TD 3 – Pointeurs en C, Piles et applications

Concepts informatiques (CI2)

2011–2012

1 Exercice

Dans un ordinateur classique un programme peut accéder directement à virtuellement n'importe quelle adresse de la mémoire (ou au moins de la partie de la mémoire qui lui est accessible) pour lire son contenu ou y stocker une information, ou encore il peut utiliser de l'espace mémoire pour implémenter des variables contenant des objets de types divers.

Par ailleurs, certaines machines (virtuelles) qui doivent effectuer des tâches limitées et relativement simples ont des capacités de gestion de la mémoire bien inférieures de celles des ordinateurs classiques. Parmi ces machines, on cite les *Automates à Pile* (AAP) dont la tâche est de vérifier qu'un mot donné (sur un alphabet quelconque) possède certaines propriétés.

La seule mémoire auxilliaire d'un AAP est une pile (dans laquelle on peut en principe stocker des objets de n'importe quel type). On n'a donc pas la possibilité de manipuler des variables entières pour implémenter des compteurs, par exemple. Les seules opérations de manipulation de la mémoire que l'on peut faire sont donc les opérations fondamentales de pile : empilement, dépilement, test de pile vide.

1.0 Écrire les différentes fonctions de manipulation d'une pile de caractères (en utilisant par exemple un tableau de caractères et un «indice» de sommet) : `empile(...)`, `depile(...)`, `estUnePileVide(...)`.

1.1 Écrire un programme qui simule le comportement d'un AAP qui vérifie si une chaîne de caractères représente une expression bien parenthésée. Autrement dit, en ignorant tous les caractères qui ne sont pas des "`(`" ou des "`)`" (qui n'ont aucune relevance pour le problème donné) la machine doit :
- reconnaître comme "bons" les mots `()`, `()(())` ou `(((())())())`
- reconnaître comme "mauvais" les mots `)(`, `()(()` ou `(((())))))`.

1.2 Modifier le programme pour qu'il puisse reconnaître les expressions bien parenthésées contenant deux types de parenthèses (par exemple `( )` et `[ ]`).

1.3 Écrire un programme qui simule le comportement d'un AAP qui vérifie si une chaîne de caractères (ne contient que des lettres a et b) possède autant d'occurrences de a que d'occurrences de b. Par exemple $aababbbaab$.

1.4 Écrire un programme qui simule le comportement d'un AAP qui vérifie si une chaîne de caractères (ne contenant que des lettres a et b) possède un nombre d'occurrences de a égal au double du nombre d'occurrences de b. Par exemple $aabaab$ ou $babaaa$ (il y a 4 occurrences de a et 2 de b).

2 Exercice

Il a été montré en cours qu'une expression arithmétique écrite en forme post-fixe peut facilement être évaluée à l'aide d'une pile. Cependant, la représentation "classique" des expressions arithmétiques est en forme infixe (les opérateurs sont plaçés *entre* et non *après* les deux opérandes, ce qui nécessite l'utilisation de parenthèses). On se propose donc de programmer la conversion d'une forme vers l'autre.

On commencera par convertir en forme infixe une chaîne de caractères `Exp` représentant une expression arithmétique écrite en forme post-fixe. Plus précisément, le programme devra lire une seule fois, de la gauche vers la droite, la chaine `Exp` et, en utilisant de manière approprié une pile, affichera l'expression infixe correspondante. Pour simplifier l'analyse de la chaîne `Exp`, on pourra se limiter au cas où les opérandes sont toutes constituées d'une seule lettre et où les seuls opérateurs sont `+` et `*`.

Suggestion : il suffira de modifier le programme présenté en cours pour qu'il transforme l'expression en forme infixe au lieu de l'évaluer.

On va écrire maintenant un programme qui effectue la conversion inverse : en lisant une seule fois, de la gauche vers la droite la chaîne de charactères `Exp` représentant une expression arithmétique écrite en forme infixe, afficher la même expression arithmétique écrite en forme post-fixe.

Voici quelques suggestions :

- Les opérandes sont directement affichées sans passer par la pile.
- La pile ne contient donc que des opérateurs.
- A chaque opérateur, sauf "`)`", est attribuée une priorité comme suit :
  - Opérateur `*` : priorité 2
  - Opérateur `+` : priorité 1
  - Opérateur `(` : priorité 0
- En fonction des priorités de l'opérateur couramment lu et de celui se trouvant au sommet de la pile, on devra effectuer des opérations différentes sur la pile (ne pas oublier la gestion des parenthèses fermantes), c'est à vous de déterminer lesquelles.

Vous pourrez vérifier si l'algorithme que vous envisagez est correct en le simulant sur les exemples suivants :

1. `A + B;`
2. `A + B * C;`
3. `(A + B)*C;`
4. `A + B * C + D * E;`
5. `A * B + ( C + D ) + E;`

Écrire enfin le programme correspondant.