Nom: MOSAAD
Prénom: Chehab
Numéro étudiant : 22106126

TP2 : Algorithme de shunting-yard

1) Analyse lexicale et découpage en Token :

1. Lecture des chaines de caractères contenues dans le fichier de données :

Dans la fonction void computeExpressions(FILE *input), j'ai premièrement utiliser la fonction getline pour lire chaque ligne d'entrée. J'ai initiale 3 variables pour pouvoir utiliser la fonction getline ; la variable buffer qui est un pointeur sur char qui pointe vers un emplacement dans la mémoire où getline stocke la ligne lue, la variable size est une variable de type size_t qui contient la capacité initiale allouée pour ligne et la variable longligne est une variable de type ssize_t qui contient le nombre de caractères lus par getline pour la ligne courante.
Puis j'ai créé une boucle while pour parcourir chaque ligne et afficher la ligne d'entrée après l'affichage « Input : » . Ensuite j'ai libéré la mémoire allouée pour cette ligne en utilisant la fonction free et je réinitialise les valeurs des pointeurs buffer à NULL et de size à 0. J'ai utilisé également une condition if, hors de la boucle while, pour vérifier si le pointeur line n'est pas NULL ; s'il n'est pas NULL je libère la mémoire allouée pour ce pointeur pour que la mémoire soit libérée même si la dernière ligne d'entrée n'a pas été lue avec succès.

2. Transformation d'une chaine de caractères en file de Token :

1. Dans la fonction Queue *stringToTokenQueue(const char *expression), j'ai commencé par créer une nouvelle file token_queue vide pour stocker les tokens extraits. J'ai initialisé également le pointeur de caractère courant curpos pour pointer vers le début de la chaîne et la longueur du token courant lg à 1. Puis j'ai fait une boucle principale while qui parcourt tous les caractères de la chaîne ; elle ignore les espaces et les retours à la ligne en augmentant simplement le pointeur de caractère courant et lorsqu'elle trouve un caractère qui n'est pas un espace ni un retour à la ligne, elle vérifie s'il s'agit d'un token. Ensuite, j'ai fait la fonction bool isSymbol(char c) qui renvoie true si le caractère est l'un des symboles d'opérateur ou de parenthèse. Si le caractère courant n'est pas un symbole et que le caractère suivant n'est pas non plus un symbole, cela signifie que nous avons trouvé un nombre. Dans ce cas, la fonction parcourt la chaîne jusqu'à ce qu'elle trouve un symbole ou la fin de la chaîne, en augmentant la longueur du token courant à chaque étape. De même, lorsqu'un token est trouvé, la fonction crée un nouveau token à partir de la sous-chaîne actuelle et l'ajoute à la file de tokens. Elle passe ensuite à la prochaine sous-chaîne en augmentant le pointeur de caractère courant de la longueur du token courant et en réinitialisant la longueur du token courant à 1. Enfin, je retourne la file de tokens token_queue.

2. Dans la fonction void printToken(FILE *f, void *e), j'ai premièrement converti le pointeur générique e en pointeur Token. Ensuite, je ai vérifié si le token est un nombre en appelant la fonction tokenIsNumber(). Si c'est le cas, je fais fprintf() pour afficher la valeur du nombre en appelant la fonction tokenGetValue(). Si le token est un opérateur, j'utilise la fonction tokenGetOperatorSymbol() pour afficher le symbole de l'opérateur. Si le token est une parenthèse, j'utilise la fonction tokenGetParenthesisSymbol() pour afficher le symbole de la parenthèse.

   J'ai aussi modifié la fonction void computeExpressions(FILE *input) pour qu'elle appelle fonction stringToTokenQueue et qu'elle affiche le contenu de la file. J'ai converti chaque chaîne en une file de tokens en utilisant la fonction stringToTokenQueue. Ensuite, dans boucle while, je parcours chaque token dans la file et j'utilise la fonction printToken pour

afficher chaque token. Enfin, j'ai libéré la mémoire allouée pour la file de tokens à la fin de la boucle while.

3. Pour la libération des ressources, il était nécessaire de parcourir la file de tokens et de libérer chaque token individuellement. Ensuite, la file de tokens elle-même doit être libérée, et enfin la chaîne de caractères initiale doit également être libérée. Donc j'ai ajoute 3 boucles une pour libérer chaque token, une autre pour libérer la chaîne de caractères initiale, et une dernière pour libérer la file de tokens. Également j'ai libéré chaque élément de la file avec queuepop, et j'ai utilisé deletequeue pour supprimer la file elle-même à la fin.

2) <u>Algorithme de Shunting-yard :</u>

Pour la fonction Queue *shuntingYard(Queue* infix), j'ai commencé par initialiser la file de sortie et la pile d'opérateurs. Puis j'ai fait une boucle while pour lire chaque Token de la file d'entrée.
Si le Token est un nombre, il est ajouté directement à la file de sortie. Si la pile d'opérateurs est vide, le Token est empilé.
Sinon, si le Token est un opérateur, la fonction compare la priorité de l'opérateur en haut de la pile avec celle de l'opérateur courant, en tenant compte de leur associativité. Si l'opérateur en haut de la pile a une priorité supérieure ou égale, il est retiré de la pile et ajouté à la file de sortie, jusqu'à ce qu'un opérateur de priorité inférieure ou une parenthèse ouvrante soit rencontré. Enfin, l'opérateur courant est empilé.
Si le Token est une parenthèse ouvrante, il est directement empilé.
Si le Token est une parenthèse fermante, les opérateurs sont retirés de la pile et ajoutés à la file de sortie jusqu'à ce qu'on atteigne la parenthèse ouvrante correspondante, qui est retirée de la pile.
Enfin la fonction se termine en retirant les opérateurs restants de la pile et en les ajoutant à la file de sortie. Alors que si la pile contient une parenthèse non fermée, un message d'erreur est généré. Je supprime la pile d'opérateurs car je n'en ai plus besoin et je renvoie la file de sortie qui contient les tokens postfixés.

J'ai aussi modifié la fonction void computeExpressions(FILE *input) tel que à chaque itération de la boucle while, la fonction affiche la ligne lue (buffer) comme expliquer à la première question. Ensuite, la fonction convertit la chaîne de caractères en une queue d'objets Token appelée infix_queue à l'aide de la fonction stringToTokenQueue. Puis afficher la queue infix_queue en notation infixée à l'aide de la fonction queueDump. Ensuite, elle convertit la queue infix_queue en notation postfixée à l'aide de la fonction shuntingYard, stockant le résultat dans une nouvelle queue postfix_queue. Puis afficher la queue postfix_queue en notation postfixée à l'aide de la fonction queueDump. À la fin de chaque itération de la boucle, les deux queues postfix_queue et infix_queue sont supprimées à l'aide de la fonction deleteQueue() pour éviter les fuites de mémoire. Enfin, la fonction libère la mémoire allouée pour buffer à l'aide de la fonction free().

3) <u>Evaluation d'expression arithmétique :</u>

Dans la fonction float evaluateExpression(Queue* postfix), j'ai premierement initialise une pile vide appelée stackEvaluation, qui sera utilisée pour stocker les opérandes et les résultats intermédiaires de l'évaluation. Ensuite, j'ai fait une boucle while pour parcourir tous les éléments de la file postfix. À chaque itération de la boucle, la fonction retire le premier élément de la file avec queueTop et le supprime de la file avec queuePop.
Si l'élément retiré est un opérateur, la fonction récupère les deux opérandes précédemment stockés dans la pile stackEvaluation, effectue l'opération en utilisant la fonction evaluateOperator (décrite en bas ↓) et stocke le résultat dans un nouveau Token appelé res. Les opérandes et l'opérateur sont supprimés et le nouveau Token res est stocké dans la pile stackEvaluation.
Si l'élément retiré est un nombre, la fonction stocke simplement le Token contenant le nombre dans la pile stackEvaluation.
Après avoir parcouru tous les éléments de la file postfix, la fonction récupère le résultat final en utilisant tokenGetValue pour récupérer la valeur numérique du Token en haut de la pile stackEvaluation. Le Token en haut de la pile stackEvaluation est ensuite supprimé et la pile stackEvaluation est aussi supprimée.

Enfin, la fonction retourne le résultat final res_float.

Pour la fonction Token *evaluateOperator(Token *arg1, Token *op, Token *arg2), j'ai commencé par appeler la fonction "tokenGetOperatorSymbol" qui récupère l'opérateur en tant que caractère. Ensuite, je récupère la valeur numérique de chaque opérande à l'aide de la fonction "tokenGetValue". Les valeurs numériques sont stockées dans les variables "op1" et "op2". Ensuite, j'ai implémenté un switch pour déterminer quelle opération mathématique doit être effectuée en fonction de l'opérateur.
Si l'opérateur est une addition (+), la fonction ajoute les deux opérandes.
Si l'opérateur est une soustraction (-), la fonction soustrait le deuxième opérande du premier.
Si l'opérateur est une multiplication (*), la fonction multiplie les deux opérandes.
Si l'opérateur est une division (/), la fonction divise le premier opérande par le deuxième.
Si l'opérateur est une puissance (^), la fonction calcule le premier opérande à la puissance du deuxième opérande.
Finalement, je crée un nouveau Token en utilisant la fonction "createTokenFromValue()" et y stocke le résultat, puis je le retourne. Le nouveau Token contient le résultat de l'opération évaluée en tant que valeur numérique.

J'ai aussi modifié la fonction void computeExpressions(FILE *input) en ajoutant à la boucle while principale, après l'affiche la queue postfixée, j'évalue l'expression postfixée à l'aide de la fonction evaluateExpression, qui retourne un nombre flottant. Puis j'affiche le résultat de l'évaluation à l'aide de printf.