Partie LEXICALE : Récupération d'une chaîne de caractères en entrée et renvoie un tableau de Jetons.

Benjamin JACQUOT

Arthur KINDELBERGER

Les fonctions supportées sont les suivantes :

- +
- -
- *
- /
- Puissance : ^
- SIN et sin
- COS et cos
- TAN et tan
- EXP et exp
- LOG et log
- SQRT et sqrt
- Variable : X et x
- Pour la fonction « -x » faire : « 0-x »
- Les réels

Dans la partie lexicale, on commence à analyser au premier caractère autre qu'un espace.

Par la suite, on va analyser chaque caractère avec un switch case et modifier le tableau de jetons en fonction. Lorsque l'on rencontre un chiffre, on va continuer de regarder ce qui suit en gardant les chiffres précédents en gérant aussi la virgule « . » et en compilant à la fin sous forme d'un réel. On fait la même chose lorsqu'on rencontre un caractère, on regarde tant qu'il y a un caractère puis on compacte tout cela et compare avec les fonctions connues.

Ainsi les erreurs traitées sont : les fonctions inconnues, les doubles virgules, les caractères inconnus comme : « # ».

Partie Analyse Syntaxique : Conversion de la chaîne de tokens (fournie par l'analyseur lexical) en arbre syntaxique de données.

Drobyshevski Nikolaï
Trelcat Jean

Le parseur construit un arbre syntaxique à partir des tokens fournis par le module lexical. Il reconnaît :

- Les nombres réels
- Les variables (ici x)
- Les fonctions mathématiques (par expl sin, cos, exp…)
- Les opérateurs classiques (+, -, *, /, ^)
- Les parenthèses

Le fichier inclue également des fonctions pratiques pour voir le détail seul de l'analyseur syntaxique, avec un affichage de l'arbre et un nettoyage de la mémoire.

Fonctions principales :

Node *analyserSyntaxe(typejeton tokens_array[])
Point d'entrée de l'analyse syntaxique.
Crée l'arbre syntaxique à partir du tableau de jetons

Node *expr()
Gère les opérateurs de niveau supérieur : addition et soustraction

Node *term()
Gère la multiplication et la division

Node *factor()
Gère les éléments de base :

- nombres réels
- variables
- fonctions
- expressions entre parenthèses

Node *power()
Gère la fonction puissance (opérateur ^)

Node *create_node()
Crée simplement un nœud pour l'arbre syntaxique

Fonctions de lecture pratique de l'arbre pour debug :

void print_ast(Node *root, int level, char branch)
Affiche l'arbre de manière indentée
Utilise des symboles pour représenter les liens entre opérateurs/fonctions et variables/réels

const char *token_type_to_str(typelexem t)
Convertit un type de jeton en chaîne de caractères pour le représenter dans l'arbre

Sur demande lors de l'évaluation : explication de la partie Test :
Un fichier test est à disposition dans le dossier :

A la suite des run_test, un tableau de jetons est écrit

Rapport de la Partie Évaluation

Auteur : Mathias Kalisz Pereira

Introduction

Ce document décrit ma contribution à la réalisation du projet Grapheur. Mon rôle était de développer la partie « évaluateur » du programme, c'est-à-dire l'évaluation numérique d'une expression mathématique à partir d'un arbre syntaxique abstrait (AST). Cette évaluation est nécessaire pour afficher les valeurs de la fonction au terminal et, à terme, permettre son tracé graphique.

Objectif de ma partie

- Recevoir un arbre syntaxique de type `Node*` construit à partir d'une expression mathématique.
- Évaluer cette expression pour une valeur donnée de `x`.
- Gérer les opérations classiques (addition, multiplication, division, etc.) ainsi que les fonctions mathématiques (sin, cos, log, exp...).
- Assurer la stabilité du programme : détection des divisions par zéro ou des log de valeurs négatives.

Structure du fichier `evaluateur.c`

Fonction principale :

double evaluate_expression(Node *root, double x_value);

Étapes de la fonction `evaluate_expression`

Cette fonction est récursive. Elle parcourt l'arbre en profondeur et évalue chaque nœud selon son type.

1. Cas de base : noeud nul

if (root == NULL) {

```
    fprintf(stderr, "Erreur : nœud NULL\n");
    return 0.0;
}
```

Ce test protège contre **une tentative de lire une valeur à partir d'un nœud inexistant**.

2. Cas des feuilles : `REEL` ou `VARIABLE`

```
if (root->type == REEL) return root->value;
if (root->type == VARIABLE) return x_value;
```

- Un nœud `REEL` contient directement une constante.
- Un nœud `VARIABLE` correspond à la variable `x`.

3. Cas des opérateurs binaires

On évalue les sous-arbres gauche et droit, puis on combine leurs résultats :

```
case PLUS:
    return evaluate_expression(root->fg, x_value) + evaluate_expression(root->fd, x_value);
```

Idem pour `MOINS`, `FOIS`, `DIV`, `PUIS` avec des précautions :

- Division : on vérifie que le diviseur n'est pas nul.
- Puissance : utilisée via `pow(...)` de `math.h`

4. Cas des fonction à un seul argument: `SIN`, `COS`, `EXP`, `LOG`, `TAN`

Ces fonctions prennent un seul argument à gauche (sous-arbre `fg`) :

```
case SIN:
    return sin(evaluate_expression(root->fg, x_value));
```

- Pour `LOG`, on vérifie que l'argument est strictement positif.

5. Cas par défaut : erreur

Si le type n'est pas reconnu :

```
fprintf(stderr, "Erreur : type de nœud non géré (%d)\n", root->type);
```

Intégration dans le projet

Une fois la fonction `evaluate_expression` fonctionnelle, je l'ai intégrée dans le fichier `main.c`, juste après l'appel à `analyserSyntaxe(T)`.

Les résultats sont affichés dans le terminal pour des valeurs de x contenues dans un intervalle précis et avec un pas précis (1/le nombre de valeurs totales de x). Extrait typique :

```
for (double x = a; x <= b; x += 1/nbx) {
   double y = evaluate_expression(arbre, x);
   printf("f(%.1f) = %.4f\n", x, y);
}
```

Conclusion

La partie évaluation est pleinement fonctionnelle et intégrée au projet. Elle permet l’évaluation précise d’expressions mathématiques avec prise en charge des cas particuliers. Ce travail constitue une base solide pour un tracé graphique fiable.

Résumé du fonctionnement de l'algorithme – Grapheur SDL2

Résumé du fonctionnement de l'algorithme Grapheur SDL2

Objectif du programme :

Ce programme permet d'afficher graphiquement une fonction mathématique saisie par l'utilisateur, à l'aide de la bibliothèque SDL2.

Fonctionnement global :

1. Initialisation de SDL2 :

- Création d'une fenêtre (SDL_Window) et d'un moteur de rendu (SDL_Renderer).

- Chargement de polices et d'images nécessaires à l'interface graphique.

2. Entrée utilisateur :

- L'utilisateur entre une fonction mathématique sous forme de texte.

- Le programme la passe à un l'analyseur lexical et syntaxique.

3. Evaluation et dessin :

- Si la syntaxe est correcte, la fonction passe dans l'évaluateur et on récupère un tableau de couple x, f(x) - Les points sont transformés en coordonnées écran puis dessines avec SDL (axes, courbe, etc.).

4. Affichage des erreurs :

- En cas d'erreur de syntaxe ou d'évaluation, un message est affiché en rouge.

Details techniques :

Le fichier main.c orchestre la logique principale.

Le fichier gestionGraphique.c fournit les fonctions pour :

Initialiser SDL,

-Créer la fenêtre et le renderer,

-Charger les images/typos,

-Dessiner avec des formes personnalisées (ex. rectangles arrondis).