Semaine 8
Piles - Evaluation d'expressions arithmétiques

Note aux enseignants : A partir de cette semaine, nous retrouvons une alternance Algorithmique en TD et C++ en TP.
En TD, en algorithmique, on n'utilise pas la notation "Objet" (pas de notation pointée), on reste le plus général possible pour décrire la notion de Type Abstrait de Données. Se référer absolument au Cours pour comprendre l'esprit.
En TP, bien sûr, on adapte et on utilise la version "Objet" de l'implémentation.
Donc, en gros, la différence entre algo et C++, c'est que les fonctions de manipulation prennent en paramètres le TAD en algo, et s'applique sur l'objet courant avec la notation pointée en C++. Faire remarquer tout ça aux étudiants dès maintenant.

A distribuer aux étudiants lors de ce TD :

- `ficheTAD1.pdf` : fiche récapitulative des primitives des types abstraits (avec des exercices supplémentaires pour les courageux). La consultation de cette fiche sera autorisée lors des évaluations.

- `Algorithme evalECP` : fourni en annexe de ce TD.

1 Evaluation d'une expression complètement parenthésée

Exemple : $((((A+B)*C)/D)-(E-F))$

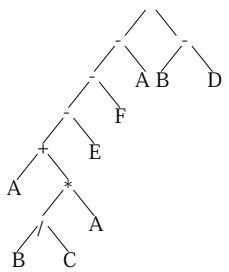
Par souci de simplicité et de clarté, on considère que tous les opérateurs sont binaires (pas d'opérateurs unaires tels que (non A), (-A) ...), que les opérateurs $\in \{+,*,-,/\}$ (résultats numériques). Les variables sont représentées par des lettres $\in \{A,B,...,Z\}$.

Définition d'une expression complètement parenthésée (ECP) :

- Une variable simple est une ECP.
- Si x et y sont des ECP et α un opérateur binaire alors $(x\alpha y)$ est une ECP.

Exercice 19 : Transformer sous forme complètement parenthésée l'expression : $A+B/C*D-E-F-A-(B-D)$

Réponse : $(((((A+((B/C)*D))-E)-F)-A)-(B-D))$



L'expression à évaluer est représentée sous la forme d'un tableau de caractères terminé par le caractère spécial '%' . On suppose l'expression correctement codée dans le tableau `ecp`.

La fonction ci-dessous évalue une expression complètement parenthésée placée dans le tableau `ecp` (distribuer aux étudiants une version papier de l'annexe jointe à ce TD contenant cette fonction) :

```
Fonction évalECP( ecp : tableau[ MAX ] de caractères ) : réel
  var i : entier
      op : caractère
      valdroite, valgauche, résultat : réels
      p : TPile
  Début
    créerPile( p )
    i <- 0
    Tant Que ecp[ i ] <> '%'
    Faire Début
          Si variable( ecp[ i ] )
          Alors empiler( p, valeur( ecp[ i ] ) )
          Sinon Début
                Si opérateur( ecp[ i ] )
                Alors empiler( p, ecp[ i ] )
                Sinon Si ecp[ i ] = ')'
                Alors Début
                      valdroite <- valeurSommet( p )
                      dépiler( p )
                      op <- valeurSommet( p )
                      dépiler( p )
                      valgauche <- valeurSommet( p )
                      dépiler( p )
                      empiler( p, oper2( valgauche, op, valdroite ) )
                      Fin
                Fin
          i <- i+1  // Si on lit '(', on avance simplement
          Fin
    résultat <- valeurSommet( p )
    dépiler( p )
    retourner( résultat )
  Fin
```

Exercice 20 :

Faire tourner l'algorithme sur l'exemple initial avec l'environnement : $A = 5$, $B = 3$, $C = 6$, $D = 2$, $E = 5$, $F = 3$. Dessiner l'évolution de la pile à chaque étape.

Réponse : $((((A + B) * C)/D) - (E - F))$ va donc nous donner :

```
()
(5)             (8)             (48)          (24)              (2,-,24)   (22)
(+,5)           (*,8)           (/,48)        (-,24)                       ()
(3,+,5)         (6,*,8)         (2,/,48)      (5,-,24)
                                              (-,5,-,24)
                                              (3,-,5,-,24)
```

1. Quels sont pour vous, les problèmes posés par cette évaluation ?

   On peut faire remarquer bien évidemment que l'algorithme suppose l'existence des fonctions booléennes `variable(caractère)` et `opérateur(caractère)`, et des fonctions réelles `valeur(caractère)` et `oper2(réel,caractère, réel)`.

   On peut aussi remarquer qu'un tel algorithme est assez abstrait, assez éloigné de toute implémentation, puisqu'on empile dans la même pile des caractères et des réels. Pour les curieux, comment résoudre ce problème ? on pourraît définir des constantes réelles :

   ```
   PLUS = 0.0, MOINS = 0.1, etc...
   ```

   On travaillerait alors avec une pile de réels, et cela ne poserait pas de problème dans la mesure où dans cet algorithme on sait si ce qu'on dépile est un opérateur ou un opérande ; mais ça fait un peu "bricolage".

   Une autre solution serait d'utiliser deux piles : opérateurs et opérandes.

2. Quand vous représentez des expressions mathématiques, sont-elles sous cette forme complètement parenthésée ? Qu'est-ce qui vous permet de l'écrire sous la forme "habituelle" (dite infixée) sans ambiguïté ?

   Réponse : règles de priorité.

L'évaluation directe d'une expression infixée est délicate, c'est pourquoi l'on réalise en général tout d'abord une traduction de l'expression infixée en une expression postfixée puis une évaluation de cette expression postfixée qui est beaucoup plus simple.

2 Evaluation d'une expression postfixée

Définition d'une expression postfixée (uniquement des opérateurs binaires) :

- Une variable simple est une expression postfixée.
- Si x et y sont des expressions postfixées et α un opérateur binaire alors $xy\alpha$ est une expression postfixée.

La représentation d'une expression sous forme postfixée est unique sans ambiguïté et n'utilise aucune parenthèse.

Exercice 21 : Ecrire sous forme postfixée l'expression de l'exemple initial ainsi que celle du second exemple.

Réponse : $((((A+B)*C)/D)-(E-F))$ donne $AB+C*D/EF--$

et $A+B/C*D-E-F-A-(B-D)$

soit l'ECP $(((((A+(((B/C)*D)))-E)-F)-A)-(B-D))$

donnera $ABC/D*+E-F-A-BD--$

L'algorithme n'a cette fois plus besoin d'empiler des opérateurs. Dès qu'il rencontre un opérateur lors de l'évaluation, il peut dépiler les deux opérandes, réaliser l'opération puis replacer le résultat sur le sommet de la pile.

Exercice 22 : Dessiner l'évolution de la pile sur l'exemple initial avec l'environnement déjà présenté et écrire l'algorithme d'évaluation d'une expression postfixée.

Evolution de la pile

```
()
(5)      (8)     (48)     (24)       (2,24)    (22)
(3,5)    (6,8)   (2,48)   (5,24)               ()
                          (3,5,24)
```

Remarque : L'évaluation d'une expression postfixée consomme moins de mémoire qu'une expression infixée. En effet, si on suppose que la pile représente des registres de calcul, l'évaluation de l'expression précédente a besoin de 5 registres pour une expression infixée alors que, avec une expression postfixée, 3 registres seulement sont nécessaires.

Algorithme

```
Fonction évalExpPF( epf : tableau[MAX] de caractères ) : réel
 var i : entier
     valdroite, valgauche, résultat : réels
     p : TPile
 Début
    créerPile( p )
    i <- 0
    Tant Que epf[ i ] <> '%'
    Faire Début
          Si variable( epf[ i ] )
          Alors empiler( p, valeur( epf[ i ] ) )
          Sinon Début // on a trouvé un opérateur
                valdroite <- valeurSommet( p )
                dépiler( p )
                valgauche <- valeurSommet( p )
                dépiler( p )
                empiler( p, oper2( valgauche, epf[i], valdroite ) )
                Fin
          i <- i+1
          Fin
    résultat <- valeurSommet( p )
    dépiler( p )
    retourner résultat
    Fin
```

3 Passage d'une expression infixée à une expression postfixée

Il est possible de passer d'une expression infixée à une expression postfixée en utilisant une pile (encore) et en empilant uniquement des opérateurs comme vous pouvez le voir sur l'exemple qui suit. Nous ne donnons pas ici l'algorithme qui est un peu compliqué car il faut tenir compte des priorités des opérateurs.

Exemple de transformation de l'expression infixée $A * B + C/(D * E)$ en une expression postfixée.

expression infixe (Entrée)	expression postfixée (Sortie)	pile (travail)
A * B + C / (D * E) I	A	{}
A * B + C / (D * E) I	A	{*}
A * B + C / (D * E) I	AB	{*}
A * B + C / (D * E) I	AB*	{} puis {+}
A * B + C / (D * E) I	AB*C	{+}
A * B + C / (D * E) I	AB*C	{/, +}
A * B + C / (D * E) I	AB *C	{(, /, +}
A * B + C / (D * E) I	AB* CD	{(, /, +}
A * B + C / (D * E) I	AB*C D	{*, (, /, +}
A * B + C / (D * E) I	AB*C DE	{*, (, /, +}
A * B + C / (D * E) I	AB*C DE *	{(, /, +}
A * B + C / (D * E) I	AB*C DE *	{/, +}
A * B + C / (D * E) % I	AB*C DE * /	{+}
A * B + C / (D * E) % I	AB*C DE * / +	{}