L2 Algo – Feuille de TP 3

Equipe pédagogique du L2 Algorithmique

Résumé

L'objectif de ces exercices est :
— de vous familiariser avec la structure de donnée pile
— de construire des programmes multi-ficher en utilisant des `Makefile`
— d'utiliser les chaînes de caractère en C

1 Utilisation de la pile

Avant de démarrer le TP, il faut d'abord télécharger depuis Moodle l'archive tp3.tgz et la décompacter dans votre répertoire de travail :

```
$ tar xvf tp3.tgz
$ cd tp3
```

1.1 Compilation multi-fichier

Pour l'instant, nous ne nous intéressons qu'aux fichiers suivants :

test.c Fichier source C contenant un programme de test pour la structure de pile.

pile.h Fichier en-tête C contenant la définition la structure du type `pile_t` et les fonctions travaillant sur cette structure.

pile.c Fichier source C contenant l'implantation des fonctions travaillant sur la pile.

test.h Fichier en-tête contenant la définition des macros permettant le test.

Makefile Fichier texte contenant le script de construction de l'application [1].

Pour l'instant, on ne s'intéresse pas aux autres fichiers source disponible.
Pour construire l'application, il suffit de taper :

```
$ make
```

`make` est un utilitaire qui permet de construire facilement un programme constitué de plusieurs fichiers sources. La construction est minimale : il ne recompile que les fichiers sources qui ont changé (dont la date de modification est plus récente que celle du programme). Pour ce faire, il faut lui fournir un script dans un fichier nommé `Makefile` où on documente toutes les dépendances entre le programme, les fichiers sources et les fichiers en-tête. Par exemple, ici, `test.c` et `pile.c` incluent l'en-tête `pile.h` : il faut donc les re-compiler chaque fois que `pile.h` est modifié.

Le fichier `pile.h` contient les définitions des types et des signatures de fonctions implantées dans `pile.c` et utilisées dans `test.c`. Il représente l'interface entre le *module* `test` et le module `pile`, assurant la cohérence entre les sources correspondants.

1.2 La pile

La *pile*, aussi appelée LIFO (*Last-In First-Out*), est une structure de donnée contenant un ensemble de données empilées dont seulement celle du sommet est directement accessible. Cependant, il est possible d'empiler de nouveaux éléments ou de dépiler un élément précédemment empilé. Pour plus de détails, vous pouvez vous référer à Wikipedia.

Dans notre cas, on va utiliser un pile d'entiers implantée par la structure suivante :

1. Vous pouvez l'éditer pour comprendre comment il fonctionne.

```
typedef struct pile_t {
    int elts[PILE_TAILLE];
    int som;
} pile_t;
```

Les éléments de la pile seront stockés dans un tableau nommé `elts` (contenant au maximum `PILE_TAILLE` éléments) et l'indice de l'élément du sommet est `som`. `som` est initialisé à -1, est incrémenté à chaque empilement et décrémenté à chaque dépilement.

L'ensemble des fonctions disponibles pour traiter les piles et leur documentation sont fournis dans le fichier `pile.h`.

À faire : l'implantation des fonctions de pile est incomplet. Ouvrez le fichier `pile.c` et complétez le corps des fonctions marquées `/* A faire */`. Le programme principal dans `test.c` contient des séquences de test pour tester les fonctions de pile : utilisez-le pour vérifier l'implantation des fonctions de pile en appelant le programme :

```
$ ./test-pile
```

2 Appariement des parenthèses

Dans cet exercice, nous allons utiliser la structure de pile vu précédemment pour vérifier si un programme est bien parenthésé. L'entrée du programme sera une chaîne de caractère et la sortie sera l'affichage de `oui` si le texte entré est bien parenthésé, sinon l'affichage sera `non`. Le texte sera lu au clavier caractère par caractère et se terminera par le caractère '$'.

À faire

1. En utilisant la pile définie précédemment, complétez le programme `verif-parent.c` pour vérifier que chaque parenthèse '(' ouvrante est bien refermée par une parenthèse ')'.
2. Modifiez le programme précédent pour supporter les autres types de parenthèse : '[' et ']', '{' et '}'.
3. Modifiez le programme précédent pour que le numéro de ligne et le numéro de colonne soit affiché quand une erreur est trouvée.
4. (Optionnelà) Modifiez le programme précédent pour que le numéro de ligne et de colonne de la parenthèse fautive soit affiché quand ce n'est pas le bon de type de parenthèse qui est fermé.

On pourra tester le programme ci-dessous avec les chaînes de caractère :
— "for(int i = 0; i < (10 + 1); i++) f(1, 2, 3);"
— "f(x, &y, 0, (x + 2)"
— "t[i] = t[i + 2] − (f(t[i]) − 3);"
— "{ t[i] = (t[i] + 1);"

3 Dérécursification en utilisant une pile

Le programme `somme.c` propose 2 solutions pour calculer une somme exprimée sous forme de chaîne de caractère. Par exemple, la chaîne de caractère "123+5+10" s'évaluera en 138. La chaîne de caractère n'est composée que de chiffres et de symboles '+' et se termine, comme toute bonne chaîne C, par un caractère nul, '\0'.

Le fichier `somme.c` fournie déjà dans la fonction `somme_rec()` une version récursive de cette évaluation :

```
int somme_rec(char *p, int s) {
        if(*p == '\0')
                return s;
        else if(*p == '+')
                return s + somme_rec(p + 1, 0);
        else
                return somme_rec(p + 1, s * 10 + *p - '0');
}
```

Cette version récursive n'est pas très performante car (1) elle n'est pas terminale et (2) un appel récursif est réalisé pour chaque caractère : on souhaite en faire une version itérative.

À faire : Écrire le corps de la fonction somme_iter() qui réalise le même calcul que somme_rec() mais en utilisant une pile à la place des appels récursifs.

4 Calculatrice en polonaise inversée (optionnel)

L'objectif de cet exercice est de réaliser une calculatrice d'expressions notées en polonaise inversée. La *polonaise inversée* ou *notation postfixe* consiste à d'abord écrire les opérandes d'une opération puis son opérateur. Ainsi, l'addition "$1+2$" s'écrira "1 2 +", l'expression "$3\times(2+5)$" s'écrira "3 2 5 + ×" ou l'expression "$(1+2)\times 3$" deviendra "1 2 + 3 ×".

L'avantage de la notation en *polonaise inversée*, outre qu'elle ne nécessite pas de parenthèse, est qu'elle est très facile à programme si on dispose d'une pile :

— si le terme courant est un nombre, on l'empile ;
— si le terme courant est un opérateur, on dépile les opérandes et on empile le résultat.

Le code à réaliser prendra en entrée une chaîne de caractère représentant l'expression en *polonaise inversé* où chaque caractère aura la signification suivante :

chiffre *c* nombre dont la valeur est c[2].

espace ignoré.

'+' opération d'addition (binaire).

'-' opération de soustraction (binaire).

'*' opération de multiplication (binaire).

'/' opération de division (binaire).

'^' opération de puissance (binaire).

', opération de négation (unaire).

'!' opération de calcul de factorielle (unaire).

À faire (a) : écrire le programme de calcul d'expression en *polonaise inversée* qui saisit une chaîne de caractère au clavier et affiche le résultat s'il n'y a pas d'erreur.

Pour la compilation, il faudra mettre en place les règles dans le `Makefile`.

On pourra utiliser les expression suivantes :

— "1 2 3 + +" = 6
— "6 5 * 3 /" = 10
— "1 1 + 2 2 + *" = 8
— "2 3 8 + 4 /" = 4
— "1 _ 4 ! +" = 23

À faire (b) : On désire maintenant que l'utilisateur puisse saisir une expression sous forme infixe. Pour ce faire, on va lire les caractères tapés au clavier et construire la chaîne de caractère en polonaise inversée. Pour cela, on va utiliser l'algorithme Shunting-yard.

On dispose d'une chaîne résultat résultat, e (initialement vide), et d'un pile des opérateurs, p (initialement vide).

A chaque caractère c lu,

— si c est un chiffre, il est ajouté à e
— si $c='('$, il est ajouté à p,
— si $c=')'$, on dépile les éléments de p et on les ajoute à e jusqu'à trouver '(',
— si c est un opérateur,

1. on dépile les éléments de p qu'on ajoute à e jusqu'à trouver un opérateur moins prioritaire,
2. on ajoute c à p

À la fin, on vide la pile p et on ajoute ses éléments à e.

La priorité des opérateurs est la suivante :

2. On se souviendra qu'en ASCII les chiffres sont codés de manière successive à partir du caractère '0'.

Opérateur	Priorité
’+’, ’-’	0
’*’, ’/’	1
’	2
’	3
’(	4