Compte rendu de la séquence 2 - algo3

Le but de ce travaille était d'implémenter une évaluation d'expressions arithmétiques, en utilisant des structures de données vue en cours d'algorithmique 3, tel que la Pile, ou la File.
Le but de ce document est d'expliquer mes choix d'implémentations, et de données un analyse de mon travail.

Choix d'implémentations

Au cours de ce travil il nous a été demander d'implémenter plusieurs fonction :

- `void computeExpressions(FILE* input)`
- `Queue* stringToTokenQueue(const char* expression)`
- `Queue* shuntingYard(Queue* infix)`
- `float evaluateExpression(Queue* postfix)`
- fonctions auxiliares

ComputeExpression

Le but de cette fonction est de lire les données contenues dans le fichier de test, et d'y appliquer les autres fonctions écrites pour cette exercices.

```
#define BUFFER_SIZE 256
void computeExpressions(FILE* input) {

 char *buffer =unwrapMalloc(sizeof(char) * BUFFER_SIZE);

 size_t buffer_size = BUFFER_SIZE;
 ssize_t readed;
 int i = 0;

 readed = getline(&buffer,&buffer_size,input);
 while(readed != -1){
  if(readed && *buffer != '\n'){
   i++;
   printf("Input : %s",buffer);
   /*Some code*/
  }

  readed = getline(&buffer,&buffer_size,input);
 }
 free(buffer);
}
#undef BUFFER_SIZE
```

Cette fonction lit les lignes une par une, et y applique les fonctions demander (dans la partie commenter `/*Some code*/`)

Gestion de la mémoire dans ComputeExpression J'ai choisi d'y définir une constante `BUFFER_SIZE` afin de pouvoir alouer de façon simple ma mémoire en lisant chacune des ligne. Cela suppose donc que chacune des lignes du fichier fait moins de 256 charactères, sinon cela entrainera un dépassement de mémoire, ce fonctionnement aurait pût être améliorer, en cherchant d'abord la taille de la ligne, et en allouant en conséquence, ce qui aurait réduit la compléxité en mémoire, mais ce n'était pas le but de l'exercice, j'ai donc décider de ne pas me compliquer la tâche et de rester simple. J'ai pris soin de `undef` buffer après la définition de cette fonction, afin de pouvoir définir d'autre constantes au même nom, pour de potentiel future fonction.

Pour mon allocation mémoire j'utilise la fonction auxiliaire `unwrapMaloc`.
Et j'ai aussi pris soin de libérer la mémoire à la fin de chaque éxécution de boucle, afin de garantir qu'il n'y est pas de fuite de mémoire.

stringToTokenQueue

La fonction `stringToTokenQueue` convertie, comme son nom l'indique, une chaîne de charactère en tokens, elle renvoie donc une une File (`Queue *`) remplie de tokens.

```
#define BUFFER_SIZE 256
Queue * stringToToken(const char* expression){
 Queue * queue = create_queue();

 const char* curpos = expression;
 while (*curpos != '\0'){
  Token * p_token = NULL;
  if (*curpos != ' ' && *curpos != '\n'){

    if(isSymbol(*curpos)){
      /*read the token as an operator or parenthesis
    and go to the next element if possible*/
    }
    else if (isFloatingPointNumber(*curpos)){
      /*read the token as a float
    and go to the next element if possible*/
    }
  }
  else {
   /*go to the next element if possible*/
  }
 }
 return queue;
```

```
}
#undef BUFFER_SIZE
```

Cette fonction fonctionne en lisant le charactère à la position `curpos` (initialiser au début de la chaîne), regarde s'il est valide, et gére diférent cas en fonction de s'il s'agit d'une parenthése, d'un nombre, ou d'un opérateur.
J'utilise des fonctions auxiliaires afin de savoir si un charactère correspond a un symbole, un opérateur ou une parenthése, (`isSymbole`) , ou s'il s'agit d'un nombre (`isFloatingPointNumber`).

Gestion de la mémoire dans stringToTokenQueue La constante `BUFFER_SIZE` fonctionne comme pour la fonction précédente.

Cette fonction alloue la mémoire pour des tokens en utilisant les fonctions de type `Token * create_token_from_`, il faut donc penser à libérer la mémoire après coup. Pour cela j'utilise une fonction auxiliaire `freeTokenQueueMap` avec la fonction `queue_map`.

ShuntingYard

Cette fonction implémente l'algorithme de *Shunting Yard*, qui a pour but de faire passer une notation *infixe* (standard : `a+b`) a *postfixe* (polonaise inverse `a b +`). Il s'agit de la fonction que j'ai eu le plus de mal à implémenter, j'ai du m'y reprendre à plusieur fois, et la gestion de mémoire n'était pas facile non plus.

```
Queue * shuntingYard(Queue * infix){

 struct shuntingYardParams params;
 params.output = create_queue();
 params.operators = create_stack(queue_size(infix));

 queue_map(infix,shuntingYardMap,&params); //Do the first part of the algorithm

 //and the second one
 while (!stack_empty(params.operators)){
  /*push operators into infix*/
 }

 free(params.operators);
 return params.output;
}
```

On peut voir que `params.operators` est définie comme étant une Pile de même taille que la File `infix` (`queue_size(infix)`), cela permet d'être sur que la Pile d'opérateurs sera assez grande, car même dans le pire cas, `infix` sontient strictement moins d'opérateurs que son nombre total d'éléments.

La partie dure de cette implémentation était de parcourir la File `infix` sans rajouter de compléxiter en mémoire, j'ai donc opter pour la solution de rajouter une fonction à utiliser avec la fonction `queue_map` :

```
void shuntingYardMap (const void *v_token, void *params){
 struct shuntingYardParams * p_params = params;
 Queue * output  = p_params->output;
 Stack * operator= p_params->operators;
 Token * token = (Token *) v_token;

 if(token_is_number(token)){
  /*handle the case when the token is a number*/
 }
 else if(token_is_operator(token)){
  /*handle the case when the token is an operator*/
 }
 else if(token_is_parenthesis(token)){
  if(token_parenthesis(token) == '('){
   /*handle the case when the token is a left bracket (i.e. "(" )*/
  }
  else /*if equal to ')'*/{
    /*handle the case when the token is a right bracket (i.e. ")" )*/
  }
 }
}
```

Cela nous permet de faire la premier partie de l'algorithme (où il faut parcourir toute File). On peut voir que le paramétre `void *params` corespond à un pointeur vers une structure, cette structure nous sert en effet de moyen de faire passer des paramètres dans la fonction :

```
struct shuntingYardParams{
 Queue * output;
 Stack * operators;
};
```

Elle ne contients que deux champs, un champs `output` qui nous serviras de sorties pour continuer l'éxécution de l'algorithme. et `operators` qui est une autre sortie, une Pile, contenant tout les opérateurs à traiter.

Gestion de la mémoire dans ShuntingYard Au cours de l'éxécution de cette fonction des donneés sont copier au de la File `infix` vers la file retourner (que l'on va nommer `postfix`). Dans les faits, il ne s'agit pas d'une copie profonde ("deep copy"), c'est à dire que les données des éléments ne sont pas copier, mais plutôt les pointeur ves ces éléments, ce qui fait que `postfix` ne fait que contenir des éléments de `infix` juste ordonnées à sa manière, cela est le cas aussi pour les autres Piles et Files utiliser dans cette fonction.

Cela permet de n'avoir à liberer la mémoire de ces éléments qu'une fois, avec la File `infix`.

evaluteExpression

Le but de cette fonction est de calculer le résultat d'une expression arithmétique en nottation *postfix* et de retourner sa valeur.

```
float evaluateExpression(Queue* postfix){
 const Token * token = queue_empty(postfix) ? NULL : queue_top(postfix);
 Stack *stack = create_stack(queue_size(postfix));
 Queue *to_free_queue = create_queue();
 while (!queue_empty(postfix)){

  if(token_is_operator(token)){
   /*handle the case when token is an operator*/
  }
  else if(token_is_number(token)){
   /*handle the case when token is an operator*/
  }

  token = queue_empty(queue_pop(postfix)) ? NULL : queue_top(postfix);
 }
 float result = token_value(stack_top(stack));

 /*free memory*/
 free(stack);
 bool param = true;
 queue_map(to_free_queue,(QueueMapOperator)freeTokenQueueMap,&param);
 delete_queue(&to_free_queue);

 return result;
}
```

Si `postfix` n'est pas une File représentant la notation *postfix* valide, le comportement de cette fonction est indéfinie. En effet, cela a de fortes chances de produire un accés à de la mémoire non aloué, ou aloué pour autre chose. Ceci pourrait être réglé en vérifiant que l'on ne dépasse jamais la taille de la File, et en retournan une erreur dans ce cas, cela aurait demander une utilisation de mémoire suplémentaire, et une vérification redondante, n'étant pas le but de cette exercice, j'ai fais le choix de ne pas en accomoder cette fonction.

Gestion de la mémoire dans evaluteExpression De nouveaux Token son créer dans cette fonction, ils faut donc libérer la mémoire qui leur a était aloué, pour cela j'ai décider de créer une File `to_free_queue`, dans laquelle je pousse chaque token que je créer, à la fin de ma fonction, je libére cette File avec la

fonction `freeTokenQueueMap` dont le comportement a était expliqué un peu plus haut.

Fonctions auxiliaires

Au cours de l'écriture de ces fonctions j'ai de écrire plusieur fonction auxiliare, demander ou non :

-

freeTokenQueueMap Lors de l'éxécution de mon programme, j'ai plusieurs `Token` qui sont aloué et placer dans des File, il faut pouvoir libérer la mémoire occupé par ces `Token`. Pour cela j'ai définie une fonction `freeTokenQueueMap` :

```
void freeTokenQueueMap(void * token_ptr, void * set_null_usr_param/* *bool */)
```

Cette fonction à pour but de libérer tous les token contenue dans une File, en utilisan la fonction `queue_map`, elle peut être utiliser de la manière suivante :

```
bool usr_prm =true;
bool * p_usr_prm = &usr_prm;

queue_map(queue,(QueueMapOperator)freeTokenQueueMap,(void *) p_usr_prm);
```

Le cast en `QueueMapOperator` est ici obligatoire, car cela permet d'éviter les Warning, et donc de compiler quand `-Werror` est ajouter lors de la compilation. En effet `freeTokenQueueMap` n'est pas du bon type, car `token_ptr` n'est pas constant, afin de pouvoir libérer sa mémoire.

unwrapMaloc Pour éviter les comportements non définie qui peuvent subvenire quand une allocation mémoire échoue avec la fonction `malloc` J'ai définie une fonction `unwrapMalloc`

```
void * unwrapMalloc(size_t size){
 void * result = malloc(size);
 if (!result){
  fprintf(stderr,"could not initialize %ld bytes into the heap\n",size);
  perror("stopping program ");
  exit(1);
 }
 return result;
}
```

Cette fonction me permet de renvoyer une erreur quand un `malloc` échoue, ce qui me permet de ne pas avoir à faire la vérification manuelle à chaque allocation de mémoire.

isSymbole Le but de la fonction `isSymbole` est de savoir si un charactére correspond a un opérateur ou a une paranthése.

```
bool isSymbol(char c){
 switch (c){
 case '+': case '-': case '*': case '/': case '^': case '(': case ')':
  return true;

 default:
  return false;
 }
}
```

J'ai choisi d'utilser un "switch case" car cela permet un fonctionnement plus rapide du programme (sans optimisation du compilateur) et augmente la lisibilité (faire une condition avec `(c == '+' || c == '-' ||...)` aurait été moins lisible).

isFloatingPointNumber Retourne si un charactère fais partie d'une chaîne de charactère correspondant a un flottant.

```
bool isFloatingPointNumber(char c){
 return (c >= '0' && c <= '9') || c == '.';
}
```

Il n'y a pas grand chose a dire sur cette fonction.