Modules : exercices corrigés

Objectifs

– Comprendre la récursivité ;
– Comprendre la notion de type abstrait de données ;
– Savoir structurer une application en modules.

Exercice 1 : Pile de caractères
Une pile est une structure de données contenant des données de même type. Le principe d'une pile est que le dernier élément ajouté est le premier à en être retiré. Si on considère une pile d'assiettes, on pose une nouvelle assiette au sommet de la pile. Si on prend une assiette, c'est naturellement celle du sommet de la pile. On dit que la pile est une structure de données de type LIFO (Last In, First Out : dernier entré, premier sorti). Les opérations sur une pile sont les suivantes :

– initialiser une pile (une variable de type Pile). Une pile ne peut être utilisée que si elle a été initialisée ;
– empiler un nouvel élément dans une pile ;
– dépiler une pile (supprimer le dernier élément ajouté dans la pile) ;
– accéder à l'élément en sommet de pile ;
– savoir si une pile est vide ou non.

1.1 *Spécification des opérations de la pile.* Définir l'interface des sous-programmes correspondants aux opérations définies ci-dessus.

1.2 *Utilisation de la pile.* La pile étant définie, on peut écrire une application l'utilisant même si l'implantation des opérations de la pile n'est pas encore définie.

Par exemple, on peut utiliser une pile de caractères pour afficher un entier à l'écran en utilisant simplement l'affichage d'un caractère. Écrire un tel sous-programme. On appliquera le schéma de Horner.

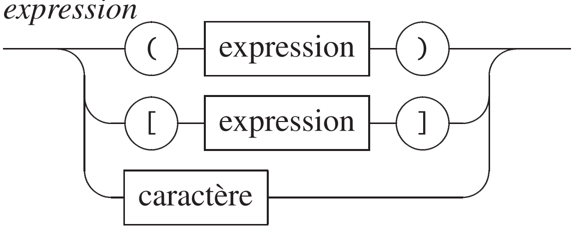
1.3 *Implantation des opérations de la pile.* On choisit de représenter la pile par un enregistrement composé d'un tableau de caractères (les éléments mis dans la pile) et un entier (le nombre d'éléments dans la pile). Le i^e élément ajouté dans la pile est à la i^e position du tableau.

Écrire les implantations des sous-programmes spécifiés à la question 1.1.

Exercice 2 : Expressions bien parenthésées
l'objectif de cet exercice est de vérifier que des expressions sont bien parenthésées par rapport à l'imbrication des parenthèses et des crochets.

Une expression est bien parenthésée si elle respecte le diagramme syntaxique suivant :

expression



Sachant que « caractère » est n'importe quel caractère, à l'exception des parenthèses et des crochets bien sûr.

Voici quelques exemples d'expressions bien parenthésées :

```
2 * (x + y)
occurrences[chr(ord('0') + (entiers[i] mod 4))]
[ ( ( [ ] ) [ ( ) ( ) ] ) ]
```

et d'expressions mal parenthésées :

```
2 * (x + y))
occurrences[chr(ord('0')] + (entiers[i] mod 4))
[[[[
]
( ( ] )
```

Pour vérifier qu'une expression est bien parenthésée, on peut utiliser une pile. Les symboles ouvrant sont empilés et lorsque l'on rencontre un symbole fermant, on vérifie que le sommet de la pile contient bien le symbole ouvrant correspondant. Si ce n'est pas le cas, c'est que l'expression est mal formée. De plus, en fin d'analyse de l'expression, la pile doit être vide ce qui garantit que toutes les parenthèses ont été fermées.

Écrire un programme qui permet de vérifier qu'une expression lue au clavier est bien parenthésée. Si elle n'est pas bien parenthésée, le programme devra signaler la position du caractère sur lequel le mauvais parenthésage a été détecté.