Compte Rendu

TP1-TP2

Expression arithmétique

A l'aide un arbre

Algorithmes de trie

Filière Ingénieur :

Ingénierie Logicielle et Intégration des Systèmes Informatiques

Réalisé par :

OUTGOUGA Jalal eddine

EZADDI Abdelmajid

Encadré par :

Prof. Abdelkrim BEKKHOUCHA

2022/2023

I- L'analyse du problème :

Problème :

Résoudre une expression arithmétique à l'aide d'un arbre binaire. **(Figure 1)**

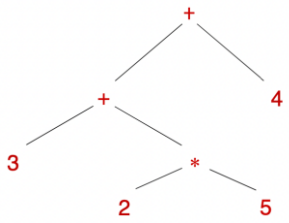

Figure 1: 3+2*5+4=17

Extraction des données à exploiter :

- L'utilisateur va saisie une chaine de caractère.
- La lecture de l'expression se fait caractère par caractère.
- Dans une expression saisie, il faut distingue entre les opérations et les opérandes
- Les opérations possèdent un ordre par rapport à leur priorité :
  - (+, -) ont la même priorité entre eux ;
  - (*,/) ont la même priorité entre eux ;
  - (*, /) sont prioritaires sur (+, -), si ces derniers sont binaires ;
  - (+, -) s'elles sont unaires (début de l'expression), elles sont considérées comme étant un signe ;
  - Pour les opérations (*,/) le calcul doit se faire dans le sens gauche->droite ;
- La validation d'une expression :
  - L'expression n'est pas valide s'il contient des caractères qui ne sont ni opérande ni opération.
  - Si l'expression saisie commence par une opération (*, /), ça sera non valide ;
  - On ne peut pas diviser par 0 ;
  - Un opérande ne peut contenir qu'une seule virgule s'il est décimal ;
  - (+,-) ne peuvent pas être suivies par (* ou /) ;

II- Analyse fonctionnelle :

1- **Fonction principale ''traitement'' :** Elle donne le résultat du calcul de l'expression arithmétique, si la dernière est valide.

- **Spécifications des données et des résultats :**

  Une expression arithmétique sera insérée sous forme une chaine des caractères, dont les opérateurs et opérandes seront extrait.

Le résultat soit du résultat du calcul ou message d'erreur au cas où l'expression est invalide.

2- **Fonction ''est_valide'' :** Elle vérifie la validité d'une expression arithmétique.

- **Spécification des données et des résultats :**

Une chaine des caractères sera traitée caractère par caractère. Le résultat soit valide ou invalide. Si l'un des caractère est diffèrent de l'un des composantes d'un expression arithmétique {+,*,- ,/,0,1,2,3,4,5,6,7,8,9} ou d'une virgule {.} ou du caractère de fin de chaine '\0', l'expression sera jugée invalide, sinon valide.

3- **Fonction ''chaine_vers_tab'' :** Elle traduise l'expression sous forme d'une chaine de caractères en des données manipulables (operateurs et opérandes).

- **Spécification des données et des résultats :**

Une chaine des caractères valide . Le résultat un ensemble des opérandes signés et operateurs {+,*,/}. L'operateur {-} sera inclue dans l'opérande qui lui suie et remplacé par {+} pour faciliter les calcule, ex : {2-3+4}->{2+(-3)+4}. La fonction aussi retourne le nombre des composants de l'expression.

4- **Fonction ''chiffre_extraire'' :** Elle extrait un nombre (décimal ou entier) d'une chaine des caractères.

- **Spécification des données et des résultats :**

Une chaine des caractères. Le résultat est un décimal ou un message d'erreur. Fonction déjà traitée dans le TP1 du module SDD.

5- **Fonction ''inserer_arbre'' :** Elle insère tous les éléments de l'expression dans un arbre après les rendre manipulables

- **Spécification des données et des résultats :**

Une chaine des caractères et un arbre.

Un arbre dont les nœuds sont des opérateurs et les feuilles des opérandes.

- **Spécification fonctionnelle:**

le nombre des éléments d'une expression arithmétique est toujours impaire, chose qui va nous aider à avoir une arbre équilibre selon le principe de l'insertion dont nous expliquons en dessous.

La racine de notre arbre sera un operateur qui se trouve au milieu de l'expression tous les éléments avant seront insérés dans le sous arbre gauche et le reste à droite. Ce principe a des limites, si l'opérateur dont nous parlons est {*,/} nous risquons d'avoir des résultats incorrects, ce problème peut être corrigé par une insertion itérative du 1er élément jusqu'à la fin, mais dans ce cas nous n'allons pas avoir un arbre équilibré, la plus part des éléments sera insérée dans le sous arbre gauche ou droite.

6- **Fonction ''inserer_exp_arbre'' :** Elle insère un élément de l'expression dans un arbre.

- **Spécification des données et des résultats :**

Un operateur ou un opérande et un arbre.

Un arbre dont l'élément est inséré.

- **Spécification fonctionnelle :**

Le premier élément est un opérande sera la racine, puis fils gauche lorsqu'un operateur lui suie. Si un opérande est inséré maintenant dans un arbre contenant des éléments, il sera fils droit si ce dernier n'existe pas, sinon sera le fils le plus droite dans tous les cas il sera fils gauche si un operateur lui suie. Si c'est un operateur est inséré {*,/} est la racine est {+} , l'operateur devient fils droit de l'arbre et l'ancien fils droit devient fils gauche de l'opérateur, tout ça pour respecter l'ordre de priorité des operateur, dans le reste des cas la racine de l'arbre devient fils gauche de l'operateur quelque soit la nature de l'opérateur racine. (Figure 2)

7- **Fonction ''opper_effect'' :** Elle effectue une opération dans un arbre.

- **Spécification des données et des résultats :**

Un operateur et deux opérandes.

Résultat de calcul.

8- **Fonction ''calcule_expres'' :** Elle calcule une expression arithmétique.

- **Spécification des données et des résultats :**

Un arbre.

Résultat de calcul.

- **Spécification fonctionnelle :**

la fonction effectue les calculs récursivement sur le sous arbre gauche et droit et effectue l'opération (fils gauche racine fils droit).

III- Dossier de programmation :

```
typedef union opFl
{
        float valeur; //champ operande
        char oper;//champ operateur
}expre:// nom de l'union
```

```
typedef struct Nd
{
    expre info; //etiquette du noeud
    struct Nd* left;//pointeur sur le s a g
    struct Nd* right;//pointeur sur le s a d
}Noeud; //nom de la structure
```

```
int est_num_Un(expre elem)
{//si elem est un operateur retourner 0
   if((elem.oper=='+')||(elem.oper=='-')||
        (elem.oper=='*')||(elem.oper=='/'))
      return 0;
```

```
int est_num(char car)
{//si car est une valeur entre 0 ou 9
retourner 1
   return((int)((car>='0')&&(car<='9')));
}
```

```
Noeud* root_be_left(Noeud *arb,Noeud *nv)
{
   nv->left=arb;//arb devient fils gauche de nv
   return ((Noeud*)nv);//nv devient racine
```

```
float opper_effect(float val1, char operateur,  float val2)
{
   // selon l'opperation on retourne le resultat
   switch(operateur)
   {
      case '*': return((float)val1*val2);
      case '+': return((float)val1+val2);
      case '/': return((float)val1/val2);
   }
   return 0;
}
```

```
float calcule_expres(Noeud* arb)
{
   // si c'est un opperande
   if(est_num_Un(arb->info)) return arb->info.valeur;
   // sinon on fait le calcule entre gauche et droite
   return opper_effect(calcule_expres(arb->left),
                arb->info.oper,calcule_expres(arb->right));
}
```

```
Noeud* inserer_exp_arbre(Noeud *arb,expre elem)
{
   Noeud *nv=NULL,*tmp=NULL;
   nv=Creer_Noeud(elem);
   if(!arb) return ((Noeud*)nv);//arbre vide
   //nv est un valeur
   if(est_num_Un(elem))
   {// si la racine n'as pas de fils froit
      if(!arb->right) arb->right=nv; // nv devient fils dt
      else //sinon il est inserer etant le fils le plus a droite
      {
         tmp=arb->right;
         tmp->right=nv;
      } //(ceci sera a gauche si un opperateur le suive)
   }
   else// si nv est un opperateur
   {   // si la racine est + et nv * ou /
      if ( arb->info.oper=='+'&&(elem.oper=='/' || elem.oper=='*'))
      {
         tmp=arb->right; // garder le fils droit de la racine
         arb->right=nv; // le fils droit devient  nv
         nv->left=tmp; // le fils gauche de nv recoit tmp
      }
      //sinon il devient racine et l'arbre son fils gauche
      else arb=root_be_left(arb,nv);
   }
   return((Noeud*)arb);
}
```

```
int est_oper(char car)
{
   if(car=='+' || car=='-' || car=='/' || car=='*')
      return 1;
   return 0;
}
```

```
int car_inconu(char*opperation)
{
   int i;
   for(i=0;i<strlen(opperation); i++)
   {   if(opperation[i]=='.') continue;
      if((!est_num(opperation[i])) && (!est_oper(opperation[i])))
         return 1;//caractere inconnu
   }
   return 0;
}
```

```c
int est_valide(char *opperation)
{
   int ind;
   if(car_inconu(opperation)) return 0;
   // une chaine d'opperation n'est pas valide dans les cas suivante:
   // si la chaine comence ou termine par une opperation * ou /
   if ( opperation[0]== '*' || opperation[0]== '/'||
        opperation[strlen(opperation)-1]== '*'
        || opperation[strlen(opperation)-1]== '/'
       ) return((int)0);

   // on parcours la chaine element par element
   for(ind=1;ind<strlen(opperation)-1; ind++)
   {
      // si c'est un opperant
      if (!est_num(opperation[ind]))
         // si il est suivie par * ou / alors non valide
         if(opperation[ind+1]== '*' || opperation[ind+1]== '/' ) return((int)0);
      // si on a une devision sur 0 donc la caine est non valide
      if(opperation[ind]== '/')
         if(opperation[ind+1]== '0') return((int)0);
   }
   // sinon la caine est valide on retourne 1
   return((int)1);
}
```

```c
int entier(double *chifre,char *chain,int deb)
{
   int j=deb;// deb l'indice du debut de la partie entier
   while((chain[j]!='.')&&(chain[j]!='\0'))
   {
      if(est_num(chain[j]))
      {
         (*chifre)=(((*chifre)*10.0)+(double)(chain[j]-'0'));
         j++;
      }
      else return-1;//un car inconnu
   }
   return j;
}
```

```c
int signe_num(double *signe, char *chain)
{
   int i=0;//pour parcourir la chaine
   while((chain[i]=='-')|| (chain[i]=='+'))
   {
      if(chain[i]=='-')
      {   //si c'est negatif on multiplie sign par -1
         (*signe)*=-1.0;
      }
      i++;
   }
   return i;
}//fin
```

```c
int decimale(double *chifre,char *chain,int deb)
{
   int i=0;
   int j=deb+1;// deb l'indice du debut de la partie entier
   // j pour parcourir la chaine (deb+1) pour depasse (.)
   while((chain[j]!='\0'))
   {
      if(est_num(chain[j]))
      { //calcule
         (*chifre)=((*chifre)+(double)((chain[j]-'0')*pow(10,--i)));
         j++;
      }
      else return -1;// un virgule ou un car inconnu
   }
   return j;
```

```c
double chifre_extraire(char *chain)
{

   double chifre=0.0,signe=1.0;
   int ind1=0;
   //determiner le signe
   if((chain[0]=='-')|| (chain[0]=='+'))
     ind1=signe_num(&signe,chain);
   //caractere non numerique
   if(!est_num(chain[ind1]))
   {
      printf("caractere inconnu au debut\n");
      exit(0);
   }
   //calcule de la partie entier
   ind1=entier(&chifre,chain,ind1);
   if(ind1==-1)//caractere non numerique
   {
      printf("un caractere inconu inserer \n");
      exit(0);

   }
   //si c'est un entier simple
   if(chain[ind1]=='\0') return((double)(signe*chifre));//entier simple
   //calcule de la partie decimale
   ind1=decimale(&chifre,chain,ind1);
   if(ind1==-1)//caractere non numerique
   {
      printf("un virgule de plus ou un car inconnu \n");
      exit(0);

   }
   return((double)(signe*chifre));
}//fin
```

```
void insere_tmp(char* opperation, char *tmp,int *ind1,int *indT)
{
   // tantqu'on a pas atteint la fin de la chaine
   while(opperation[*ind1]!='\0')
   {
      /*si le dernier caractere enregistrer dans tmp une opperation
      et que le suivant a stocker n'est pas un . */
      if(est_num(tmp[*indT])|| tmp[*indT]=='.')
      {
         if(est_num(opperation[*ind1]) || opperation[*ind1]=='.')
            tmp[++(*indT)]=opperation[(*ind1)++]; // on le stock
         else break;
      }
      else tmp[++(*indT)]=opperation[(*ind1)++];
      }
}
```

```
int chaine_vers_tab(char* opperation, expre tableau[100])
{
   int ind1=0,ind2=0,indT=0;
   //ind1 pour la chaine, indT chaine tmp et ind2 tableau d'union
   float var=0.0;
   char tmp[10]; // tableau de caractere temporaire
   /** debut de recuperation du premier opperant */
       // on insere dans le tmp le 1er caractere
      tmp[indT]=opperation[ind1++];
      insere_tmp(opperation,tmp,&ind1,&indT);
      tmp[++indT]='\0'; // on ajoute le caractere de fin de chaine
      indT=0; // on remis l'indice du parcous du tempat a 0
      var=chifre_extraire(tmp); // on recupere l'opperant
      // on le stocke dans le tableau d'union
      tableau[ind2++].valeur=var;
   /** fin de recup?ration du premier opperant**/
     while(opperation[ind1]!='\0')
     {
       // l'element a inserer est surrement un opperateur
       if(opperation[ind1]=='-')
       {   //le signe moins toujours inserer avec une valeur
          tmp[indT]=opperation[ind1++];
          tableau[ind2++].oper='+';
       }
       else
       {
          tableau[ind2++].oper=opperation[ind1++];
          // suivant un opperant qui peut etre signe ou nn
           // on insere dans le tmp le 1er caractere
          tmp[indT]=opperation[ind1++];
       }
       insere_tmp(opperation,tmp,&ind1,&indT);//inserer dans tmp
       tmp[++indT]='\0'; // on ajoute le caractere de fin de chaine
       indT=0; // on remis l'indice du parcous du temp a 0
       var=chifre_extraire(tmp); // on recupere la valeur
       // on le stocke dans le tableau d'union
       tableau[ind2++].valeur=var;
    }
   // on retourne la taille du tableau finale
    return ((int)ind2);
}
```

```
Noeud*inserer_smp_arb(Noeud *arb,expre tab[100],int taille)
{
   int ind;
   for(ind=0; ind<taille ; ind ++)
      arb=inserer_exp_arbre(arb,tab[ind]);
   return arb;
}
```

```
Noeud* inserer_arbre(Noeud *arb,char *expression)
{
   expre T[100];
   int ind,mil,taille;
   //extraction des elements
   taille=chaine_vers_tab(expression,T);
   mil= taille/2;
   //milieu une operateur de periorite
   if(T[mil].oper=='*' || T[mil].oper=='/')
         return inserer_smp_arb(arb,T,taille);
    //milieu une operande
   if(est_num_Un(T[mil]))
   {   //l'operateur avant et apres (* ou /)
      if(T[mil+1].oper!='+' || T[mil-1].oper=='+')
         return inserer_smp_arb(arb,T,taille);
      //prendre l'operateur +
      (T[mil+1].oper=='+')?(mil++):(mil--);
   }
   //inserer le milieu comme racine
   arb=inserer_exp_arbre(arb,T[mil]);
   //inserer les element avant mil a gauche
   for(ind=0; ind<mil ; ind ++)
   {
      arb->left=inserer_exp_arbre(arb->left,T[ind]);
   }
   //inserer les element avant mil a droite
   for(ind=mil+1; ind<taille ; ind ++)
   {
      arb->right=inserer_exp_arbre(arb->right,T[ind]);
   }
   return ((Noeud *)arb);
}
```

```
void traitement(char*chaine)
{
   Noeud *monArb=NULL;
   int taille,ind=0;
   expre T[100];
      if(est_valide(chaine))
      {
         monArb=inserer_arbre(monArb,chaine);
         affichage_horiz(monArb,0);
         printf("\n %s = %f\n\n",chaine,calcule_expres(monArb));
      }
      else
         printf("Opperation inacceptable");
}
```