POLYTECH MARSEILLE Aix*Marseille Université **HUGO3**

Algorithmique

TD n°6 : Piles et files

Exercice 1 – Fonctions simples sur les piles

Q 1.1 Implantez une fonction `copy` qui prend en argument une pile et renvoie une nouvelle pile du même type. Vous écrirez une version pour les deux implantations vues en cours (par liste simplement chaînée et par tableau).

Q 1.2 Écrivez une fonction `iter` qui, étant donné une pile et une fonction prenant en argument un élément de la pile et renvoyant un void, applique cette fonction à tous les éléments de la pile. Vous écrirez votre fonction de telle sorte qu'elle soit indépendante de l'implantation des piles (autrement dit, elle utilisera des push, pop, *etc.*).

Exercice 2 – évaluation d'une expression arithmétique

On considère un type `token_t` pour représenter un nombre ou un opérateur arithmétique ($+$, $-$ ou $\times$). Une expression arithmétique sera représentée dans cet exercice par une liste de tokens. Pour cela, on définit un token comme une structure pouvant contenir un entier, un nombre réel ou bien un opérateur arithmétique (en fait, il conviendrait d'utiliser ici une union, mais c'est hors programme). Afin de déterminer quel champ du `token_t` il faut utiliser, on définit une énumération `expr_t`.

```
typedef enum {
  N, // les nombres entiers
  R, // les nombres réels
  Op // les opérateurs
} expr_t;

typedef struct {
  expr_t type; // le type du champ à utiliser
  int    nb_int;
  double nb_float;
  char   op;
} token_t;

typedef struct expr_list {
  token_t           token;
  struct expr_list* next;
} expr_list_t;
```

Q 2.1 Définissez trois fonctions `create_token` prenant en argument, respectivement, un `int`, un

`double` et un `char`, et qui renvoient le token de type `token_t` correspondant.

Pour représenter sous forme **postfixée** l'expression infixe $2 \times ((5.5 + 6.5) + (7 \times 8))$, on peut définir une variable de type `expr_list_t` de la manière suivante :

```
expr_list_t expr = create ();
append (&expr, create_token (2));
append (&expr, create_token (5.5));
append (&expr, create_token (6.5));
append (&expr, create_token ('+'));
append (&expr, create_token (7));
append (&expr, create_token (8));
append (&expr, create_token ('*'));
append (&expr, create_token ('+'));
append (&expr, create_token ('*'));
```

Q 2.2 Écrivez une fonction `eval` qui, étant donné une expression **postfixée** de type `expr_list_t`, renvoie le nombre réel correspondant à la valeur de l'expression. Si celle-ci n'est pas bien formée, votre fonction doit lever une exception. Par exemple, $2 \times 5 +$ n'est pas bien formée car il manque un nombre avant le $\times$ ou le 2.

Idée : utiliser une pile pour les opérandes. On parcourt la liste de tokens représentant l'expression arithmétique. Chaque fois que l'on rencontre un opérande, on l'empile. Chaque fois que l'on rencontre un opérateur, on dépile les deux derniers opérandes empilés, on effectue l'opération arithmétique et on empile le résultat. Une fois l'algorithme terminé, la pile des opérandes contient un seul nombre : la valeur de l'expression.

Q 2.3 Définissez une variable de type `expr_list_t` représentant sous forme **infixe totalement parenthésée** l'expression infixe $2 \times ((5.5 + 6.5) + (7 \times 8))$. Les parenthèses sont considérées comme des opérateurs.

Q 2.4 Écrivez une fonction `eval2` qui, étant donné une expression **infixe totalement parenthésée** de type `expr_list_t`, renvoie le nombre réel correspondant à la valeur de l'expression. Les parenthèses sont considérées comme des opérateurs. Si l'expression n'est pas bien formée, votre fonction doit lever une exception.

Idée : utiliser une pile pour les opérandes et une pile pour les opérateurs. Chaque fois que l'on rencontre un opérande, on l'empile dans la pile des opérandes. Chaque fois que l'on rencontre un opérateur différent de «) », on l'empile sur la pile des opérateurs. Lorsque l'on rencontre une «) », on effectue toutes les opérations empilées sur la pile des opérateurs et ce jusqu'à ce que l'on atteigne une « (».

Exercice 3 – Implantation d'une file

On représente une file par une liste doublement chaînée dont on peut aisément obtenir le premier et le dernier élément (comme on l'a vu en cours).

Q 3.1 Proposez une structure pour représenter de telles files.

Q 3.2 Écrivez les fonctions usuelles sur les files (`create`, `push`, `pop`, `front`, `empty`, `clear`).

Q 3.3 On représente maintenant les files par des tableaux (comme vu en cours). Proposez une structure pour représenter de telles files et réécrivez les fonctions de la question précédente pour cette nouvelle structure.

Exercice 4 – simulation d'un pipe Unix

Dans cet exercice, on veut simuler une commande Unix similaire à `cat td6.cpp | sed 's/^ *//g'`. Sous Unix, la commande `cat` affiche sur la sortie standard (`stdout`) le contenu d'un fichier. Le pipe (`|`) connecte la sortie standard de la commande à gauche du pipe sur l'entrée standard (`stdin`) de la commande à droite du pipe. Ainsi, dans la commande ci-dessus, le contenu du fichier `td6.cpp` est passé en entrée de la commande `sed` qui, grâce au script `'s/^ *//g'`, l'affiche tout en supprimant les espaces en début de ligne. Après le TD, vous pourrez regarder les mans de `cat` et de `sed` pour améliorer votre culture Unixienne. Remarquons qu'ici on aurait pu aussi bien utiliser la commande `sed 's/^ *//g' td6.cpp`.

Q 4.1 Écrivez une fonction `skip_spaces` qui, étant donné une chaîne de caractères, modifie celle-ci de manière à supprimer toutes les espaces en début de ligne.

Q 4.2 Écrivez une fonction `f` qui prend en argument une chaîne de caractères contenant le nom d'un fichier ainsi qu'une file de chaînes de caractères. La fonction lit ce fichier et enfile chaque ligne lue dans la file. Pour cela, vous pourrez vous appuyer sur le code suivant :

```
#include <stdio.h>

// ouverture en lecture d'un fichier ("r" = lecture)
FILE* fichier = fopen("nom_du_fichier", "r");
if (fichier == NULL) {
  printf("impossible d'ouvrir le fichier nom_du_fichier\n");
  throw std::exception ();
}

// lecture de toutes les lignes du fichier, en supposant
// qu'aucune de ces lignes ne dépasse 99 caractères
char chaine[100];
while (fgets (chaine, 100, fichier)) {
  ..........chaîne contient la ligne que l'on vient de lire
}

// on referme le fichier après l'avoir lu
fclose(fichier);
```

Q 4.3 Écrivez une fonction `g` qui prend en argument une file de chaînes de caractères, qui défile l'ensemble des chaînes de cette file et qui, pour chaque chaîne, supprime les espaces en début de ligne et affiche le résultat à l'écran.

L'application de la fonction `f` puis de la fonction `g` sur la même file simule un pipe Unix (à ceci près que la fonction `f` se termine avant que la fonction `g` ne soit lancée alors que, sous Unix, les deux fonctions s'exécutent en concurrence).