RAPHAËL GOUL - Rapport Algo3 - Séquence 2

---

1. Analyse lexicale et découpage

1.1 computeExpressions

Le but étant d'écrire une fonction d'affichage de flux d'entrée (venant d'un fichier), il fallait donc faire une boucle permettant de lire chaque ligne du fichier stipuler (préalablement ouvert). Il était préciser d'utiliser la fonction `size_t getline`. En m'inspirant d'exemple de mon propre code que j'avais réaliser l'année dernière en **ProgC**, je suis parvenu au code suivant :

```
void computeExpressions(FILE* input) {

    char * line = NULL;
    size_t len = 0; // <- Buffer = 0 car getline va allouer tout seul
    ssize_t read;

    while ((read = getline(&line, &len, input)) != -1) {
        if(read>1) {
            printf("Input : %s", line);
        }
    }
```

> Comme ont peut le remarquer il n'y a pas de libération de mémoire (en effet je m'en suis occupé a la dernière question de la section 1 -> 1.3 Libération de mémoire).

1.2 stringToTokenQueue

Pour cette fonction, au fur et a mesure de mes correction, je me suis un poil écarté de l'algorithme original. Pour commencer, j'ai créer la fonction `bool isSymbol(char c)` nécessaire, l'implémentation est assez simple, c'est juste une fonction qui retourne un bool si `c` correspond a l'un des symbole désigné `+ ; - ; * ; / ; ^ ; ( ; )` :

```
bool isSymbol(char c) {
    return c=='+'||c=='-'||c=='*'||c=='/'||c=='^'||c=='('||c==')';
}
```

> La où j'ai eu un peu plus de problème c'est quand j'ai du vérifier si c'était un chiffre. Disons qu'il est sensé existé une fonction `isdigit(chiffre)` dans la bibliothèque `<ctype.h>`, cependant je ne sais pas du tout pourquoi je n'ai pas réussi a la faire fonctionner. Le compilateur m'afficher des erreur et quand j'ai voulu voir sur stackoverflow si des gens avait la même erreur, je n'ai pas trouver de solution. Je ne me suis donc pas cassez la tête et j'ai recodé rapidement une fonction `bool isDigit(char c)`, elle fonctionne exactement comme `isSymbol`

```
bool isDigit(char c) {
    return
c=='0'||c=='1'||c=='2'||c=='3'||c=='4'||c=='5'||c=='6'||c=='7'||c=='8'||c=='9'
;
}
```

A la toute fin de la séquence, quand j'avais tout fini. Tout marcher bien, et je me suis dit que j'allait tester mon propre fichier test. J'ai donc créer un fichier avec un contenu simple :

```
1+2^(3*4)
1+2
```

Quand j'ai exécuter ça, j'ai eu un énorme problème, je n'avais aucune idée pourquoi mais l'output ressembler à quelque chose comme ca :

```
Input : 1+2^(3*4)
Infix : (9) -- 1.000000 + 2.000000 ^ ( 3.000000 * 4.000000 )
Postfix : (7) -- 1.000000 2.000000 3.000000 4.000000 * ^ +
Evaluate : 4097.000000

Input : 1+2
Infix : (9) -- 1.000000 + 2.000000 + ( 3.000000 * 4.000000 )
Postfix : (6) -- 1.000000 2.000000 3.000000 4.000000 * +
Evaluate : 15
```

En effet, j'avais des résidu dans mon *read*, ce qui est vraiment étrange car ça n'avais poser aucun problème jusque là (pour le fichier `exercice1.txt` aucun résidu n'était présent). Après avoir tenter de débugger tout ça, d'avoir même demander a ChatGPT qui n'arrivait pas a corriger ma fonction sans la recoder entièrement. J'ai finalement réussi a corrigé le bug, en effet en parlant a d'autres camarade de classe j'ai réalisé que ma vérification dans le *while* ne

s'appliquer pas tout le temps J'avais oublier `if (*curpos == '\0') break;`. En effet, lorsque je suis dans le second *while*, j'avance mon curseur. Cependant, si j'avance et que j'arrive a la fin du fichier, le premier *while* ne fait pas de vérification puisque on y retourne pas. Donc, en ajoutant une seconde vérification je permet de régler cet inconvénient.

> J'avoue que même avec ça je ne sais pas pourquoi il y avait un résidu, puisque je donne une ligne a la fonction `stringToTokenQueue`, il est pas sensé pouvoir lire derrière ou après celle-ci. Mais mon problème est régler donc je ne vais pas me poser plus de question.

```
Queue* stringToTokenQueue(const char* expression) {

    Queue* q = create_queue(); // <- Création d'une queue
    const char* curpos = expression; // <- --- et initialisation du curseur
    while (*curpos != '\0') {
        // \r est vérifier au cas où
        while (*curpos == ' ' || *curpos == '\n' || *curpos == '\r' ) {
            curpos++;
        }

        if (*curpos == '\0') break; // <-- Ligne qui ma valu tant de maleur
        if (isSymbol(*curpos)) {
            // CAS SYMBOLE //
            Token* t = create_token_from_string(curpos,1);
            queue_push(q,t);
            curpos++;
        } else if (isDigit(*curpos)) {
            // CAS CHIFFRE //
            int i=1; // <- Décalage a appliqué (pour les nombre > 9)
            while(isDigit(*(curpos+i))) {
                i++;
            }
            Token* t = create_token_from_string(curpos,i);
            queue_push(q,t);
            curpos+=i;
        } else {
            // AUTRE //
            curpos++;
        }
    }
    return q;
}
```

1.3 Libération de mémoire

C'était assez simple pour `computeExpression`, il s'agit d'un *while* ou je supprime les *token* créer dans la queue puis je supprime les *queue* créer.

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

En n'oubliant pas :

```
if (line) free(line);
```

> Petite précision : ici je libère le postfix, en effet c'est parceque j'ai fini le tp et j'ai déjà coder `Queue* shuntingYard(Queue* infix)`. Ce qui ce passe c'est que au moment du codage de `Queue* stringToTokenQueue(const char* expression)` je doit libéré `q` que j'ai créer. Mais a la fin du tp `q` est vide a cause de `shuntingYard`, en effet tout les token dans `q` sont allez dans `postfix`.

---

2. Algorithme de Shunting-yard

C'est probablement la partie qui m'a donner le plus de fil a retordre. En effet, j'ai respecté étape par étape l'algorithme proposer dans le document, malheureusement j'ai eu des complication. Lorsque j'avais fini mon premier jet de code, j'ai lancé le programme et boucle infini. C'était évident que le problème venait de mes condition if, et j'ai eu énormément de mal a régler le problème qui était pourtant très simple. J'avais mal lu la moitié des condition ...

En effet, après relecture et tentative de différentiation entre les `stack` et les `queue` qui était indiqué dans le document (j'avais du mal avec les nom donnée dans le document et ceux les librairie qu'on avait), j'ai réussi à régler le problème et corrigé les erreur que j'avais marqué (c'était principalement des inversion de condition).

> Un autre endroit où j'avais du mal, c'est que j'était partie du principe pour simplifier la lecture de créer des variable temporaire `Token* actual = (Token *)stack_top(s)`.

> Problème : lorsque je suis dans un while, je vais souvent pop le stack, mais la variable que j'ai créer actual, ne va pas être update. C'est une complication qui m'a pris beaucoup de temps a résoudre. Il fallait pourtant juste mettre des `(Token *)stack_top(s)` pour que la valeur ce mettent a jour sans problème. Cela créer des condition énorme et pas très lisible mais au moins elles marchent.

```
while (!stack_empty(s) &&
((token_operator_priority((Token *)stack_top(s)) >=
token_operator_priority(t)) || (token_operator_priority((Token *)stack_top(s))
== token_operator_priority(t) && token_operator_leftAssociative(t))) &&
(token_parenthesis((Token *)stack_top(s)) != ')') ) {
    queue_push(output,stack_top(s));
    stack_pop(s);
}
```

Pour la partie de libération de mémoire j'ai eu une montagne d'erreur valgrind à corrigé. Je n'avais pas compris au départ qu'il fallait que je libère les `token` du `stack`, cependant lorsque je m'en suis rendu compte j'ai mit la condition nécessaire pour libérer la mémoire et j'ai régler la plus part des problème.

```
while (!stack_empty(s)) {
    if (token_is_parenthesis((Token *)stack_top(s))) {
        Token *tk = (Token *) stack_top(s);
        delete_token(&tk);
        stack_pop(s)
    } else {
        queue_push(output,stack_top(s));
        stack_pop(s);
    }
}
delete_stack(&s);
```

Par contre, ce n'était toujours pas parfait, il me manquer quelque byte a libérer en plus. Il me manquer dans l'algorithme une libération de mémoire lorsque `token_parenthesis(t) == ')'`. J'ai du faire énormément de test, jusqu'à que je me rende compte qu'il fallait aussi supprimer `Token* t` a ce moment là.

```
else if (token_parenthesis(t) == ')') {
    while (!stack_empty(s) && (!token_is_parenthesis((Token *)stack_top(s)) &&
token_parenthesis((Token *)stack_top(s)) != '(')) {
        queue_push(output,stack_top(s));
```

```
            stack_pop(s);
        }
        if (stack_empty(s)) {
            break;
        }
        Token *tk = (Token *) stack_top(s);
        stack_pop(s);
        delete_token(&tk);
        delete_token(&t);
    }
```

---

3. Evaluation d’expression arithmétique

3.1 evaluateOperator

Je suis d'abord partie sur l'implémentation de `evaluateOperator`. Comme pour `isSymbol`, il s'agit d'une fonction très simple. J'ai opté pour un switch qui retourne directement un `token` résultat de l'opération des deux valeur récupéré. La gestion d'erreur c'est juste un default qui renvoie un NULL. Cependant je n'ai pas aperçu de cas où c'était intéressant de mettre une condition vérifiant ce genre d'erreur.

```
Token* evaluateOperator(Token* arg1, Token* op, Token* arg2) {
    switch (token_operator(op)) {
        case '+':
            return create_token_from_value(token_value(arg1) +
token_value(arg2));
        case '-':
            return create_token_from_value(token_value(arg1) -
token_value(arg2));
        case '*':
            return create_token_from_value(token_value(arg1) *
token_value(arg2));
        case '/':
            return create_token_from_value(token_value(arg1) /
token_value(arg2));
        case '^':
            return create_token_from_value(pow(token_value(arg1),
token_value(arg2)));
        default:
            return NULL;
```

```
    }
}
```

3.2 evaluateExpression

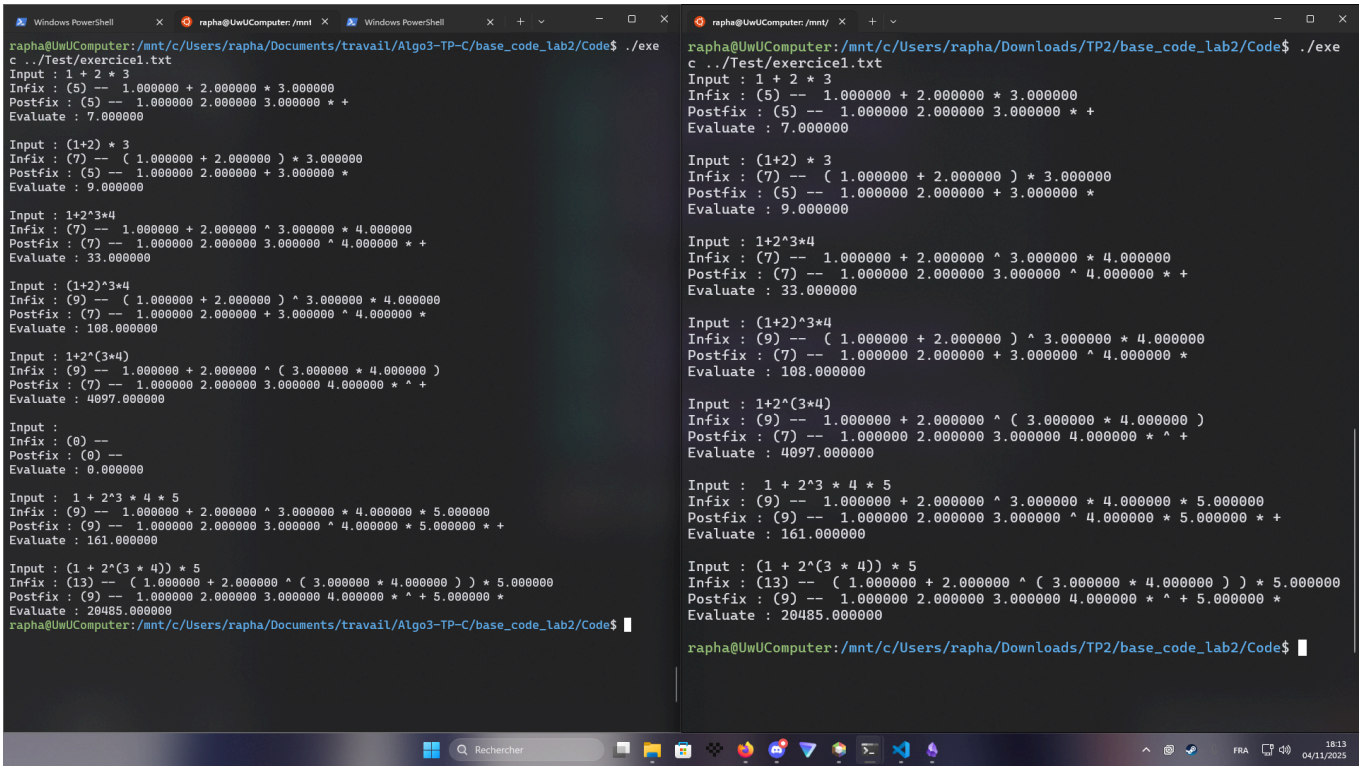
En suivant l'algorithme donner je n'ai pas eu trop de mal a l'implémenter. J'évite les variable temporaire inutile en mettant l'appel a la fonction `evaluateOperator` directement dans le `stack_push`. La stack fait la taille de la queue parce que en théorie, ça ne dépassera pas, et pour libérer la mémoire c'est comme d'habitude, ont prend les `token` de la stack et ont les enlève un par un.

---

4. Conclusion

C'est vraiment que à la fin que je me suis rendu compte qu'on avait créer une calculatrice. Et la ou je suivais bêtement ce que l'algorithme me disait de faire, j'ai compris c'était quoi la logique derrière. J'ai encore beaucoup de chose a revoir sur le maniement de la mémoire et l'utilisation de valgrind, mais je pense m'en être bien sortie, 2. Algorithme de Shunting-yard étant la partie la plus compliqué pour moi. J'ai pas forcément mis de message d'erreur, j'ai juste fait en sorte que sa crash pas (normalement).

> Petite note : Pour rendre le TP j'ai modifier le makefile, pour pouvoir récupéré les lib depuis le dossier Lib. Cependant, je ne sais pas pourquoi, si je prend le nouveau makefile et que je compile il y a un caractère vide qui est lu. Ce qui est très très très bizarre puisque, ce n'était pas le cas avec mon ancien makefile. (Voir page suivante)



> Je n'arrive pas a comprendre comment cela ce fait, j'ai tenter d'ajouter des détection de caractère vide d'espace etc.. C'est toujours lu même si je supprime le caractère. Je laisse l'ancien makefile au cas ou (attention il faut que toute les lib soit dans le dossier et lib2... nommé staticstack)