# RAPPORT DU PROJET IN104 –MINIWOLFRAM

Fodé Amadou CAMARA, Franck Wilson KOUASSI, Gabriel LEGROS

14 MAI 2024

PLAN DU RAPPORT

# I-) INTRODUCTION

Le projet "MiniWolfram" vise à développer une version simplifiée et accessible de ces outils de calcul puissant que sont les calculateurs Wolfram. MiniWolfram est conçu pour être un interpréteur de calculs capable d'effectuer des opérations arithmétiques de base, des évaluations de fonctions mathématiques, et de résoudre des expressions algébriques simples. Nous avons réussi à faire une calculatrice dans laquelle nous avons implémentés beaucoup de fonctions dont nous vous expliquerons le fonctionnement et les limites dans certains cas.

# II-) LES DIFFERENTS AXES DU PROJET

## 1) MANIPULATION ET ADAPTATION DES TOKENS DANS LE LEXER :

Le rôle du lexer est de transformer une chaîne de caractères en une séquence de tokens les tokens représentent des entités de type entiers ou réels, opérateurs, les variables, des parenthèses, des points, des virgules, des points virgules etc. Notre lexer traite bien les cas des parenthèses et des espaces .

-CAS DES ENTIERS : Un token pour un entier est une séquence de chiffres sans aucun séparateur, le lexeur (fichier lexical) identifie les séquences de chiffre comme des tokens d'entiers.

On avait pas eu de difficultés à ce niveau(c'était déjà bouclé pour la mi-parcours),le processus d'analyse lexicale était alors le suivant :

1) Le lexer prend en entrée une chaîne de caractères
2) Lit chaque caractère de la chaîne un par un
3) Dès qu'un caractère non numérique est rencontré, il calcule ce qu'il a lu et le met dans un token
4) Le lexer enregistre ce token( qui es de type entier et a pour valeur celle stocké après analyse lexicale) et passe au caractère suivant pour continuer l'analyse

-CAS DES REELS : L'analyse lexicale pour les réels suit un processus similaire à celui des entiers, avec des étapes supplémentaires pour gérer les parties décimales s'il rencontre une virgule ou un point-virgule (c'est donc une generalisation du cas des entiers)

1) Le lexer prend en entrée une chaîne de caractères
2) lit chaque caractère de la chaîne un par un
3) Il continue de lire les caractères tant qu'ils sont numériques. S'il rencontre un point ou une virgule il calcul ce qu'il a lu et le stocke dans un token (ce qui correspond à la partie entière du nombre)

4) Il continue la lecture (de ce qui correspond à la partie décimale du nombre)
5) Dès qu'un caractère non numérique est rencontré, il calcule et ajoute ce qu'il a lu dans le même token que le précédent (on reconstitue alors le nombre réel=partie entière +partie décimale)

-CAS DES VARIABLES :On a redéfinie les tokens car dans le code initial fourni,à chaque token était associé un type et une valeur qui est constante. Nous avons ajouté 5 chaines de caractères et 6 valeurs doubles (5 pour créer des variables et la 6ème pour les nombres complexes dans la suite) pour chaque token pour pouvoir créer des tokens de type var noté x afin de manipuler des expressions littérales.

Les 5 chaines de caractères sont : name ,name2, name3, name4 et name5

Chacun de ces name est relié respectivement aux valeurs Coeff, coeff2, coeff3, coeff4, coeff5

Le programme manipule des polynômes de variable x de degré au plus 5 de telle sorte que Le caractère 'name' reçois la var x et la valeur 'Coeff 'reçois le coefficient de x, de même la chaine de caractère 'name2' reçois la var x^2 et la valeur coeff2 reçois le coefficient de x^2 et ainsi de suite.

(Voir le code du lexer dans le fichier lexical.c)

Remarque : on ne peut manipuler que des expressions de degré inferieur ou égale à 5 ce qui est une limite de la calculatrice et un axe d'amélioration

-CAS DES COMPLEXES : Pour travailler sur les nombres complexes, dans la continuité du code on a jugé plus malin de considérer l'imaginaire pur i comme une variable (pour pouvoir utiliser le résultat sur les variables produit somme et puissance) vérifiant i^2=-1, î^3=-i, i^4=1 mais là on doit différencier les variables x et i dans le lexer (lexical.c) pour se faire on utilise la 6ème valeur attribuée au token 'estim.' qui vaut 1 si c'est i et 0 si c'est x.

## 2-) FONCTIONNEMENT DU PARSER AVEC L'ALGO DE SHUNTING-YARD

On a utilisé algorithme de Shunting-Yard pour convertir une expression infixée (où les opérateurs sont situés entre les opérandes, par exemple **3 + 4**) en une expression postfixée (notation polonaise inverse ou RPN, où les opérateurs suivent les opérandes, par exemple **3 4 +**). Cet algorithme facilite l'évaluation des expressions arithmétiques en respectant la priorité des opérateurs et les parenthèses.

On utilise comme structures de données une pile (pour les opérateurs + - * / % ) et une queue (qui est une file de sortie pour l'expression postfixé), voici les différentes étapes de la démarche de notre algorithme :

1) Lire les tokens un par un de l'expression (infixée)produite par le l'analyseur lexicale(lexer).
2) Si le token est un nombre, ajouter le nombre directement à la queue ;

3) Si le token est un opérateur, tant que l'opérateur au sommet de la pile a une priorité supérieure ou égale à l'opérateur actuel et que l'opérateur au sommet de la pile n'est pas une parenthèse gauche, transférer l'opérateur du sommet de la pile à la queue puis pousser l'opérateur actuel sur la pile.
4) Si le token  est une parenthèse gauche ( , Pousser la parenthèse gauche sur la pile
5) Si le token est une parenthèse droite) , Transférer les opérateurs du sommet de la pile à la queue jusqu'à ce qu'une parenthèse gauche soit au sommet de la pile. Enlever la parenthèse gauche de la pile
6) Après avoir lu tous les tokens, transférer les opérateurs restants de la pile à la queue

## 3-) EVALUATION DE L'EXPRESSION RPN AVEC UN ALGO BASE SUR LES PILES

L'idée est de lire les tokens de l'expression postfixée un par un, de pousser les opérandes sur une pile, et de calculer les résultats au fur et à mesure que les opérateurs sont rencontrés. Voilà en gros les différentes étapes de l'algorithme.

1) On crée et initialise une pile vide pour stocker les valeurs intermédiaires
2) On parcourt l'expression postfixée (RPN) token par token
3) Si le token est un nombre , on le pousse sur la pile
4) Si le token est un opérateur : on extrait les deux dernières valeurs de la pile. La première valeur extraite est l'opérande droite et la deuxième est l'opérande gauche
   Remarque : pour les fonctions à une variable comme COS il extrait juste une seule valeur
5) Puis applique l'opération indiquée par l'opérateur au deux opérandes extraits.
6) On pousse le résultat obtenu de l'opération sur la pile
7) une fois que tous les tokens ont été traités, la pile devrait contenir une seule valeur, qui est le résultat final de l'expression
8) On s'assure que la pile contient exactement une valeur à la fin

   REMARQUE : Pour évaluer l'expression on vérifie tout d'abord si c'est un calcul intégral en vérifiant si le dernier  Token à la fin de l'expression en RPN est INTGL, qu'on enleve et on met la fonction insérée dans l'intégrale(voir evaluation.c).
   Dans le cas contraire on remet le dernier Token qui après vérification n'est pas INTGL puis on suit les étapes citées ci-haut

## III-) FONCTIONNALITES IMLPLEMENTEES :

Vous pourrez lire toutes ces fonctionnalités sur l'interface graphique, dans ce volet je vous explique comment marchent ces fonctionnalités et quelques exemples à insérer dans l'interface graphique puis certaines limites à nos fonctionnalités ainsi que les difficultés rencontrées.

1)COS pas de difficulté ; exemple : COS(PI)

2)SIN pas de difficulté ; exemple : SIN(PI/2)

3)TAN pas de difficulté ; exemple : TAN(PI/4)

4)LOG pas de difficulté ; exemple : LOG(e)

5)EXPO REEL pas de difficulté cependant pour l'EXPO COMPLEXE on vérifie d'abord si l'argument de l'exponentiel est complexe dans ce cas on déroule le calcul (voir evaluatiion.c) ; exemple : EXP(i*PI)

5)POW OU PUISSANCE ^ pas de difficulté majeure (on a pas traité le cas un nombre à la puissance une variable ou une variable à la puissance une variable qui se ramène tous à la forme exponentielle précédente) ; exemple : 0,1^2 , 4^0.5

6)FTGAMMA la Fonction gamma d'Euler qui joue le rôle de factorielle généralisée aussi car FTGAMMA(n)=factorielle de n-1 ; exemple : FTGAMMA(4)=6 (qui est égale à la factorielle de 3)

7)ZETA la fonction zêta de Riemann ; exemple : ZETA(2)=PI^2 /6

8)LES CONSTANTES PI, e, GAMMA

9)VARIABLLE x ,a , b et c(voir le VOLET MANIPULATION ET ADAPTATION DES TOKENS) ,on pourra développer réduire et simplifier des expressions littérales. Exemple : (1-x)(1+x)=1-x^2

Remarque : les variables a, b, c et x ne devront pas être mélangé dans une même expression. A la base on utilisait juste la variable x ,on a créé les variables a, b et c pour stocker les solutions des équations résolues avec SOLVE.

110)ASSIGNED qui permet de donner une valeur à une variable. Exemple : ASSIGNED(x=4)

Remarque : Le calcul de la variable de x auquel on "ASSIGNED" une valeur ne marche qu'avec les fonctions polynomiales. On a pas traité le cas pour les autres fonctions.

11)LES NOMBRES COMPLEXES Pour travailler sur les nombres complexes, dans la continuité du code on a jugé plus malin de considérer l'imaginaire pur i comme une variable (pour pouvoir utiliser le résultat sur les variables produit somme et puissance) VOIR LE VOLET MANIPULATION ET ADAPTATION DES TOKENS

On pourra faire la somme, le produit, la division et la puissance entière à tout ordre des nombres complexe ; exemple : (1-i)(1+i) ;(1+i)^2 ; (1+i) /(1-i)

12)CONJ donnera le conjugué de tout nombre complexe ; CONJ(-i) ; CONJ(5+6i)

13)SOLVE qui permet de résoudre les équations linéaires et quadratiques ; exemple : SOLVE (4x-2=7 ;x) ; SOLVE (2x-x^2+4=10 ;a;b)

Remarque : Le second membre dans ces équations doit être nécessairement une constante

14)INTGL qui est l'intégrale de fonctions, pas de difficultés majeures, en revanche on ne mélange les intégrales avec les autres opérations comme expliqué dans le volet évaluation, par exemple attention à INTGL(1/(1+X^2) ;0 ;1) + 13

Faut juste calculer INTGL(1/(1+X^2) ;0 ;1 puis ajouter +13 au résultat obtenu

Remarque : On est sur un calcul d'intervalle définie sur un segment, pas de primitives !

## IV-) INTERFACE GRAPHIQUE

On a utilisé la bibliothèque GTK+ pour gérer les éléments de l'interface graphique en en effectuant des recherches sur comment l'installer et l'adapter à notre code, voici les commandes pour télécharger et installer GTK+ afin de pouvoir compiler notre code qui renverra sur une interface graphique ou vous pourrez trouver toutes les fonctionnalités de notre calculatrice. Les notations sont les mêmes que dans le volet FONCTIONNALITES IMPLEMENTEES

Sous linux : **`sudo apt-get install libgtk-3-dev`**

Sous Windows : Vous pouvez utiliser MSYS2

## V-) COMMENT FAIRE MARCHER NOTRE CODE(MAKEFILE)

Afin de simplifier et d'automatiser le processus d'exécution du Miniwolfram, on utilise un fichier makefile qui exécutera l'ensemble des fichiers nécessaires à la compilation du programme.

Voici les commandes à effectuer ;

make

./main.x

Ça vous renvoie sur l'interface graphique ou vous pourrez facilement interagir avec la calculatrice.

## VI-) AXES D'AMELIORATIONS ET CONCLUSION

Pour conclure, dans le cadre du module IN104, nous avons développé un programme permettant d'effectuer des calculs à la manière d'une calculatrice scientifique. Notre calculatrice peut être amélioré pour traités beaucoup plus de fonctionnalités tels que la résolution d'équations différentielles, les calculs de dérivées, afficher des graphes de fonctions etc.