Phase n°2

Parser d'expression mathématique avec fonctions

À chaque phase, vous devez rendre

- Un *rapport* (avec une vue générale des choix que vous avez eu à faire et des difficultés rencontrées)
- Votre *code* (qui doit compiler, et de préférence être commenté)
- Un système de build (CMake, Makefile, Meson)

Vous vous baserez sur votre implémentation de la phase précédente (ou si besoin du corrigé indicatif). Les ajouts demandés sont

- La gestion de fonctions prédéfinies unaires
- La gestion de fonctions prédéfinies *n*-aires
- La gestion de fonctions prédéfinies à nombre variable d'arguments

Précédemment

Vous disposez désormais d'un évaluateur qui peut

- Séparer une chaîne en tokens
- Réorganiser ces tokens vers une représentation exploitable
- Évaluer cette représentation exploitable

De plus, vos Tokens devraient être polymorphiques, et ce polymorphisme devrait être implémenté par héritage. Le code corrigé qui vous est fourni ne respecte pas ces consignes, volontairement, afin de vous pousser à utiliser votre propre code (même si vous le trouvez moche !).

Vous trouverez tout de même, en annexe, quelques bouts de code que nous considérons bons, à titre indicatif et à des fins d'exemple et de compréhension.

> **Tip**
> Créez une fonction qui retourne une lambda ayant pour paramètre un entier et qui retourne cet entier ajouté à une constante (quelconque *mais* interne à la fonction). Récupérez la lambda et utilisez-la sur un entier. Vérifiez que la constante, bien qu'appartenant à la fonction, à été "mémorisée" par dans lambda retournée.

Fonctions unaires

Votre programme devra au moins gérer les fonctions

- $cos(x)$
- $sin(x)$
- $tan(x)$
- $log(x)$
- $exp(x)$
- $sqrt(x)$

Pour cela il vous faudra étendre votre hiérarchie de Tokens, votre lexer et votre parser.

Fonctions *n*-aires

Étendez votre programme afin de gérer également les fonctions *n*-aires, dont vous présenterez le fonctionnement via les fonctions

- $pow(x, y)$
- $hypot(a, b)$
- $lerp(x, a, b)$

tels que l'on obtienne

```
pow(2, 3)
hypot(1, 1)
lerp(0.5, 0, 1)
lerp(0.3, 0, 10)
8
1.41421
0.5
3
```

Fonctions à nombre variable d'arguments

Finalement, il vous faut étendre votre programme pour pouvoir gérer des fonctions dont le nombre d'arguments *n'est pas connu à l'avance*. Réalisez la fonction

$$polynome(k, a_0, ..., a_k, x) = a_0 + a_1 x + ... + a_k x^k$$

Voici quelques exemples du comportement attendu :

```
polynome(0, 2)
polynome(2, 1, 1, 1, 2)
2
7
```

> **Note**
>
> Vous n'êtes pas censés changer l'ordre de ces paramètres. Vous pouvez le faire pour avancer et **en le précisant dans le rapport**, mais ça sera une pénalité.

Annexes

Fermetures et pointeurs de fonctions

Function objects

Correspond au type `std::function<T>`, avec `T` un type de fonction, comme

```
bool f(double a, double b) {
    return a == b ;
}

using Predicate =
    std::function<bool(double, double)> ;

const Predicate p = f ;
assert(p(3, 4) == f(3, 4)) ;
```

On pourrait voir ça comme un pointeur de fonction avancé (et potentiellement plus lent). Les détails sont sans grande importance et il faut surtout retenir que n'importe quel object function *est un objet C++ valide*.

Lambda functions

Il est possible de définir des fonctions associées à un environnement (une fermeture) en C++. Un exemple pourrait être

```
/// ...
const uint value = 3u ;
const auto addValue =
    [&value](uint p) -> uint {
        return p + value ;
    } ;
// ...
addValue(2) ; // → 5
// ...
```

Leur syntaxe est, en général, complexe et sort du cadre de ces annexes. Vous pouvez lire la page cp-preference associée pour plus d'informations.