Université du Québec à Chicoutimi **vendredi 20 février 2015**
Dépt. Informatique et Mathématique

Devoir 2
Programmation, Structure de données et algorithmes
(8SIF109)

Ce devoir doit être remis au plus tard
le vendredi 14 mars 2015 avant 16h.

Instructions

* Pour faciliter la correction de vos programmes, il est recommandé de bien les commenter.
* Dans votre compte sur le sunensa, créez un répertoire dont le nom est devoir2.
* Travaillez en équipe au plus de deux étudiant(e)s, remettez une seule copie par équipe. Il est strictement interdit pour une équipe de copier le travail d'une autre équipe.
* Remettre une feuille où les informations suivantes sont indiquées:
  1) Le nom et prénoms des co-équipiers.
  2) Le nom d'usager du compte sur le sunens où se trouve votre code source et exécutable.
  3) Le mot de passe pour accéder à ce compte.
* Les feuilles sont à déposer dans la boite aux lettres du cours (**8SIF109: programmation, structures de données et algorithmes**) située juste à l'extérieur de la direction du DIM au 4ème étage du pavillon principal ou alors l'envoyer à votre assistant : houssem.amamou@gmail.com.

* **Attention:** ne modifiez plus vos programmes après la date limite.

* **Important:** n'oubliez pas d'interdire l'accès à vos fichiers.

Objectifs du travail demandé

La notation préfixée consiste à placer l'opérateur, suivi de ses arguments. La notation postfixée (notation inverse) consiste à placer les opérandes devant l'opérateur. La notation infixée (parenthèsée) consiste à entourer les opérateurs par leurs opérandes. Nous avons étudié en cours la notation postfixée. Par exemple, l'expression infixée (a+b)*c – d/k s'écrit en

notation préfixée : - * + a b c / d k

notation postfixée : a b + c * d k / -

Dans ce TP, nous considérons la notation **préfixée**. Le but est de transformer une expression donnée en infixe en une expression préfixée et ensuite de l'évaluer. L'expression arithmétique de départ est une suite de caractères constituée de :

1. valeurs numériques, et
2. opérateurs **(** , **)** , ***** , /, **%**, **+**, et **-** (pour parenthèse ouvrante et fermante, multiplication, division, modulo, addition et soustraction).

Tout autre caractère est considéré comme invalide pour cette expression. Les priorités des opérateurs, listés ci-dessus, sont dans l'ordre décroissant comme suit:

1. **(** et **)**
2. ***** / et **%**
3. **+** et **-**

Bien entendu, à l'entrée, ces éléments sont lus comme une chaîne de caractères. Il vous revient à vous de les transformer aux types voulus. Pour simplifier les choses, on peut supposer que cette expression est correcte et les valeurs numériques qui la composent sont sur un seul digit.

Écrire un programme C++ implantant les fonctions de passage de l'écriture infixe à préfixe et d'évaluation d'une expression préfixée, en utilisant les **STL Stack** et **Vector** de C++.

Pour ce faire, on utilise une classe **PREFIXE** définie comme suit :

Template <element class>
Class PREFIXE {
Private
Stack <element> Pile;
Vector <element> Tableau;
Public
PREFIXE ( ) : constructeur ;
~PREFIXE : destructeur ;
bool lecture(vector<element> tableau);// lit l'expression à évaluer, à partir du clavier, dans tableau et valide si l'expression ne contient que les caractères ci-dessus, à savoir les nombres entiers composés de caractères nombres, et les opérateurs cités ci-dessus.
bool valider_expression (vector<element> tableau); // teste si l'expression est bien définie (le nombre de parenthèses ouvrantes et fermantes est le même).
void transformerennombres (vector <element> tableau); // transforme les nombres lus en caractères en valeurs numériques
void transformerenpostfixe(stack<element> Pile, vector<element> tableau);// transforme l'expression lue en une expression prefixée et l'afficher.
int evaluer_expression(stack<element> Pile, vector<element> tableau); // évalue l'expression préfixée et affiche sa valeur.
}

Un bonus de 10% est accordé aux étudiants qui :

1. Prennent en compte le fait qu'un nombre peut-être sur plusieurs digits.
2. Utilisent une déclaration dynamique du tableau et de la pile. Dans ce cas, il est nécessaire de déclarer un destructeur. De plus, les itérateurs **begin** et **end** sont utilisés avec **->**. Par exemple, si on utilise **vector<char> *tableau = new vector<char> (n)** pour déclarer d'une manière dynamique le vecteur tableau de n caractères, alors pour accéder au kème élément de tableau, on fait (*tableau)[k], au premier élément, on fait tableau->begin( ), etc.

Exemple de processus d'évaluation :

```
- * / 15 - 7 + 1 1 3 + 2 + 1 1  =
 - * / 15 - 7   2   3 + 2 + 1 1 =
 - * / 15     5     3 + 2 + 1 1 =
 - *        3       3 + 2 + 1 1 =
 -          9         + 2 + 1 1 =
 -          9         + 2   2   =
 -          9         4         =
                5
```

Ci-dessous le squelette d'un algorithme qui convertit une expression infixe en une expression prefixe. Pour ce qui est de l'évaluation proprement dite de l'expression, c'est à vous de concevoir l'algorithme correspondant.

Algorithm ConvertInfixtoPrefix

```
Créer une pile vide operand et une autre pile vide operator.
// Tant que  l'expression n'est terminée, lire le prochain et traiter le prochain token.
while( l'expression n'est pas finie ) {
   lire le prochain token à partir de l'expression
   // Tester si token est un opérande ou un opérateur
   if  ( token est un opérande )
      // empiler l'opérande dans la pile operand.
      empiler_Operand(token)
    else // Token doit être un opérateur.
    if ( token est '(' or OperatorStack.IsEmpty() or OperatorPrecedence(token) >
       OperatorPrecedence(OperatorStack.Top()) )
      // empiler '(' dans la pile operator
      Empiler_Operator( token )
    else
     if( token est ')' )
      // Continuer à dépiler les piles operator et operand, construisant
      // l'expression prefix jusqu'à ce la parenthèse '(' soit trouvée.
      // ensuite cette expression prefix est retournée  dans la pile operand
      // comme soit un operand gauche ou droit pour le prochain opérateur.
      while( OperatorStack.Top() pas égal à  '(' )
         Depiler_Operator(operator)
         Depiler_Operand(RightOperand)
         Depiler_Operand(LeftOperand)
         operand = concatener operator  avec LeftOperand  et RightOperand
         Empiler_Operand(operand)
     endwhile
     // Dépiler la parenthèse gauche de la pile operator.
     Depiler_Operator(operator)
     else
        if( la précédence de l'operateur  token n'est pas plus grande que celle du sommet de la pile
            operator) {
           // Continuer à dépiler les deux piles, construisant l'expression prefix
           // jusqu'à ce que la pile devient vide ou un operateur au sommet de la pile operator
          // a une precedence plus petite que celle de token .
          while( !OperatorStack.IsEmpty() and  OperatorHierarchy(token) lessThen Or Equal to
                OperatorHierarchy(OperatorStack.Top()) ) {
             Depiler_Operator(operator)
             Depiler_Operand(RightOperand)
             Depiler_Operand(LeftOperand)
             operand = concatener operator avec LeftOperand et RightOperand
             Empiler_Operand(operand)
          } // fin du while
         // Empiler l'opérateur de moindre precedence dans la pile operator
            Empiler_Operator(token)
        } // endif
      } // endif
    } // endif
   } // endif
} // fin du while
```

// Si la pile operator n'est pas vide, continuer à dépiler les piles operator et operand, construisant
//l'expression prefix jusqu'à ce que la pile operator devient vide.

```
while ( !OperatorStack.IsEmpty() )  {
  Depiler_Operator(operator)
  Depiler_Operand(RightOperand)
  Depiler_Operand(LeftOperand)
  operand = concatener operator puis LeftOperand puis RightOperand
  Empiler_Operand(operand)
} // fin du while
```

L'expression prefix de l'expression de départ se trouve au sommet de la pile Operand.