BABU Bastien
L2 MIDL
22302768

# Compte Rendu TP Algo 3
# Séquence 2 : évaluation d'expressions arithmétiques.

## 1) Description des choix d'implémentation

- **Transformation d'une chaîne de caractère en file de Token :** (1)

L'implémentation de la fonction *stringToTokenQueue* transforme une chaîne de caractères en une file de tokens en veillant à rester performante et robuste. Premièrement, la performance est assurée par l'utilisation d'un curseur *curpos*, qui permet un parcours unique de la chaîne et réduit ainsi les redondances de calcul. Ensuite, la robustesse est renforcée par le fait que les espaces et retours à la ligne sont ignorés dès le début, ce qui rend la fonction résistante aux variations de format de la chaîne lue. Enfin, l'utilisation de la fonction *isSymbol* a permis de factoriser le code afin d'améliorer sa lisibilité.

→ La complexité en temps et en espace de cette fonction est **O(n)**, où n est la longueur de la chaîne de caractères, cela relate d'un algorithme performant.

- **Algorithme de Shunting-Yard :**

L'implémentation de la fonction *shuntingYard* transforme une file de tokens en notation infixe en une file de tokens en notation postfixe tout en veillant à rester performante et robuste. Premièrement, la performance est assurée par l'utilisation de la structure de données *Stack* pour gérer les opérateurs de la chaîne. Ainsi, cela permet un accès plus rapide et une réduction des redondances de calcul. Ensuite, la robustesse est renforcée par une gestion des différents types de tokens. En effet, les nombres/chiffres sont directement ajoutés à la file de sortie, alors que les opérateurs sont manipulés en fonction de leur priorité. Enfin, la gestion dynamique de la mémoire est cruciale et cette fonction s'assure de libérer la mémoire en détruisant la pile d'opérateurs et chaque token qui la constituent renforçant ainsi sa performance et sa robustesse.

→ La complexité en temps et en espace de cette fonction est **O(n)**, ce qui assure une efficacité de la fonction pour les expressions en notation infixe.

- **Évaluation d'expression arithmétique :**

L'implémentation de la fonction *evaluateOperator* permet de gèrer les opérations arithmétiques de base en prenant en compte les opérateurs et en vérifiant les erreurs potentielles, comme la division par zéro. Ce contrôle renforce la robustesse de l'implémentation.

→ La complexité en temps et en espace de cette fonction est **O(1)**, car elle traite qu'un nombre constant de d'opérations et celle-ci n'alloue pas dynamiquement de mémoire ce qui implique une implémentation efficace et rapide.

(1) Fonction issue du première exercice : Analyse lexicale et découpage en Token d'une expression arithmétique

L'implémentation de la fonction *evaluateExpression* permet d'évaluer une expression en notation postfixe en utilisant une structure de données de type Stack/Pile. De cette manière, la fonction assure un accès rapide aux éléments afin de les évaluer par la suite. De plus, chaque token est traité de manière appropriée : les opérateurs sont appliqués aux deux éléments au sommet de la pile, alors que les nombres sont directement ajoutés à la pile. Enfin, la fonction s'assure de libérer la mémoire allouée en supprimant chaque token de la pile ainsi que celle-ci.

→ La complexité en temps et en espace de cette fonction est **O(n)**, où n est le nombre de tokens dans l'expression postfixe.

- **Fonction finale *computeExpression* :**

L'implémentation de la fonction *computeExpressions* permet d'évaluer des expressions arithmétiques lues depuis un flux d'entrée et d'afficher le résultat dans le buffer de sortie *stdout*. Tout d'abord, elle utilise la fonction getline pour lire chaque ligne du flux d'entrée. Cette lecture ligne par ligne permet de convertir chaque expression en file de tokens grâce à *stringToTokenQueue*, ce qui optimise la performance de la fonction en assurant un traitement efficace des données. Ensuite, la conversion de la notation infixe à la notation postfixe est réalisée par la fonction *shuntingYard*, garantissant un accès rapide aux opérateurs en utilisant une pile. En outre, la fonction évalue l'expression postfixe avec *evaluateExpression*, affichant le résultat pour chaque entrée. Pour finir, la mémoire allouée est libérée à la fin de chaque itération (chaque ligne lue), ce qui évite les fuites de mémoires.

→ La complexité en temps de cette fonction est **O(n)**, où n est la longueur de la ligne (nombre de caractères).
→ La complexité en espace de cette fonction est **O(m)**, où m est le nombre de tokens générés par la ligne.

## 2) Analyse personnelle du travail effectué, problèmes rencontrés et solutions mises en place

Dans le cadre de ce TP, j'ai implémenté un algorithme d'évaluation d'expressions arithmétiques, utilisant notamment l'algorithme de Shunting-Yard pour la conversion des notations infixe à postfixe. Tout au long de ce travail, j'ai rencontré plusieurs défis techniques mais aussi futiles qui ont chacun nécessité une réflexion approfondie et des ajustements dans mon code.

Premièrement, l'un des principaux problèmes rencontrés a été une erreur de segmentation (cf. Annexe, page 4, image (1)). Après m'être cassé la tête pendant plusieurs minutes de vérifications, j'ai identifié que la source du problème venait de la syntaxe incorrecte d'une ligne de code dans ma fonction *shuntingYard*. J'ai tout simplement écrit involontairement *stack_pop* au lieu de *stack_top*, ce qui n'a plus aucun sens. Cette erreur d'écriture a notamment été corrigé à l'aide de ChatGPT, qui m'a explicitement expliqué l'origine du problème (cf. Annexe, page 4, image (2)).

De plus, dans la fonction *shuntingYard*, j'ai rencontré des problèmes d'affichage dans le buffer de sortie *stdout*, où les caractères *'('* n'étaient pas supprimés correctement de la pile, ce qui entraînait un affichage erroné (cf.

Annexe, page 4, image (3)). J'ai rapidement réalisé que l'erreur provenait de la condition dans la boucle *while*, qui vérifiait si le token n'était pas une parenthèse ouvrante ou fermante. Cette condition s'est révélée contradictoire, ce qui m'a obligé à la corriger pour garantir un affichage correct par la suite (cf. Annexe, page 4, image (4)).

Enfin, j'ai également dû faire face à des problèmes de fuites de mémoire. Grâce à l'aide d'un de mes camarades, qui m'a expliqué l'idée pour résoudre ce problème, j'ai pu mettre en place des pratiques efficaces pour garantir que toutes les allocations étaient correctement libérées à la fin de chaque itération. Il m'a notamment aidé dans la fonction finale *computeExpressions* our libérer les ressources utilisées que j'avais oubliées (cf. Annexe, page 4, image (5)).

## 3) Références concernant les documentations et aides utilisées

Pour la réalisation de ce projet, j'ai consulté plusieurs sources afin de m'assurer que mon implémentation était correcte et efficace :

- ChatGPT : Pour obtenir des conseils techniques et résoudre des problèmes spécifiques de codage.
- Camarades : Discussions et échanges d'idées pour clarifier certains concepts.
- Wikipedia : Pour une compréhension théorique des algorithmes et des structures de données.
- PDF du cours : Matériel fourni en cours, essentiel pour la compréhension des concepts abordés dans le TP et la réalisation de celui-ci.

# Annexes

```
(base) bastien@MacBookAir Code % make
Generating in release mode
(base) bastien@MacBookAir Code % ./expr_ex1 ../Test/exercice1.txt
Input    : 1 + 2 * 3
Infix    : (5) --  1.000000 + 2.000000 * 3.000000
Postfix  : (5) --  1.000000 2.000000 3.000000 * +
Evaluate : 7.000000

Input    : (1+2) * 3
Infix    : (7) --  ( 1.000000 + 2.000000 ) * 3.000000
Postfix  : (5) --  1.000000 2.000000 + 3.000000 *
Evaluate : 9.000000

Input    : 1+2^3*4
Infix    : (7) --  1.000000 + 2.000000 ^ 3.000000 * 4.000000
Postfix  : (6) --  1.000000 2.000000 3.000000 0.000000 4.000000 *
Evaluate : 12.000000

Input    : (1+2)^3*4
Infix    : (9) --  ( 1.000000 + 2.000000 ) ^ 3.000000 * 4.000000
zsh: segmentation fault  ./expr_ex1 ../Test/exercice1.txt
```

(1)

```
Queue* shuntingYard(Queue* infix){
    Queue* postfix = create_queue();
    Stack* operateurStack = create_stack(0);

    while(!queue_empty(infix)){
        Token* token = (Token*) queue_top(infix);
        queue_pop(infix);

        if(token_is_number(token)){
            queue_push(postfix, token);
        }else if (token_is_operator(token)){
            while(!stack_empty(operateurStack) && token_is_operator((Token*)stack_top(operateurStack))
            && ((token_operator_priority((Token*)stack_top(operateurStack)) > token_operator_priority(token))
            ||(
                token_operator_priority((Token*)stack_top(operateurStack)) == token_operator_priority(token)
                &&
                token_operator_leftAssociative(token)))
                ){

                queue_push(postfix, stack_pop(operateurStack));
                stack_pop(operateurStack);
            }
            stack_push(operateurStack, token);
```

(2)

```
(base) bastien@MacBookAir Code % make
Generating in release mode
(base) bastien@MacBookAir Code % ./expr_ex1 ../Test/exercice1.txt
Input    : 1 + 2 * 3
Infix    : (5) --  1.000000 + 2.000000 * 3.000000
Postfix  : (5) --  1.000000 2.000000 3.000000 * +
Evaluate : 7.000000

Input    : (1+2) * 3
Infix    : (7) --  ( 1.000000 + 2.000000 ) * 3.000000
Postfix  : (5) --  1.000000 2.000000 3.000000 * (
Evaluate : 6.000000
```

(3)

```
} else if (token_is_parenthesis(token) && token_parenthesis(token) == ')'){

    while(!stack_empty(operateurStack) &&
    (!token_is_parenthesis((Token*)stack_top(operateurStack)) && token_parenthesis((Token*)stack_top(operateurStack)) == '(')){
        queue_push(postfix, stack_top(operateurStack));
        stack_pop(operateurStack);
    }
```

(4)

Je me suis rendu compte en fait que ce que j'avais écrit était contradictoire. En effet, on regarde si il n'y a pas une parenthèse au top de la stack **ET** si le token au top de la stack est '(' ce qui implique une contradiction.

```
resultat = evaluateExpression(postfix);
printf("Evaluate : ");
printf("%f\n", resultat);
printf("\n");

while(!queue_empty(postfix)){
    Token* token = (Token*) queue_top(postfix);
    delete_token(&token);
    queue_pop(postfix);
}
delete_queue(&postfix);

while(!queue_empty(tokenQueue)){
    Token* token = (Token*) queue_top(tokenQueue);
    delete_token(&token);
    queue_pop(tokenQueue);
}
delete_queue(&tokenQueue);
```

(5)