

Programmation Impérative – TP 6

Calculatrice

Responsable

Philippe SWARTVAGHER
philippe.swartvagher@enseirb-matmeca.fr

Intervenants

Guillaume MERCIER
mercier@enseirb-matmeca.fr
Augusta MUKAM
augusta.mukam@u-bordeaux.fr

2024 – 2025

On va développer un programme qui permet d'effectuer le calcul d'une expression qui utilise la notation préfixe. Avec cette notation, l'opérateur est placé *avant* ses opérandes. La notation préfixe permet de se passer de parenthèses. Par exemple, avec un tel système, l'opération : $2 + 3$ (notation infixe) sera notée : $+ 2 3$ (notation préfixe). Quand on trouve un opérateur (i.e. $+$, $-$, $*$ et $/$), il faut procéder au calcul de l'opération en regardant la suite des arguments et procéder à leur évaluation. Il y a deux cas :

- soit on trouve un nombre et l'évaluation est directe : c'est la valeur de ce nombre ;
- soit on trouve un nouvel opérateur et on recommence de façon recursive le processus d'évaluation en regardant les arguments suivants.

À la fin de la séance, il faudra déposer sur Thor, dans le projet PG109 > TP6, tous vos fichiers sources (.c) en les mettant dans une archive .tar.gz :

```
tar cvzf archive.tar.gz *.c
```

Lisez bien l'intégralité du sujet avant de commencer à écrire la moindre ligne de code. Le TP peut sembler long, mais ne vous inquiétez pas : je n'attends pas de vous que vous le terminiez.

Puisqu'on est toujours dans le module de PG109, **l'utilisation de LLM est toujours interdite.**

Récupérez le code de base.

Au fur et à mesure du TP, pour tester que votre implémentation est correcte, vous pouvez exécuter **sur une machine de l'école** le script `test.sh` :

```
./test.sh
```

Premières hypothèses

Dans un premier temps, nous allons formuler les hypothèses suivantes :

- les arguments du programme lui seront passés via la ligne de commande. Par exemple :

```
./calc + 2 - 3 1
```

(qui renverra comme résultat 4) ;

- la suite d'arguments passée au programme est correcte, c'est-à-dire qu'il n'est pas nécessaire de gérer le cas où l'utilisateur fait une erreur de saisie quand il rentre une suite d'opérations (pour le moment) ;
- les arguments passés au programme sont de deux types uniquement : des nombres ou bien des opérateurs (*cf* liste ci-dessus).
Cas particulier : l'opérateur de multiplication `*` est interprété d'une façon non convenable par le shell, il faudra donc mettre des doubles guillemets (donc `"*"` à la place de `*`) ;
- les nombres sont des entiers positifs ou nuls (pour le moment) ;
- les opérateurs sont tous binaires (pour le moment).

Implémentation

En ce qui concerne l'implémentation de cette calculatrice, nous allons utiliser une structure d'*arbre binaire*. Une telle structure va utiliser la structure `struct node_s`, déjà définie dans le code de base. Chaque nœud de l'arbre de calcul sera donc représenté par une variable de ce type.

La figure 1 montre comment l'expression préfixe `+ 2 - 3 1` est transformée en arbre binaire, puis comment cet arbre est implémenté à l'aide du type `struct node_s`. Les champs inutilisés de chaque structure ne sont pas représentés sur le schéma.

Exercice 1 – Construction de l'arbre

Complétez les fonctions `build_tree()` et `free_tree()` qui respectivement construisent l'arbre de calcul et libèrent les ressources associées à un arbre de calcul.

La fonction `build_tree()` sera récursive. Sans doute que les fonctions `isdigit()` ou `atoi()` vous seront utiles...

Exercice 2 – Évaluation de l'expression

Complétez la fonction `compute_result()` qui calcule le résultat de l'arbre binaire représentant l'expression passée en paramètre.

Cette fonction sera récursive :

- quand le nœud représente un nombre, c'est la valeur à renvoyer ;

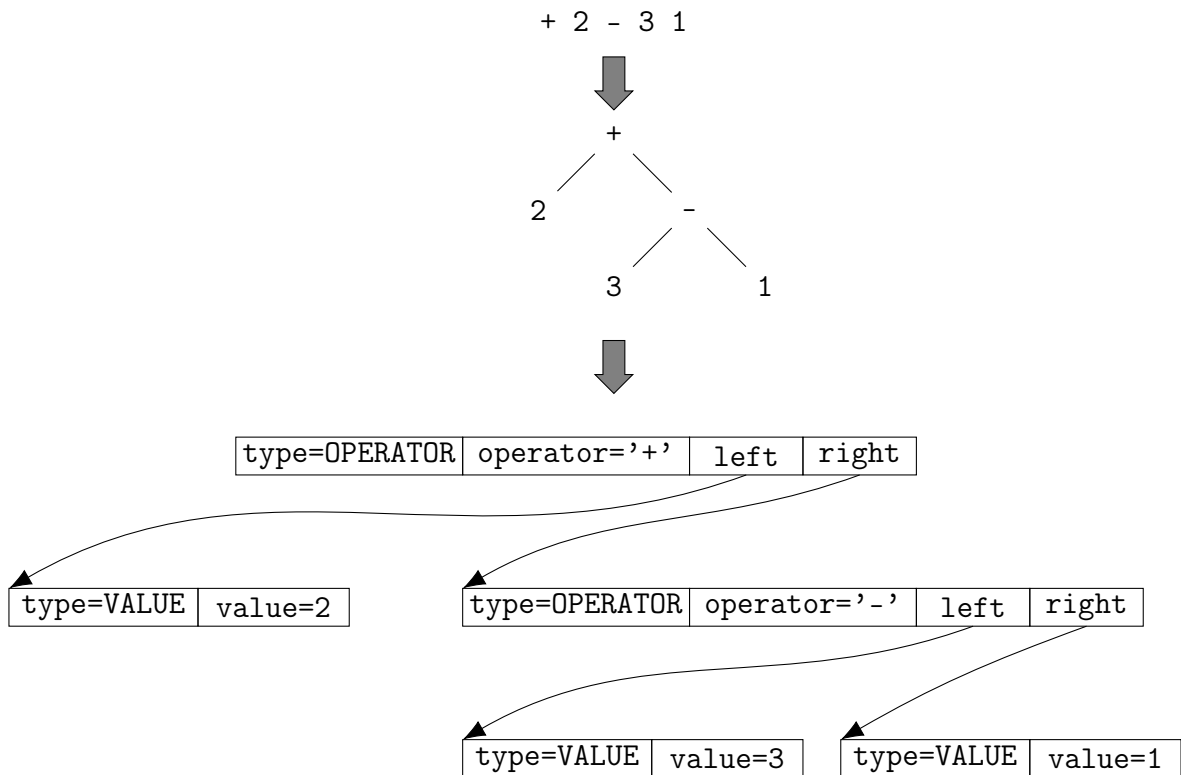


FIGURE 1 – Exemple d'arbre de calcul pour l'expression + 2 - 3 1.

- quand le nœud représente un opérateur, il faut évaluer le sous-arbre de gauche, puis le sous-arbre de droite et enfin effectuer l'opération avec les deux valeurs obtenues.

Exercice 3 – Affichage de l'expression en notation infixe

Complétez la fonction `print_infix()` qui affiche l'expression passée en paramètre en notation préfixe en une notation infixe.

Exemple d'exécution :

```
./calc "*" 2 + 3 4
Result = 14
Infix notation: (2 * (3 + 4))
```

Cette fonction sera récursive :

- quand le nœud représente un nombre, on l'affiche à l'écran ;
- quand le nœud représente un opérateur, on affiche le sous-arbre gauche, puis on affiche l'opérateur et enfin on affiche sous le arbre droit.

On pourra commencer par ne pas afficher les parenthèses.

Exercice 4 – Affichage de l’expression en notation postfixe

Complétez la fonction `print_postfix()` qui affiche l’expression passée en paramètre en notation préfixe en une notation postfixe. La notation postfixe est l’inverse de la notation préfixe : les opérateurs sont affichés *après* les opérandes.

Exemple d’exécution :

```
./calc "*" 2 + 3 4
Result = 14
Infix notation: (2 * (3 + 4))
Postfix notation: 2 3 4 + *
```

Exercice 5 – Support des entiers négatifs

Modifiez votre programme pour qu’il puisse traiter des nombres entiers négatifs.

Exercice 6 – Support des nombres flottants

Modifiez votre programme pour qu’il puisse traiter des nombres flottants.

Si besoin, vous pouvez modifier les prototypes des fonctions et les définitions des types.

Exercice 7 – Suppression du paramètre `position`

Modifiez la fonction `build_tree()` pour qu’elle n’ait plus besoin du paramètre `position`. Le nouveau prototype de la fonction pourra être :

```
struct node_s* build_tree(char*** argv);
```

Exercice 8 – Ajout de fonctions

Modifiez votre programme pour qu’il accepte des fonctions qui nécessitent deux paramètres. On pourra commencer par la fonction `pgcd`, par exemple :

* 3 `pgcd - 16 4 + 5 4`
calculera $3 \times \text{pgcd}(16 - 4, 5 + 4)$.

Cet exercice demande de réorganiser une grande partie du code, copiez votre fichier `calc.c` en un fichier `calc9.c` où vous réaliserez cet exercice.

Pour cet exercice (et le suivant), il sera sans doute judicieux de revoir comment sont stockés les opérateurs : l’utilisation de pointeurs de fonctions s’avérera sans doute utile.

Exercice 9 – Ajout de fonctions avec différents nombres de paramètres

Modifiez votre programme pour qu'il accepte des fonctions qui n'acceptent qu'un seul paramètre, par exemple la racine carrée `sqrt` :

```
* 3 sqrt / 16 4
```

calculera $3 \times \sqrt{\frac{16}{4}}$.

Modifiez ensuite votre programme pour qu'il accepte des fonctions qui peuvent prendre n'importe quel nombre de paramètres. Le nombre de paramètres acceptés sera le suffixe du nom de la fonction à appliquer. Par exemple :

```
max3 * 5 2 - 12 4 + 6 18
```

calculera $\max(5 \times 2, 12 - 4, 6 + 18)$.

Cet exercice demande de réorganiser une grande partie du code, copiez votre fichier `calc9.c` en un fichier `calc10.c` où vous réaliserez cet exercice.

Exercice 10 – Détection des erreurs

Modifiez votre programme pour qu'il s'arrête et prévienne si l'expression à traiter n'est pas valide :

- le nombre d'éléments dans l'expression peut être incorrect, par exemple : `+ 3 * 4 ;`
- les éléments dans l'expression peuvent ne pas être reconnus par la calculatrice, par exemple : `+ 3 z @ ;`
- l'évaluation de l'expression amène à devoir réaliser une division par zéro, par exemple : `/ 27 - 4 * 2 2.`

En cas d'erreur, le programme affichera uniquement `ERROR` :

```
./calc + 3 * 4  
ERROR
```

Pour cela, vous pouvez ajouter un paramètre à la fonction `build_tree()`.