), creer(),creerf('!'));
                                    }
                        break;
                        }
                  i++;
                  }
      }
}

list newelem(char *cara)
      {
      list p;
      p = (list) malloc (sizeof(elemlist));
      p ->var = cara[0];
      p ->suivant = NULL;
      return p;
      }

list ajouterqueu (list l,char *cara)
      {
      list p;
      if (l == NULL) return newelem(cara);
      else
            {
            p=l;
            while (p ->suivant != NULL)
                  {
                  p = p ->suivant;
                  }
            p ->suivant =newelem(cara);
            return l;
            }
      }


list creationlist (char *chaine)
      {
      int t = strlen (chaine) ;
      list p,j;
      int trouver =0;
      p=NULL  ;
      for ( int i=0 ;i<t;i++)
            {
            if ( chaine[i]!='(' && chaine[i]!=')' && chaine[i]!='+' &&  chaine[i]!='.'
            && chaine[i]!='(' && chaine[i]!='!' && chaine[i]!='1' && chaine[i]!='0')
                  {
                  j = p ;
                  while (j != NULL  )
                        {
                        if (chaine[i] == p->var ) trouver  = 1 ;
```

```
                        j = j -> suivant ;
                        }
                if (trouver != 1 ) p=ajouterqueu(p ,&chaine[i] ) ;
                }
        }
 return p ;
}


int  evaluer (arbre b ,list p)
{
list j=p ;
int res ;
if (b->filsd==NULL  && b->filsg==NULL )
        {
        if (b->cara=='1')
                {
                return 1 ;
                }
        if (b->cara=='0')
                {
                return 0 ;
                }
                else
                {
                while (b->cara!=j->var)
                        {
                        j = j->suivant ;
                        }
                return j->valeur ;
                }
        }
        else
                {
                if (b->cara =='+')
                        {
                        return (evaluer(b->filsd,p)|evaluer(b->filsg,p)) ;
                        }
                if (b->cara =='.')
                        {
                        return (evaluer(b->filsg,p)&evaluer(b->filsd,p)) ;
                        }
                if (b->cara =='!')
                        {
                        if (b->filsg !=NULL)
                                {
                                return (!evaluer(b->filsg,p));
                                }
                        }
                }
}
```

```
int main ()
	{
	char chaine[50] ;
	arbre b ;
	list variable ,p ;
	printf ("Entrer l expression a  evaluer : ") ;
	scanf("%s",&chaine) ;
	b =  construction_arbre(&chaine , 0 , strlen (chaine)-1 ) ;
	variable = creationlist(&chaine) ;
	p=variable ;
	while ( p!=NULL )
		{
		printf ("\n entrer la valeur de %c ( 0 ou 1 ):",p->var );
		scanf ("%d",&p->valeur) ;
		p=p->suivant ;
		}
	p= variable ;
	printf ("\nLa valeur de l expression est de %d",evaluer(b,p));
	getchar()  ;
	return 0 ;
}
```

Remarque : ce programme gère une chaîne jusqu'à 50 caractères.