Un drôle de calcul druide (mini projet) Rendre un travail (PDF) en binôme – programmation libre

Calcul n°1 : 3 5 + donne 8
Calcul n 2 : 4 7 + 3 * donne 33
Calcul n 3 : 3 4 7 + * donne 33
Calcul n 4 : 10 4 + 2 - donne 12
Calcul n 5 : 2 10 4 + - donne -12

Question 1- Analyse de la stratégie (AB)

Calcul n 5 : 2 10 4 + - donne -12 = 2 - (10 + 4)

1. Lire 2 → c'est un nombre → on l'empile : `[2]`
2. Lire 10 → c'est un nombre → on l'empile : `[2, 10]`
3. Lire 14 → c'est un nombre → on l'empile : `[2, 10, 4]`
4. Lire + → on dépile les deux derniers nombres : 10 et 4
   → on calcule `10 + 4 = 14`
   → on empile le résultat : `[2, 14]`
5. Lire - → on dépile les deux derniers nombres : 2 et 14
   → on calcule `2 - 14 = -12`
   → on empile le résultat : `[-12]`

Prenons l'exemple :
`4 7 + 3 *`

Étapes :

1. Lire 4 → c'est un nombre → on l'empile : `[4]`

2. Lire 7 → c'est un nombre → on l'empile : `[4, 7]`

3. Lire + → on dépile les deux derniers nombres : 7 et 4
   → on calcule `4 + 7 = 11`

→ on empile le résultat : [11]

4. Lire 3 → on empile : [11 , 3]

5. Lire → on dépile : 3 et 11
   → on calcule 11 * 3 = 33
   → on empile le résultat : [33]

le calcul empile les chiffres et quand il rencontre un opérateur fait le calcul avec les chiffres précédents, puis il prend le résultat et continue de la même manière.

La technique informatique utilisée par Zarhbic est basée sur une structure de données “pile”soit autrement dit “LIFO”.

Question 2: Les contraintes sont: s' il y a trop de nombre par rapport aux opérateurs ou trop d’opérateurs par rapport aux nombres. Car la méthode est dite “Binaire”.
Exemple:
3 + → Impossible
4 5 + * → 4 +5 =9 mais 9 reste inutilisé.

Question 3:

services	Description	Valeurs ajoutés
lecture_expression()	Lit la chaîne de calcul	Permet de séparer proprement les entrées/sorties du traitement.
tokenizer()	Découpe la ligne de texte brute en un tableau de mots (tokens) séparés par des espaces.	Transforme une phrase complexe en éléments unitaires manipulables, simplifiant les étapes suivantes.
verifier_syntaxe()	Valide chaque token (nombre ou opérateur), vérifie que l’expression respecte les règles RPN.	Détecte tôt les erreurs et évite les crashs durant l’évaluation.
evaluer_rpn()	Exécute le calcul en notation polonaise inversée à l’aide d’une pile.	Cœur de l’algorithme, structure claire et robuste, compatible avec d’autres opérateurs.
afficher_resultat()	Affiche le résultat final ou un message d’erreur lisible.	Donne un retour clair à l’utilisateur, facilite le débogage et l’usage du programme.

Stratégie de gestion des erreurs

Pour éviter que le programme plante ou produise un résultat incorrect, plusieurs vérifications sont mises en place tout au long du traitement. L'idée est de détecter les problèmes le plus tôt possible et d'afficher un message clair à l'utilisateur.

- **Manque d'opérandes (underflow)**
  Lorsqu'un opérateur apparaît, le programme vérifie qu'il y a bien deux valeurs dans la pile.
  Si ce n'est pas le cas, il signale l'erreur avec un message du type :
  « L'opérateur + nécessite deux opérandes. »

- **Division par zéro**
  Avant d'effectuer une division, le programme vérifie que le diviseur n'est pas égal à zéro.
  En cas de problème, il affiche simplement :
  « Division par zéro. »

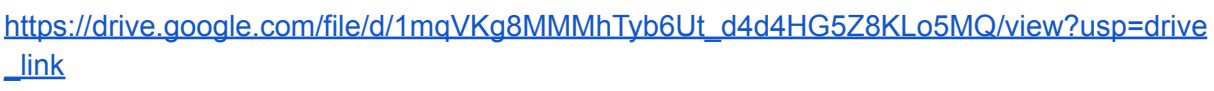
- **Opérandes en trop à la fin du calcul**
  Une fois l'expression entièrement évaluée, la pile doit contenir exactement un seul résultat.
  Si plusieurs valeurs sont encore présentes, cela signifie que l'expression est incorrecte.
  Message renvoyé :
  « Expression RPN invalide. »

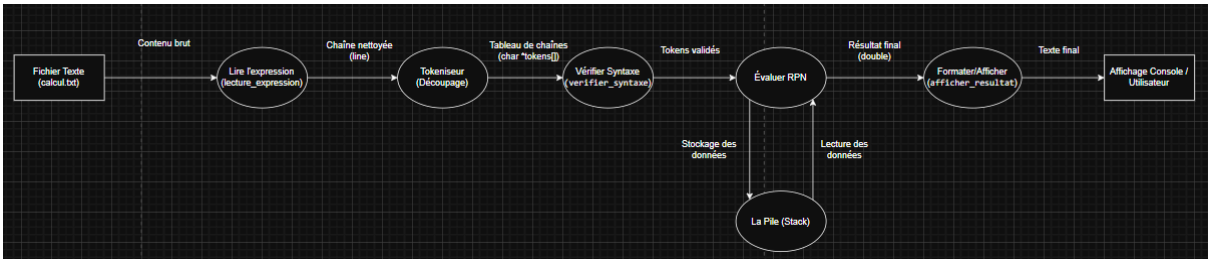
- **Erreur lors de la lecture du fichier**
  Si le fichier n'existe pas, est vide ou impossible à lire, le programme avertit l'utilisateur :
  « Impossible de lire le fichier. »

Principe général

Chaque erreur est signalée proprement avec un message explicite.
Le programme ne s'arrête jamais brutalement : il remonte l'erreur et continue à fonctionner correctement.

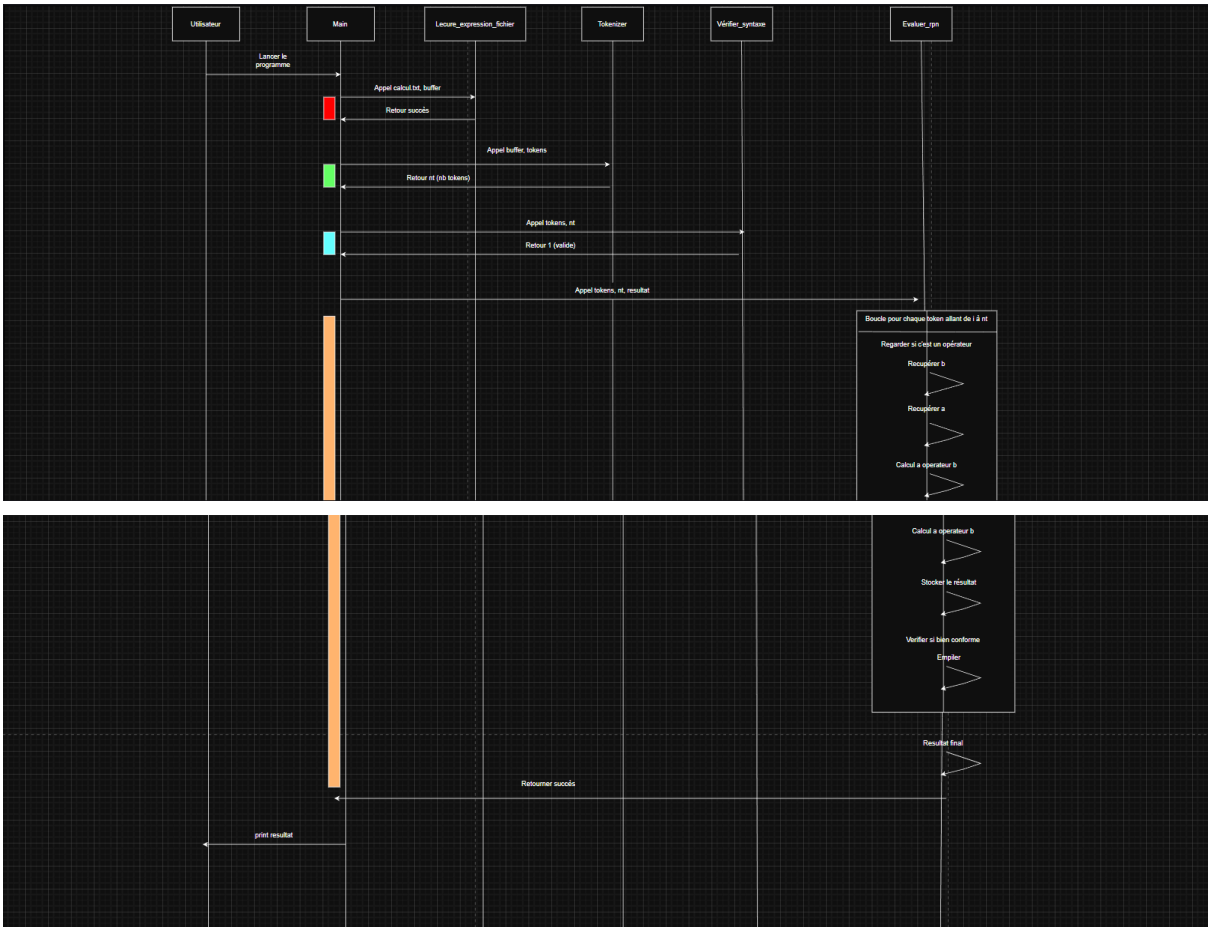
Data flow:





Sequence diagram:

https://drive.google.com/file/d/1FzRXmO_m-lh3Xka6nlx-CynTMk0rO3dr/view?usp=sharing



Analyse embold :

Analyse initiale (Le Avant) : Lors de la première analyse statique avec Embold, plusieurs "Code Issues" ont été détectées, notamment des problèmes de gestion de ressources et de clarté du code (Magic Numbers).

Erreur corrigée:

- **L'erreur :** Dans la fonction `lecture_expression`, le fichier ouvert avec `fopen` n'était pas systématiquement fermé avec `fclose` dans le cas où le fichier était vide (dans le bloc `if (!fgets...)`).
- **Pourquoi c'est une issue :** C'est une fuite de ressources. Si cette erreur se répète, le programme peut consommer tous les descripteurs de fichiers disponibles du système d'exploitation, ce qui ferait planter l'application ou empêcherait l'ouverture de nouveaux fichiers.
- **Correction :** J'ai ajouté l'instruction `fclose(file);` avant le `return 0;` dans le bloc de gestion d'erreur.

Résultat final (Le Après) : Après correction de cette fuite et ajout des commentaires de documentation, la qualité globale du code a augmenté pour atteindre un score de **4.24**, indiquant un code fiable et maintenable.

Question 4:

CODE SOURCE

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define MAXTOK 100
#define STACK_SIZE 100

/* Lit la première ligne du fichier donné, la stocke dans le buffer et
retire le saut de ligne final. */
int lecture_expression(const char *filename, char *line, size_t size) {
    FILE *file = fopen(filename, "r");
    if (!file) {
```

```
        printf("Erreur : Impossible de lire le fichier '%s'.\n",
filename);
        return 0;
    }
    if (!fgets(line, size, file)) {
        printf("Erreur : Le fichier est vide.\n");
        fclose(file);
        return 0;
    }
    line[strcspn(line, "\n")] = 0;
    fclose(file);
    return 1;
}

/* Parcourt tous les tokens pour vérifier s'ils sont soit des
opérateurs valides (+, -, *, /), soit des nombres réels. */
int verifier_syntaxe(char *tokens[], int nt) {
    for (int i = 0; i < nt; i++) {
        char *tok = tokens[i];
        if (strcmp(tok, "+") != 0 && strcmp(tok, "-") != 0 &&
            strcmp(tok, "*") != 0 && strcmp(tok, "/") != 0) {
            char *endptr;
            strtod(tok, &endptr);
            if (*endptr != '\0') {
                printf("Erreur syntaxe : Token invalide '%s'\n", tok);
                return 0;
            }
        }
    }
    return 1;
}

/* Exécute l'algorithme de la notation polonaise inverse (RPN) en
utilisant une pile pour calculer le résultat final. */
int evaluer_rpn(char *tokens[], int nt, double *resultat_final) {
    double stack[STACK_SIZE];
    int top = 0;

    for (int i = 0; i < nt; i++) {
        char *tok = tokens[i];

        if (strcmp(tok, "+") == 0 || strcmp(tok, "-") == 0 ||
            strcmp(tok, "*") == 0 || strcmp(tok, "/") == 0) {
```

```
            if (top < 2) {
                printf("Erreur : L'opérateur '%s' nécessite deux
opérandes.\n", tok);
                return 0;
            }

            double b = stack[--top];
            double a = stack[--top];
            double r = 0;

            if (strcmp(tok, "+") == 0){
                 r = a + b;
            }else if (strcmp(tok, "-") == 0){
                r = a - b;
            } else if (strcmp(tok, "*") == 0){
                r = a * b;
            }else if (strcmp(tok, "/") == 0){
                if (b == 0) {
                    printf("Erreur : Division par zéro.\n");
                    return 0;
                }
                r = a / b;
            }
            stack[top++] = r;
        } else {
            if (top >= STACK_SIZE) {
                printf("Erreur : Pile pleine (Stack Overflow).\n");
                return 0;
            }
            stack[top++] = atof(tok);
        }
    }
    if (top != 1) {
        printf("Erreur : Expression RPN invalide (il reste %d valeurs
dans la pile).\n", top);
        return 0;
    }

    *resultat_final = stack[0];
    return 1;
}
```

```c
/* Découpe la chaîne de caractères initiale en un tableau de mots
(tokens) en utilisant les espaces comme séparateurs. */
int tokenizer(char *line, char *tokens[], int max_tokens) {
    int nt = 0;
    char *t = strtok(line, " \t\n");
    while (t != NULL && nt < max_tokens) {
        tokens[nt++] = t;
        t = strtok(NULL, " \t\n");
    }
    return nt;
}

/* Affiche le résultat du calcul sur la sortie standard si le statut
est valide, ou un message d'échec sinon. */
void afficher_resultat(int status, double resultat) {
    if (status) {
        printf("Résultat : %g\n", resultat);
    } else {
        printf("Le calcul a échoué.\n");
    }
}

int main(int argc, char *argv[]) {
    char line[1024];
    char *tokens[MAXTOK];
    int nt;
    double resultat;

    char *filename = (argc > 1) ? argv[1] : "calcul.txt";

    if (!lecture_expression(filename, line, sizeof(line))) {
        return 1;
    }

    printf("Expression lue : %s\n", line);

    nt = tokenizer(line, tokens, MAXTOK);

    if (!verifier_syntaxe(tokens, nt)) {
        return 1;
    }
    int status = evaluer_rpn(tokens, nt, &resultat);
    afficher_resultat(status, resultat);
```

```
    return 0;
}
```