TP Semaine 10
Evaluation d'expressions

Dans ce TP, nous montrons comment évaluer des expressions arithmétiques données sous forme postfixée d'abord, puis sous forme infixée.

Préparation

Copiez tous les fichiers du répertoire `/net/Bibliotheque/AP2/TP_par_Semaine/Semaine10` chez vous. Tapez `make` pour construire automatiquement l'exécutable `main`. Tapez `make clean` pour effacer les fichiers objets.

Il y a trois modules : le module `main.cc` que vous allez éditer, le module `Pile.cc` qui vous fournit le type abstrait pile, le module `token.cc` qui permet de décomposer en opérandes et opérateurs les expressions tapées sur la ligne de commandes.

Dans le module `main.cc` sont décrits les différents exercices.

Les piles en C++

On vous fournit une *classe générique* `Pile` vous permettant de manipuler des piles d'objets de type quelconque. Le prototype de la classe `Pile` est dans `Pile.h`. Son implémentation est dans `Pile.cc`.

L'exemple suivant montre comment cloner une pile de flottants en C++ :

```
void clonerPile( Pile<float> entree, Pile<float> & clone )
{
  Pile<float> interm;
  while ( ! entree.pileVide() )
    {
      interm.empiler( entree.valeurSommet() );
      entree.depiler();
    }
  while ( ! interm.pileVide() )
    {
      clone.empiler( interm.valeurSommet() );
      interm.depiler();
    }
}
```

On remarque qu'il n'y a pas besoin d'initialiser la pile (c'est fait automatiquement dans la déclaration de la variable). On remarque aussi qu'on utilise la notation objet au lieu de la notation fonctionnelle (voir schéma ci-dessous).

ASD	C++
var p : pile d'entiers	Pile<int> p;
PileVide(p)	p.pileVide()
Empiler(p, val);	p.empiler(val);
Depiler(p);	p.depiler();
val ← ValeurSommet(p);	val = p.valeurSommet();

Avant de se lancer ...

Regardez la fonction `analyseExpr`. Elle permet de découper une expression arithmétique (passée sous forme de chaîne de caractères) en ses constituants (opérateurs, opérandes, divers). Elle vous inspirera pour les exercices suivants.

Vous pouvez constater aussi que vous disposez d'un ensemble de fonctions simples vous permettant de savoir si une partie d'expression est un nombre, un opérateur, etc.

Et maintenant ...

Exercice 31 : Ecrivez la fonction `evalExprPost` qui permet d'évaluer une expression postfixée donnée en paramètre.

Exercice 32 : Comme précédemment sauf qu'il faut vérifier la validité de l'expression postfixée donnée (fonction `evalSecuriseeExprPost`).

Exercice 33 : Ecrivez la fonction `exprInfVersExprPost` qui convertit une expression infixée vers une expression postfixée. Pour ce faire, il faudra utiliser une pile d'opérateurs (pile de caractères). La fonction `priorite` sera utile.

Exercice 34 : Comme l'exercice précédent mais en ajoutant la possibilité de mettre des parenthèses dans l'expression infixée (fonction `exprInfVersExprPost2`)..