

Un fichier `fenelon_tp12.zip` est à télécharger sur la page web du cours, et à extraire sur votre ordinateur.

1 Implémentation d'une structure de pile

Le but de cet exercice est d'implémenter une structure de pile en utilisant des tableaux, et dont l'opérateur "empiler" fonctionne comme le `append` des listes Python.

```
typedef int value_t;

typedef struct pile pile;
struct pile {
    value_t* tab;
    int taille;
    int nb;
};
```

Notre structure `pile` aura donc 3 champs :

- `value_t* tab` : un tableau d'éléments de type `value_t`;
- `int taille` : la taille de `tab`;
- `int nb` : le nombre d'éléments dans la pile.

Le principe de cette implémentation est le suivant :

- Les éléments de la pile sont stockés au début du tableau `tab`.
- Pour dépiler, il suffit de décrémenter `nb`.
- Pour empiler :
 - si `tab` n'est pas rempli, on utilise la première case non utilisée;
 - sinon, on crée un nouveau tableau de taille `2*taille+1` dans lequel on recopie les éléments de `tab`, et on continue de travailler avec ce tableau.

1. Ouvrir le fichier `pile.c` et implémenter les opérations sur les piles en utilisant le principe ci-dessus. (pour la fonction `creer_pile()`, on posera initialement `p->taille = 1;`)

Voici les signatures que devront avoir vos fonctions :

```
pile* creer_pile(); // crée une nouvelle pile vide
bool pile_est_vide(pile* p); // teste si p est la pile vide
void empiler(pile* p, value_t val); // empile val au sommet de p
value_t sommet(pile* p); // renvoie le sommet de p (p doit être non vide)
value_t depiler(pile* p); // dépile et renvoie l'élément au sommet de p (non vide)
void detruire_pile(pile* p); // libère p de la mémoire
```

2. Tester vos fonctions.

```
$ gcc pile.c test.c -o test; ./test
Test 1 : OK
Test 2 : OK
Test 3 (doit renvoyer 42 12) : 42 12
Test 4 : OK
Test 5 (doit renvoyer 9 8 7 6 5) : 9 8 7 6 5
Test 7 (doit renvoyer 20 19 18 17 16) : 20 19 18 17 16
Test 8 (doit renvoyer 15 --> 0) : 15 14 13 12 11 10 4 3 2 1 0
Test 9 : OK
```

3. Quelle est la complexité de chacune de ces opérations ?

2 Arbre syntaxique

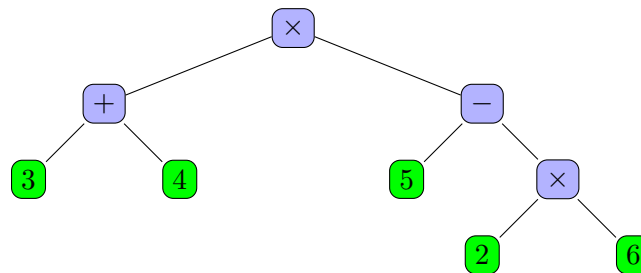
Pour représenter une expression arithmétique, nous avons l'habitude d'utiliser certains symboles :

$$(3 + 4) \times (5 - (2 \times 6))$$

En général, nous utilisons ce qu'on appelle une écriture **infixe**, c'est-à-dire qu'un **opérateur binaire** (opérateur prenant deux arguments) est écrit au milieu de ses deux opérandes (ou arguments).

Pour nos ordinateurs, il n'est pas très pratique d'utiliser la chaîne de caractères " $(3 + 4) \times (5 - (2 \times 6))$ ". Un programme préférera utiliser l'**arbre syntaxique** de cette expression, qu'il va devoir commencer par calculer.

L'arbre syntaxique de l'expression ci-dessus est le suivant :



En informatique, un arbre est une structure constituée de **nœuds** reliés entre eux de manière **hiérarchique** par des **branches**.

Sur l'arbre ci-dessus, le nœud "+" possède deux **enfants** : les nœuds "3" et "4". Les opérateurs possèdent tous deux enfants : leurs deux opérandes. On parle de **nœuds internes**. Les nombres ne possèdent pas d'enfants : on parle de **feuilles** (car ils sont au bout d'une branche). Un seul nœud ne possède pas de **parent** : c'est la **racine** de l'arbre. Tous les autres nœuds possèdent exactement un parent.

4. Donner l'arbre syntaxique de l'expression $(3 + (4 \times 5)) - (2 \times 6)$.

5. Donner l'arbre syntaxique de l'expression $3 + ((4 \times 5) - (2 \times 6))$.

Pour récupérer l'expression infixé correspondant à un arbre syntaxique, on peut procéder de la manière suivante :

- On démarre à la racine de l'arbre ;
 - pour chaque nœud interne :
 - la première fois qu'on passe par ce nœud, on descend dans son enfant de gauche ;
 - la deuxième fois qu'on passe par ce nœud, on affiche l'opérateur ;
 - la troisième fois qu'on passe par ce nœud, on remonte à son parent (si c'est la racine, on s'arrête).
 - pour chaque feuille : on écrit le nombre et on remonte au parent.
6. Effectuer un parcours infixé de l'arbre ci-dessus, ainsi que des deux arbres obtenus lors des questions précédentes. Qu'observez-vous ?

Ce qu'on vient de faire s'appelle un **parcours en profondeur infixé** de l'arbre. Il existe deux autres types de **parcours en profondeur** d'un arbre : les parcours **préfixe** et **postfixe**.

- Parcours préfixe. Pour chaque nœud interne : on traite d'abord le contenu du nœud, puis le sous-arbre gauche, puis le sous-arbre droit.
- Parcours postfixe. Pour chaque nœud interne : on traite d'abord le sous-arbre gauche, puis le sous-arbre droit, puis le contenu du nœud.

3 Écriture polonaise inversée

On peut prouver que, contrairement à l'écriture infixe, nous n'avons jamais besoin de parenthèses pour utiliser l'écriture préfixe ou postfixe. L'écriture postfixe s'appelle aussi l'**écriture polonaise inversée**.

7. Donner l'écriture polonaise inversée des 3 arbres de la partie 2.

L'écriture polonaise inversée possède deux avantages :

- il n'y a jamais besoin de parenthèses ;
- on peut évaluer la valeur d'une expression écrite en polonaise inversée simplement en parcourant cette écriture de gauche à droite et en utilisant une pile.

À cause de cela, certaines vieilles calculatrices fonctionnaient avec l'écriture polonaise inversée, car on pouvait calculer des expressions de manière très efficace (et à l'époque, les calculatrices étaient beaucoup moins puissantes qu'aujourd'hui).

Une calculatrice en écriture polonaise inversée repose sur le principe suivant :

- on démarre le calcul avec une pile vide ;
- on parcourt l'expression de gauche à droite ;
- si on tombe sur un nombre, on l'empile ;
- si on tombe sur un opérateur (**op**) :
 - on dépile les deux derniers éléments de la pile (notés **a** et **b** ; le premier dépilé est **b**) ;
 - on calcule **a op b** ;
 - et on empile le résultat ;
- si lors de ce processus, on essaye de dépiler alors que la pile est vide, c'est qu'il y a une erreur de syntaxe ;
- sinon, si à la fin de cet algorithme, la pile ne contient pas un unique élément, c'est qu'il y a une erreur de syntaxe ;
- sinon, cet unique élément dans la pile est le résultat du calcul.

8. Écrire une fonction `int nb(char n)` ; prenant en argument le caractère '0', '1', ..., '8' ou '9', et renvoyant sa valeur sous la forme d'un entier.

9. Écrire une fonction `int operation(char op, int a, int b)` ; qui prend en arguments un caractère **op** (valant '+', '-', '*', ou '/') et deux entiers *a* et *b*, et renvoie le résultat de **a op b**.

10. Écrire une fonction `int polonaise(char* expr)` ; implémentant la méthode décrite ci-dessus. Pour simplifier, on supposera que l'expression ne contient pas de nombre supérieur ou égal à 10.

11. Tester votre programme sur les expressions de la partie 2.

```
$ gcc pile.c polonaise.c -o test_polonaise; ./test_polonaise
-49
11
11
```

12. Généraliser votre code pour pouvoir gérer des nombres supérieurs ou égaux à 10.

Indication : on pourra utiliser un nouveau symbole (comme '.') pour séparer deux nombres.

Rappel : la fonction `atoi` de `stdlib.h` permet de convertir une chaîne de caractères en entier.

4 Si vous avez fini

Exercice 1 (Structure de file) : Implémenter une structure de file à l'aide de deux piles.

Exercice 2 (Listes chaînées) : Implémenter une structure de pile reposant sur le principe des listes chaînées.

Exercice 3 (Listes doublement chaînées) : Implémenter une structure de file reposant sur le principe des listes doublement chaînées.