Sujets de mini-projet 2016

Progammation C++

VAYSSADE Jehan-Antoine

Ajouter la gestion de fonctions n'ayant qu'un seul argument

Description de votre travail et de vos choix de programmation.

(récapitulatif du TP1)

- Ma version de base permettait déjà d'avoir les fonctions unaires puisque le contexte d'appel est identique à la notion d'opérateur, on peut donc ajouter certaines fonctions de cette façon et bénéficier d'optimisation (pas de chargement de Context / Mémoire comme fait plus loin).

- Pour ce faire j'ai simplement ajouté dans le Tokenizer à la détection si token= [[:alpha :]]+
  - si un opérateur unaire existe on push le pointeur (dont sin/cos/...)
  - si un opérateur binaire existe on push le pointeur
  - si un prototype de fonction existe et que le caractère suivant est '('
    - on push le warpper ExprCall avec le pointeur de la fonction et le block de parenthèses associé

- J'ai ajouté la notion de ExprCall pour stocker par pointeur/référence et binder la mémoire de la fonction au Context appelant

- J'ai ajouté la notion de Mémoire locale et de Context
  - cela permet de binder des fonctions et des variables localement à une fonction lors de l'appelle afin de permettre la résolution de nom dans les contextes de profondeurs supérieurs

- Ajout de la classe Function
  - la première permet de garder en mémoire l'expression sous forme de string
    - et donc de permettre de construire et d'ajouter les fonctions utilisateurs
  - on donne un vecteur de paramètres (x,y,z)
  - à l'appel on split le bloque de parenthèses avec ',' et on réinitialise les variables de la mémoire avec les valeurs des expressions évaluées des paramètres.
  - Le nombre de paramètres est connu à la définition

- Ajout de la classe VariadicFunction dérivée de la classe Function
  - elle prend un paramètre spécial ‘...’
  - et on construit dynamiquement l’expression à évaluer en fonction des paramètres donnés, donc calcule des indices de début et de fin des paramètre variables.
  - Cela ne permet pas de les définir à l’exécution
  - j’ai donc implémenté la notion de fonctions récursives
  - et ai commencé à ajouter le mappage de fonctions en tant que paramètre pour construire des fonctions par construction par exemple (sum(transforme(iseven, 1, 2, 3, 4, …))
  - le nombre de paramètres est connu à l’évaluation

- Ajout de la détection des fonctions récursives et donc j’ai choisi de définir une fonction interne dans la mémoire de la fonction récursive pour ‘kill’ la récursivité et donner la valeur terminale
  - il faut donc trouver une fonction mathématique qui donne à la fois la valeur de retour et la continuité de l’algorithme donc il a été nécessaire de rajouter les opérations de comparaisons par exemple y/(x<1), lorsque x>=1 la valeur est +inf ce qui permet de continuer et retourne y sinon, le cas où y=0 le résultat vaut -nan et est donc géré en interne pour mapper la valeur 0

- On a donc plusieurs méthodes de définition de fonctions afin de fournir soit une implémentation performante, soit une implémentation « runtime »

this software provide an interface for basic mathematics expressions
you must write your expression in infix notation and the evualuation will be printed in the next line.
otherwise an error is printed with small information about it
please note that all evaluated strings have their whitespaces removed

-this software include the following maths functions/operators:
  binary: + - * / ^ >< <> & | % \ >> << > < ::
  unary: cos sin tan exp log sqrt acos asin atan fact fib ! abs floor ceil round gamma lgamma fint fract ~
  const: false true zero one pi phi e inf nan
-you could also add your own function in runtime by the following syntaxe:
  functionName(p1, p2, ...) = expr as like (logb, pow, hypot, lerp)
  recursive function called are now supported make sure the terminal value will be reached
  variadic function called are now supported (sum, prod, polynome)
-you can define/erase variables by the following syntaxe:
  varName = expr
-you can access the last result by the variable "ans"