Exercices: Série 11 - Divers - ICC-C 2025-2026

1. Introduction

La notation polonaise inversée est une forme d'écriture des expressions arithmétiques. Au lieu d'écrire les opérateurs *entre* leurs opérandes, on les écrit *après* leurs opérandes. Un avantage de cette notation est que les parenthèses ne sont pas nécessaires.

Par exemple, la notation `5 6 + 3 *` représente l'expression $((5 + 6) \cdot 3)$ (qui évalue donc à 33), alors que `5 6 3 * +` représente $(5 + (6 \cdot 3))$, qui évalue à 23.

Pour « évaluer » une expression en notation polonaise inversée, on maintient une *pile* de valeurs (on supposera des `int` dans cet exercice). On parcourt les différents éléments de l'expression dans l'ordre. Quand on voit un nombre, on l'ajoute à la pile. Quand on voit un opérateur, on *dépile* ses deux opérandes (d'abord le second, puis le premier), on applique l'opération, et on *rempile* le résultat.

Voici par exemple l'évolution de la pile au fur et à mesure de l'évaluation de `5 4 6 + - 3 *` :

Élément appliqué	État de la pile
(départ)	$()$
`5`	(5)
`4`	$(5, 4)$
`6`	$(5, 4, 6)$
`+`	$(5, 10)$
`-`	(-5)
`3`	$(-5, 3)$
`*`	(-15)

Quand on a terminé, il doit rester exactement un élément dans la pile, qui est le résultat.

2. Structure et affichage

Définissez un type de données `operator_t` pour représenter un *opérateur* parmi `+`, `-` et `*`, ou pour indiquer que c'est un nombre (il y a donc 4 possibilités). Utilisez-le dans la structure suivante, qui représente un *élément* d'une expression en notation polonaise inversée :

```c
typedef struct elem {
  operator_t op;
  int value; // utilisé seulement si op dit que c'est un nombre
} elem_t;
```

Écrivez une fonction qui prend un tableau de `elem_t`, et qui les affiche dans l'ordre.

3. Récupérer les arguments

On voudra appeler votre programme en lui donnant l'expression à évaluer en *arguments* :

```
$ ./polish 5 4 6 + - 3 '*'
```

> Le caractère * ayant une signification particulière dans une ligne de commande, on devra l'encadrer avec `'*'` pour l'utiliser tel quel.

Depuis `main` (mais potentiellement *via* une fonction séparée), lisez les arguments du programme, et transformez-les en un tableau de `elem_t`.

Arrangez-vous pour pouvoir écrire des tests unitaires pour cette transformation.

4. Évaluer

Écrivez une fonction qui évalue une expression en notation polonaise inversée. Elle accepte un tableau de `elem_t` et renvoie le résultat.

À ce stade, vous pouvez lancer votre programme pour calculer.

5. Affichage infixe (difficile)

Écrivez une fonction qui accepte une expression, et renvoie une chaîne de caractères en notation *infixe*, c'est-à-dire, la notation habituelle. Mettez des parenthèses autour de chaque opérateur.

Par exemple, pour l'équivalent de `5 4 6 + - 3 *`, renvoyer `"((5 - (4 + 6)) * 3)"`.

Modifiez votre `main` pour afficher la notation infixe avant d'évaluer le résultat.

6. Opérateurs bit-à-bit

Ajoutez le support des opérateurs bit-à-bit (y compris l'opérateur unaire ~) dans votre calculatrice.

7. Simplifications (difficile)

Les compilateurs tels que `gcc` sont très puissants. Il peuvent optimiser les programmes que vous écrivez, notamment en faisant des simplifications algébriques.

Écrivez une fonction `simplify` qui accepte une expression et renvoie une nouvelle expression, qui doit être équivalente mais « simplifiée ». Utilisez-la dans `main` pour simplifier l'expression avant de l'évaluer. Affichez l'expression originale puis l'expression simplifiée.

Procédez de gauche à droite. Quand vous rencontrez un opérateur, tentez de le simplifier avec une des règles ci-dessous. Les règles peuvent être appliquées à n'importe quel *suffixe* de l'expression (autrement dit, il peut y avoir un *préfixe* quelconque, qui sera préservé).

Quelques règles de « réécriture » que vous pouvez essayer, par ordre croissant de difficulté.

Un unique y représente un nombre.

Un $\ldots x$ au milieu représente n'importe quelle sous-expression qui est bien formée, c'est-à-dire que c'est une expression complète en soi. Par exemple, `1 3 +` est bien formée. `3 +` ne l'est pas car il manque un opérande au `+`. `4 1 3 +` ne l'est pas non plus car il y a un `4` de trop. Mais `4 1 3 + -` est bien formée.

Suffixe	Remplacement	Remarques
$0\ +$		on peut retirer `0 +` sans rien rajouter
$-1\ \hat{}$	$\sim$	on peut retirer `-1 ^` si on rajoute `~`
$y\ *$	$z\ \ll$	si y est une puissance de 2, avec $z = \log_2(y)$[1]
$0\ \ldots x\ +$	$\ldots x$	comme la première, sauf qu'on reconnaît quand 0 est le *premier* opérande de `+`, plutôt que le second
$\ldots x\ \sim\ 1\ +$	$0\ \ldots x\ -$	en notation infixe, c'est $(\sim x) + 1 \to 0 - x$

[1] Cette transformation est utile car les processeurs exécutent `<<` plus vite que `*`. Les compilateurs font ça pour vous.