Projet : calculatrice NPI

Guillaume Aubian

Ce projet est à rendre pour le 10 janvier 2025, par mail. Les questions devront soit être traitées en commentaire, soit dans un fichier .txt ou pdf à part. Chaque projet est individuel, et doit être réalisé sans aide extérieure, incluant notamment ChatGPT. Toute triche sera sanctionnée d'un 0. Pour plusieurs questions, la directive de préprocesseur `#include <stdlib.h>;` sera nécessaire.

Usuellement, pour additioner 5 et 7 puis multiplier le résultat par 3, on utilise l'expression `(5 + 7) * 3`. À chaque fois qu'on désire effectuer une opération, on place le symbole correspondant à cette opération entre les deux éléments concernés : on parle donc de notation infixe. Notez que, en notation infixe, les parenthèses sont nécessaires. Ainsi, `(5 + 7) * 3` est différent de `5 + 7 * 3`.

Dans la Notation Polonaise Inversée (ou NPI), ou notation postfixe, on place les opérations après les éléments concernés. Ainsi, on écrira plutôt `5 7 + 3 *` ou `3 7 5 + *`. Dans ce dernier cas, c'est probablement plus clair pour nous, humains, d'écrire plutôt `3 (7 5 +) *`. Néanmoins, en NPI, les parenthèses sont superflues, et donc on les omet toujours.

Quelques exemples en NPI :

- `5 7 + 3 *` vaut 36;
- `1 2 3 4 5 + + + +` vaut 15;
- `1 2 3 4 + * +` vaut 15;
- `12 5 -` vaut 7;
- `5 12 -` vaut -7;
- `20 4 /` vaut 5.

1. Combien vaut `1 3 + 5 *` ? et `1 3 * 5 +` ? On dit que les expressions `1 3 + 5 * +` et `1 1 + 2 3 +` sont non-valides : pourquoi ?

Le but de ce projet sera de coder une calculatrice en NPI. Pour ce faire, nous avons besoin d'implémenter une structure de donnée : la pile. Formellement, une pile p est une structure de donnée qui stocke des éléments et peut être manipulée par les 3 fonctions suivantes :

- `is_empty(p)`, qui renvoie 1 si p est vide, 0 sinon;
- `push(p, x)`, qui ajoute l'élément x au dessus de la pile p;

– `pop(p)`, qui retire de p l'élement au dessus de p et le renvoie.

En pratique, on va implémenter une pile comme un tableau Pascal :

```
struct pile {
    int* tab;
    int taille;
};
```

Ici les éléments d'une pile p sont représentés par un tableau dont les éléments sont stockés dans les $p.taille$ premiers éléments de $p.tab$.

Pour créer une pile, on utilisera donc la fonction suivante :

```
struct pile creer_pile() {
    struct pile p;
    p.tab = NULL;
    p.taille = 0;
    return p;
}
```

2. Implémentez une fonction `void print_pile(struct pile p)` qui affiche, dans l'ordre, tous les éléments d'une pile. Cette fonction sera pratique pour débugger votre code par la suite

3. Implémentez la fonction `int is_empty(struct pile p)`.

4. Implémentez la fonction `void push(struct pile* p, int x)`. Pour ce faire, on utilisera la fonction `realloc`.

5. Implémentez la fonction `int pop(struct pile* p)`. Ne vous embêtez pas à créer un tableau de taille plus petite. De même, supposez sans vérifier que la pile p donnée en entrée n'est pas vide (appliquer la fonction pop sur une pile vide n'est pas défini).

Normalement, si vous ne vous êtes pas trompés, le code suivant :

```
int main() {
    pile p = creer_pile();
    push(&p, 1);
    push(&p, -3);
    pop(&p);
    push(&p, 4);
    print_pile(p);
}
```

devrait afficher 1 puis 4.

Une expression en NPI sera vue comme un tableau de chaînes de caractères. Chaque chaîne de caractère correspond soit directement à un entier, soit à l'une des chaînes de caractères suivante : "+", "-", "*" ou "/". Chacune de ces quatre opérations correspond aux opérations idoines

en C. On dira qu'une expression en NPI est valide si chaque chaîne de caractère est soit un entier, soit égale à "+", "-", "*" ou "/" et si on ne rencontre aucun des problèmes mis en évidence dans la question 1 de ce projet.

6. Utilisez une pile pour coder une fonction `int evaluer(char* expressionNPI[], int N)` qui évalue une expression en NPI expressionNPI de longueur N. On supposera l'expression en NPI valide. On utilisera la fonction `atoi` pour passer d'une chaîne de caractères représentant un entier à un entier.

7. Écrire une fonction `int chaine_valide(char *s)` qui prend en entrée une chaîne de caractères s et renvoie 1 si cette chaîne correspond à un nombre ou à l'une des chaînes suivantes : "+", "-", "*" ou "/".

8. En utilisant la fonction précédente et en vous basant sur la fonction `int evaluer(char* expressionNPI[], int N)`, écrire une fonction `int valide(char* expressionNPI[], int N)` qui vérifie si une expression en NPI est valide.

9. Jusqu'à maintenant on a supposé que la division était la division classique en C (i.e. on arrondit à l'entier le plus proche). Écrire une fonction `int division_exacte(char* expressionNPI[], int N)` qui teste si dans une expression en NPI valide, toutes les divisions se font sans arrondi.

Un des problèmes de votre fonction `evaluer` est sa relative lenteur. Testons-la.

10. Écrire une fonction `char* contre_exemple(int N)` qui renvoie l'expression NPI correspondant à N additions du nombre 1. Appliquez votre fonction `evaluer` à `contre_exemple(100000)`. Que constatez-vous ?

Le problème, c'est que chaque application de push va recopier entièrement la pile. Il va donc y avoir 50000 fois où on recopie un tableau de taille 50000, ce qui fait 2500000000 opérations unitaires : c'est énorme !

Une solution est la suivante : dans notre définition de la pile, on maintient un deuxième entier, qui est la taille réelle du tableau qui stocke les éléments de la pile, et on ne crée un nouveau que quand on dépasse effectivement du tableau actuel. Quand c'est le cas, on ne crée pas juste un tableau avec une case de plus, mais un tableau au moins deux fois plus grand ! Cette opération est un peu plus longue, mais in fine on y gagne tout de même.

11. Implémentez cette solution.

12. Dans les expressions NPI, on cherche à maintenir une opération supplémentaire que l'on note S. Cette opération effectue simplement la somme des entiers en argument. Ainsi `1 2 3 4 5 S` renvoie 15. Rajoutez la gestion de S dans votre fonction `evaluer`.