Phase n°3

Parser d'expression mathématique avec curryfication/spécialisation

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 curryfication de fonctions
- La spécialisation de fonctions selon certains paramètres

Prenez bien en compte les deux parties de ce TP avant de commencer votre implémentation.

Précédemment

Vous disposez maintenant d'une calculatrice qui propose un jeu de fonctions prédéfinies à $2, 3,$ ou n arguments. Si tout s'est bien déroulé ces fonctions devraient être stockées dans une table de type `map<string, Function>` (avec `Function` le type général de vos fonctions). Il est même possible que vous ayez réussi à gérer tous les cas avec la même stratégie. Comme la dernière fois, le code fourni ne propose pas tout ça, mais des éléments de réponse se trouvent en annexe.

Fonctions curryfiées

À l'instar des languages fonctionnels, votre programme va devoir gérer la création de nouvelles fonctions à partir d'une fonction prédéfinie et de quelques uns de ses premier paramètres. En OCaml, par exemple

```
# let f a b c = a + b + c ;;
val f : int -> int -> int -> int = <fun>
# let g = f 3 3 ;;
val g : int -> int = <fun>
# g 5 ;;
- : int = 11
```

Autrement dit on va transformer une fonction de n paramètres en fonction à $n - p$ paramètres, où p est le nombre partiels de paramètres. *Attention, on ne vous demande SURTOUT PAS de reproduire la syntaxe sans parenthèse du OCaml, ça n'est qu'un exemple !* Dans notre cas, on peut prendre comme exemple la fonction d'interpolation linéaire $lerp(x, a, b)$, qu'on peut utiliser avec une valeur fixe de x. On veut donc pouvoir définir une fonction $speciallerp(a, b)$ comme par exemple

```
x = 0.7 ;
speciallerp = lerp(x) ;
a = 3 ;
b = 4 ;
speciallerp(a, b)
```

qui doit donner 3.7. Si vous aviez défini la fonction comme $lerp(a, b, x)$, vous devriez pouvoir faire, de manière similaire, *speciallerp = lerp(a, b)* (i.e. on doit pouvoir spécifier un nombre indéfini d'arguments).

Fonctions spécialisées

L'ajout précédent est limitant par rapport à l'ordre des paramètres. Il serait intéressant de pouvoir définir des paramètres arbitrairement. Par exemple $pow(x, y)$ possède une spécialisation intéressante $square(x) = pow(x, 2)$, qui ne peut être gérée par simple curryfication. Votre programme pourra donc faire appel à la syntaxe

```
square = pow(_1, 2) ;
square(3)
```

qui doit donner 9. On vous demande donc de reconnaître des lexèmes de la forme `_[0-9]+`, que l'on appelera des *placeholders* ordonnés. Le comportement est similaire à ce que ferait `std::bind` (ici) pour les object functions.

Annexes

STL

Cette partie ne s'appuie plus réellement sur de nouveaux concepts. Les objects de la STL qui pourraient vous intéresser, mais que vous ne devriez pas avoir à utiliser, sont

- `std::bind`
- `std::placeholder`

Code

```
// Just the same here. I will add some code from my second phase here, but
// I'd rather know what you think about it in general [WIP].
```